COMPUTING PRACTICAL I
Part I: Functional Programming in Haskell

Tony Field

Course Outline

1. Expressions and basic types
2. Functions
3. List processing
4. Higher-order functions
5. User-defined types
6. Type classes
7. I/O and Monads

Suggested Books:

- *Haskell: The Craft of Functional Programming*
  Simon Thompson (3rd ed.)
  Addison Wesley 2011

- *Thinking Functionally with Haskell*
  Richard Bird
  Cambridge University Press 2014

- *Real World Haskell*
  Bryan O'Sullivan, John Goerzen and Donald Bruce Stewart
  O'Reilly Media 2008

- *The Haskell School of Expression*
  Paul Hudak, 2000
  Cambridge University Press
- *Functional Programming*
  Tony Field and Peter Harrison
  Addison Wesley 1988
- Also, take a look at the Haskell web site: http://haskell.org/

1. Expressions and basic types

- Mathematical expressions are already familiar to you. GHCi will evaluate an expression if you enter it after the prompt, e.g.:

```
Prelude> 2501
2501
Prelude> 2 - 3 * 4
-10
Prelude> sin (24.9) + sin 24.9
-0.4612914118547913
```

- `+`, `-`, `*`, `/` etc. are called *operators* or *infix functions*

- `sin`, `cos`, etc. are *prefix* functions (we usually drop the word "prefix")

- `x` means the same as `(x)` and juxtaposition means function *application*, e.g. `sin (24.9)` or `sin 24.9` both *apply* `sin` to `24.9`

- A Haskell *type* represents a set of "like" values; every value (hence every expression) "has a type"

- The type `Int` is the set of supported integers ("whole" numbers), like 3, 4, -10 etc.; for now, think "4 has type `Int`", but see later

- `Int`s occupy a fixed amount of space, typically 32 or 64 bits; the smallest 64-bit integer is `-9223372036854775808` and the largest is `+9223372036854775807`

- Type `Integer` is the set of arbitrary-precision integers (uses as many digits as necessary, but it's SLOW)

- `Float` and `Double` are the types of *single-* and *double-precision* "floating-point" numbers – a finite subset of the reals, e.g. 2.75, -123.64 etc.; the largest `Double` is $1.7976931348623157 \times 10^{308}$

- Only a subset of the real numbers can be represented so floating-point arithmetic is not always accurate

Bracketing

- If necessary, brackets (*parentheses*) can be used to get the right meaning. For example

```
2 - 3 * 4 / sin 24.9 * pi
```

is bracketed implicitly by Haskell as

```
(2 - (((3 * 4) / (sin 24.9)) * pi))
```

because:

  - '`*`' and '`/`' have higher *precedence* than '`-`'
  - '`*`' and '`/`' are *left-associative*
  - Prefix function application has higher precedence than infix function application

- We can put brackets where we want to make the meaning clear

Qualified Expressions

- We can also name values and use the name instead of the value
- This can be done in Haskell using a `let` expression, e.g. instead of `24.9 * cos 1.175` we could equally write

```
let f = 24.9 in f * cos 1.175
let f = 24.9 ; theta = 1.175 in f * cos theta
let g = cos in 24.9 * g 1.175
```

  All three produce the same *value* 9.600022533036805

- `f, g` and `theta` are called 'variables' or 'identifiers'
- `let` expressions are sometimes called *qualified* expressions; the bits before the '`in`' are the *qualifiers* and the final expression is called the *resultant*

Characters and Truth Values

- Two more predefined base types:
  - Characters (`Char`), e.g. `'a', 'b', 'A', '3', '!'` etc.
  - Truth values (`Bool`), which are either `True` or `False`
- Some Haskell operators work on `Bools`, including *and* (`&&`), *or* (`||`) and *not* (`not`); for example

```
Prelude> not False
True
Prelude> False && (not True)
False
Prelude> not (True && (False || not True))
True
```

- Values of type `Bool` are produced by *comparison* operators:

  `==` Equal, as in `5 == (4 + 1)`

  `/=` Not equal, as in `'a' /= 'p'`

  `>` Greater than, as in `12 > 9`

  `<` Less than, as in `(12.8 * 9) < 2`

  `<=` Less than or equal, as in `5 <= 6`

  `>=` Greater than or equal, as in `44 >= 45`

- So, we can put things together, e.g.

```
Prelude> (1 < 9) || ((4 == 7) && ('a' > 'm'))
True
Prelude> 3 > (9 - 2) || 4 / 5 <= 0.7
False
```

- Some predefined prefix functions also generate `Bool`s, e.g.:

  `even` – Returns `True` iff a given number is even

  `odd` – Returns `True` iff a given number is odd

  `isDigit` – Returns `True` iff a given character is one of `'0'`...`'9'`

  `isUpper` – Returns `True` iff a given character is upper case (`'A'`...`'Z'`)

- For example (note that `isUpper` and `isDigit` are defined in module `Char`):

```
Prelude> :m +Data.Char
Prelude Data.Char> isDigit '*' || isUpper 'n'
False
Prelude Data.Char> not (odd 7 && even 11)
True
```

Conditionals

- *Conditional* expressions are of the form

```
if P then Q else R
```

  where `P`, `Q` and `R` are expressions

- `P` is a *predicate*, i.e. an expression of type `Bool`; the types of `Q` and `R` must be the *same*

- if `P` evaluates to `True` then the overall result is `Q`; if it evaluates to `False` then the result is `R`, e.g.

```
Prelude> if False then 5 - 3 * 4 else 2
2
Prelude> let p = 'a' > 'z' in if p then True else False
False
```

  $\boxed{?}$ Can you simplify `if p then True else False`?

Aside: type classes and contexts

- What does GHC think the type of `3` is? We can ask it:

```
Prelude> :type 3    (or Prelude> :t 3)
3 :: Num t => t
Prelude> :t (3, 3)
(3,3) :: (Num t, Num t1) => (t1, t)
```

- `Num` is a *type class* – think of `Num` as the set of types (of things) that can be added, multiplied, negated etc.

- The *literal* `3` can thus be interpreted as having any numeric type that supports basic arithmetic on numbers, e.g. `Int`, `Integer`, `Float`, `Double` etc.

- `Num t => ..` is called a *context* and `Num t => t` means "any type `t` that's a *member* type of *class* `Num`" – much more later...

Tuples

- Sometimes it is convenient to be able to group a fixed number of values together, possibly of different types, e.g.
  - Triples of `Double` for representing vectors in three dimensions
  - Triples of `Int` for representing birth dates (day, month and year)
  - Pairs comprising a `Char` and an `Int` for representing chess board positions
- We can build such *tuples* by enclosing the required components in parentheses, e.g.
  - `(1.0, 0.0, 0.0)` might represent the unit vector i (in three dimensions)
  - `(3, 4, 1999)` might represent 3^{rd} March 1999
  - `('d', 5)` might represent position $d5$ on a chess board

- Tuple *types* are written using the same bracketing syntax as the tuple values

```
Prelude> (1, 5, 7)
(1, 5, 7)
Prelude> :t ('c', (True, False))
('c', (True, False)) :: (Char, (Bool, Bool))
Prelude> :t (6, 'a', True)
(6, 'a', True) :: Num t => (t, Char, Bool)
```

- Remark: as we've seen, a *one-tuple* isn't really a tuple at all, so `(5)` is the same as `5` and `Int` is the same as `(Int)`

- There is, however a *zero-tuple*, `()`, which is equivalent to `void` in some other languages; the type of `()` is `()` – the *unit type*

Lists

- Lists are sequences of objects; list *constants* are written using square brackets, with `[]` being the empty list
- The same square bracket notation is used for types, e.g.

```
Prelude> []
[]
Prelude> [False,True,False]
[False,True,False]
Prelude> :t [False,True,False]
[False,True,False] :: [Bool]
Prelude> :t [(80, True)]
[(80, True)] :: Num t => [(t, Bool)]
```

? What is the type of `[]`?

- Lists should not be confused with sets in mathematics, e.g.
  - The order in which the elements appear is important (lists can be *indexed* in a meaningful way)
  - Values may occur more than once
- Lists can be arbitrarily long (even infinite!) but the elements must have the *same* type, so `[True, 2, 'u']` is invalid ("*type error*")
- Compare this with tuples where the elements can be of mixed type

$\boxed{?}$ How come tuple elements can be of mixed type, whereas list elements must all have the same type?

Strings

- Lists of characters (i.e. `[Char]`) are called *strings* (type `String`) and can be
  written by enclosing the characters in double quotation marks, e.g.

```
Prelude> ['h', 'a', 'n', 'd', 'b', 'a', 'g']
"handbag"
Prelude> "handbag"
"handbag"
Prelude> :t "handbag"
"handbag" :: String
```

? How is `'k'` different from `"k"`?

Arithmetic Sequences

- The special form `[a,b..c]` builds the list of numbers `[a, a+(b-a), a+2(b-a), ...]` and so on until the value `c` is exceeded, e.g.

```
Prelude> [1..5]
[1,2,3,4,5]
Prelude> [2,4..11]
[2,4,6,8,10]
Prelude> [10,9..0]
[10,9,8,7,6,5,4,3,2,1,0]
Prelude> [0,0.5..4]
[0.0,0.5,1.0,1.5,2.0,2.5,3.0,3.5,4.0]
```

- Note: sequences work with `Int`s, `Float`s etc., and also `Char`s, `Bool`s etc. – try them out

List Comprehensions

- A list *comprehension* takes the form

```
[e | x1 <- g1, ..., xm <- gm, P1, ..., Pn]
```

- which is read "the list of all `e` (which may refer to any or all of the `xi`) where `x1` comes from list `g1`, ..., `xm` comes from list `gm`, and where `P1`, ..., `Pn` are all `True`"

`xi` is a variable ($1 \leq$ `i` $\leq$ `m`)

`gi` is a *generator* list ($1 \leq$ `i` – the `xi` are *bound* from left to right, one element at a time $\leq$ `m`)

`Pj` is a predicate ($1 \leq$ `j` $\leq$ `n`)

`e` is an expression, possibly involving the `xi`

`m` + `n` > 0

- The target variable of a generator can only be used to the *right* of the generator and may optionally appear to the *left* of the '`|`'
- The terms after the '`|`' can appear in any order, subject to the above

```
Prelude> [x^2 | x <- [1..10], even x]
[4,16,36,64,100]
Prelude>  [x | even x,  x <- [1..10]]
ERROR: Undefined variable "x"
Prelude> ['a' | True]
"a"
Prelude> [x+y | x <- [1..3], y <- [1..3]]
[2,3,4,3,4,5,4,5,6]
Prelude> [(x, y) | x <- [1..3], y <- [1..x]]
[(1,1),(2,1),(2,2),(3,1),(3,2),(3,3)]
```

- Note the *left to right* order in which the variables are bound

- The operators `==`, `/=`, `>`, `<`,`>=`, `<=` are
  defined on lists in an obvious way (we'll see exactly how later), e.g.

```
Prelude> [1, 1] == [1]
False
Prelude> [True, False] == [True, False]
True
Prelude> "False" /= "False"
False
Prelude> [1, 7, 9] < [2, 5, 8]
True
Prelude> "big" < "bigger"
True
Prelude> "big" < "big"
False
```

2. Functions

- Programming is all about the packaging and subsequent use of computational "building blocks" of varying size and complexity
- In Haskell, the building blocks are *functions*; you have already seen some *built-in* functions like `+, *, div, sqrt, cos` etc. but we can define our own
- A function `f` is a rule for associating each element of a source type `A` with a unique member of a target type `B` (cf domain and range in mathematics); we express this thus: `f :: A -> B`
- `f` is said to "take an argument" (or "have a parameter") of type `A` and "return a result" of type `B`
- If the function takes several arguments their types are listed in sequence, e.g. `g :: A -> B -> C -> D`

- We say what the function does, using one or more *rules* (sometimes called *equations*)
- A rule has a *left-hand side* which lists the argument(s) and a *right-hand side* which is an expression
- The rule looks like a conventional mathematical function definition except that we omit brackets around the argument(s), e.g.

```
successor :: Int -> Int
successor x
  = x + 1
```

(Note: this is similar to the built-in function `succ` which has a more general type)

- Observe that function names must begin with a lower-case letter

- Some more examples...

```
magnitude :: (Float, Float) -> Float
magnitude (x, y)
  = sqrt (x^2 + y^2)

isUpper :: Char -> Bool
isUpper ch
  = ch >= 'A' && ch <= 'Z'

even :: Int -> Bool
even x
  = x `mod` 2 == 0
```

- Note: we DON'T write `if x `mod` 2 == 0 then True else False`

- Note: Sometimes there are constraints on the values a function can take as arguments, e.g. the function to compute $\log x$ requires that $x > 0$

- Ideally, we should prevent function calls with invalid arguments (see later)

- If we don't do that we should at least state the *precondition* (a predicate) which must be true for the function to work as defined

- We'll use Haskell *comments* to specify preconditions; comments are simply annotations designed to help the reader – they are *not* executed

- A Haskell comment is prefixed with `--`, e.g.:

```
log :: Float -> Float
-- Pre: x > 0
log x = ...
```

- Once we have defined a function, we can *apply* it to given argument(s) provided the argument(s) have the right type

- The application of function `f` to an argument `a` is done by the *juxtaposition* `f a`, e.g.

```
*Main> successor 569
570
*Main> even 15
False
*Main> even (successor 19)
True
```

- However, `successor 'b'` is a *type error* (note that badly typed programs cannot be executed)

Guarded Rules

- Rules can contain one or more *guards*

  of the form `guard = expression` – a generalisation of conditionals

```
difference :: Float -> Float -> Float
difference x y
   | x >= y    = x - y
   | otherwise = y - x
```

(alternatively `difference x y = abs (x - y)`)

```
signum :: Int -> Int
signum n
   | n < 0     = -1
   | n == 0    = 0
   | otherwise = 1
```

- Note the layout: e.g. for `signum` we can lay out the clauses any way we want so long as they *all* lie textually to the right of the '`s`'
- But line them up anyway!
- Guards are tested in sequence from top to bottom:
  - If the guard condition is `True` the expression on the right of the `=` is evaluated
  - If it is `False` we proceed to the next guard
  - If we run out of guards we proceed to the next rule for the same function, *if there is one*
  - If we run out of rules we get an error (the function must be *partial* in that case)

? What is the definition of `otherwise`?!

Local Definitions

- In the same way that `let` expressions introduce definitions local to an expression, `where` *clauses* introduce definitions local to a rule

- This is useful for breaking a function down into simpler named components, e.g.

```
turns :: Float -> Float -> Float -> Float
turns start end r
  = totalDistance / distancePerTurn
    where
      totalDistance   = kmToMetres * (end - start)
      distancePerTurn = 2 * pi * r
      kmToMetres      = 1000
```

- Note that the `where` must be to the right of the left-hand side

- A `where` clause can also avoid replication and redundant computation, e.g.

```
normalise :: (Float, Float) -> (Float, Float)
normalise (x, y)
  = (x / sqrt (x^2 + y^2), y / sqrt (x^2 + y^2))
```

The common subexpression can be factored out thus:

```
normalise :: (Float, Float) -> (Float, Float)
normalise (x, y)
  = (x / m, y / m)
  where
    m = sqrt (x^2 + y^2)
```

`sqrt (x^2 + y^2)` will now be evaluated *once*

- They are also useful for naming the components of a tuple using *pattern matching*, e.g.

```
quotrem :: Int -> Int -> (Int, Int)
quotrem x y = (x `div` y, x `mod` y)

-- Converts distance in yards to a triple
-- (miles, furlongs, yards)
raceLength :: Int -> (Int, Int, Int)
raceLength y
  = (m, f, y'')
    where
      (m, y')  = quotrem y 1760
      (f, y'') = quotrem y' 220
```

E.g. `raceLength 2640 = (1, 4, 0)`, i.e. 1.5 miles exactly

- Note that you can define local *functions* too, e.g.

```
raceLength :: Int -> (Int, Int, Int)
raceLength y
  = (m, f, y'')
    where
      (m, y')  = quotrem y 1760
      (f, y'') = quotrem y' 220

      quotrem :: Int -> Int -> (Int, Int)
      quotrem x y = (x `div` y, x `mod` y)
```

- Here, `quotrem` cannot be used outside the definition of `raceLength`

- Remark: I'll often omit the type signature on local function definitions

- Remark: To aid readability we can name types using a *type synonym*, e.g. (assuming `g` is already defined),

```
type Mass     = Float
type Position = (Float, Float)
type Force    = (Float, Float)
type Object   = (Mass, Position)

force :: Object -> Object -> Force
force (m1, (x1, y1)) (m2, (x2, y2))
  = (f * dx / r, f * dy / r)
  where dx = abs (x1 - x2)
        dy = abs (y1 - y2)
        r  = sqrt (dx^2 + dy^2)
        f  = g * m1 * m2 / r^2
```

- Rule: Type synonyms must begin with a capital letter

Scope

- The *scope* of an identifier is that part of the program in which the identifier has a meaning
- All identifiers defined at the "top level" (i.e. non-local) are in scope over the entire program (they are *global*)
- Within each rule, each argument identifier *and* each local identifier is in scope everywhere throughout the rule
- Identifiers introduced in (nested) where clauses attached to local rules are in scope only in that local rule i.e. *not* in the outer rule as well

- Identifiers in `where` clauses supersede argument identifiers with the same name

- Similarly, identifiers in a nested `where` clause supersede those with the same name in an outer `where` clause, and so on, e.g.

```
f :: Int -> Int -> Int
f x y
  = x + y
  where
    y = x^2
      where
        x = 3
```

- Here, the function has the same meaning as `f x y = x + 9`; the `y` argument identifier is in scope nowhere

Evaluation

- Haskell evaluates an expression by reducing it to its simplest equivalent form (called its *normal form*) and printing the result
- Evaluation can be thought of as rewriting or *reduction* (meaning simplification); a reducible expression is called a *redex*
- Reduction works by repeatedly reducing redexes until no more redexes exist; the expression is then in normal form
- E.g. consider `double (3 + 4)`, where

```
double :: Int -> Int
double x
  = x + x
```

- One possible reduction sequence is *call-by-value*:

```
double (3 + 4)
-> double 7                 by built-in rules for +
-> 7 + 7                    by the rule for double
-> 14                       by built-in rules for +
```

- `14` cannot be further reduced (it is in normal form) and will be printed by the evaluator

- Another possible reduction sequence is *call-by-name*:

```
double (3 + 4)
-> (3 + 4) + (3 + 4)        by the rule for double
-> 7 + (3 + 4)              by built-in rules for +
-> 7 + 7                    by built-in rules for +
-> 14                       by built-in rules for +
```

- Thus evaluation is a simple process of *substitution and simplification*, using primitive rules for the built-in functions and additional function rules supplied by the programmer

- If an expression has a *well-defined value*, then the order in which the evaluation is carried out does not affect the result (the *Church-Rosser* property)

- But, the evaluator selects a redex (from the set of possible redexes) in a consistent way. This is called its evaluation/reduction *strategy*

- Haskell's reduction strategy is called *lazy evaluation*, or *call-by-need* $\equiv$ call-by-name without redundant repeated computation; it is equivalent to choosing the *leftmost-outermost* redex each time

- Lazy evaluation reduces a redex *only*

  if the value of the redex is required to produce the normal form, e.g.

```
f :: Float -> Float -> Float
f x y
  | x < 0     = 0
  | otherwise = y
```

- If `x` is negative, the second argument (`y`) is not required, hence

```
*Main> f 3 5
5.0
*Main> f 3 (6 / 0)
Infinity
*Main> f (-5) (6 / 0)
0.0
```

- More of this later...

Recursive Functions

- Let us consider functions for taking the second, third and fourth powers of a given `Float`:

```
square :: Float -> Float
square x = x * x

cube ::  Float -> Float
cube x = x * x * x

fourthpower ::  Float -> Float
fourthpower x = x * x * x * x
```

- What about computing x^n for an *arbitrary* value of $n \geq 0$?
- Problem: Written out explicitly, the number of terms in right-hand side expression would depend on the value of n

- The solution is to use a *recurrence relationship*—in this case one that defines x^n in terms of x^{n-1}:

$$\begin{aligned} x^n &= \overbrace{x \times x \times x \times \ldots \times x}^{n \text{ times}} \\ &= x \times x^{n-1} \end{aligned}$$

- This suggests the *recursive* Haskell function:

```
power :: Float -> Int -> Float
-- Pre: n >= 0
power x n
  = x * power x (n - 1)
```

- However, this is not quite right, e.g.

```
power 2 3
-> 2 * power 2 (3 - 1)  =  2 * power 2 2
-> 2 * 2 * power 2 1
-> 2 * 2 * 2 * power 2 0
-> 2 * 2 * 2 * 2 * power 2 -1
-> ...
```

- Oops! We want things to stop at `power 2 0`, since this should give 1
- The case `power 2 0` is called a *base case*
- Note: the function is not designed to work for `n`<0, although we can easily fix that

- Hence:

```
power :: Float -> Int -> Float
-- Pre: n >= 0
power x n
  | n == 0    = 1
  | otherwise = x * power x (n - 1)
```

- This function/definition is said to be *recursive*, since it calls itself
- Note that a measure of the *cost* of the function `power` is the number of multiplications required to compute `power n` for an arbitrary n
- Here the "cost" is n and we say that the function's *complexity* is "order n", written $O(n)$

$\boxed{?}$ How would you make `power` work for all integers `n`?

Recursive structure

- This is how we build *all* recursive functions

1. Define the base case(s)
2. Define the recursive case(s)
   (a) Split the problem into one or more subproblems
   (b) Solve the subproblems
   (c) Combine results from (b) to get the answer

- The subproblems are solved by a *recursive* call to the (same) function

- Important: the subproblems *must* be "smaller" than the original problem otherwise the recursion never stops, e.g.

```
loop :: Int -> Int
loop x
  | x == 0 = 0
  | x > 0  = 1 + loop x
```

- For example...

```
loop 4
-> 1 + loop 4
-> 1 + 1 + loop 4
-> ...
```

- This is called an *infinite loop* or a *black hole*; the program runs forever, or until it runs out of memory

A better version of `power`

- Idea: use the fact that x^n can be written $x^{n/2} \times x^{n/2} = (x^{n/2})^2$ if n is even and $x \times (x^{\lfloor n/2 \rfloor})^2$ if n is odd

- Graphically, for *even* n:

$$\begin{aligned} x^n &= \overbrace{x \times x \times \ldots \times x}^{n \text{ times}} \\ &= \underbrace{\overbrace{x \times \ldots \times x}^{n/2 \text{ times}}}_{x^{n/2}} \times \underbrace{\overbrace{x \times \ldots \times x}^{n/2 \text{ times}}}_{x^{n/2}} \end{aligned}$$

- Similarly for *odd* n

- The term $x^{\lfloor n/2 \rfloor}$ is referred to several times, so we'll define it using a `where`; also we'll arbitrarily use guards instead of conditionals:

```
power :: Float -> Int -> Float
-- Pre: n >= 0
power x n
  | n == 0 = 1
  | even n = k * k
  | odd n  = x * k * k     (or x * power x (n-1))
  where
    k = power x (n `div` 2)
```

$\boxed{?}$ What is the cost now, in terms of the number of multiplications, for a given n?

- Another example: Newton's method for finding the square roots of numbers. This repeatedly improves approximations to the answer until the required degree of accuracy is achieved.
- Given x, if a_n is the n^{th} approximation to $\sqrt{x}$ then

$$a_{n+1} = \frac{a_n + x/a_n}{2}$$

  gives the next approximation
- Let's define a function `squareRoot` which given a number x returns a "good" approximation to $\sqrt{x}$
- Here we will use $x/2$ as the first approximation of $\sqrt{x}$, i.e. $a_0 = x/2$

- We want to stop when the approximation is "close" to $\sqrt{x}$
- We thus check the *relative error* between x and a_n^2. If $|x - a_n^2|/x < \epsilon$ for some small value of ϵ (here 0.0001), we'll terminate the recursion:

```
squareRoot :: Float -> Float
-- Pre: x >= 0
squareRoot x
  = squareRoot' (x / 2)
  where
    squareRoot' :: Float -> Float
    squareRoot' a
      | abs (x - a * a) / x < 0.0001  =  a
      | otherwise = squareRoot' ((a + x/a) / 2)
```

- For example:

```
squareRoot 12
-> squareRoot' 6.0
-> squareRoot' 4.0
-> squareRoot' 3.5
-> squareRoot' 3.464286
-> squareRoot' 3.464102
-> 3.464102
```

since `|12-3.464102*3.464102|/12 < 0.0001`

3. List Processing

- The list square bracket notation (`[, , ,]`) is actually a shorthand
- At the simplest level lists are put together using two types of building block:
  - `[]` (pronounced "nil" or "empty-list") is used to build an empty list
  - `:` (pronounced "cons") is an infix operator which adds a new element to the front of a list
- These work like any other function, but are called *constructors* for reasons which will become apparent

- New lists can be built by repeated use of '`:`', starting with `[]`, e.g.

```
Prelude> []
[]
Prelude> True : []
[True]
Prelude> 1 : 2 : []
[1, 2]
```

- Thus, the expression `[x1, ..., xn]` is just a convenient shorthand for `x1 : ... : xn : []` (we can use either)

- Note also that `:` associates to the right so that `x : x' : xs` is interpreted as `x : (x' : xs)`

Polymorphism

- Importantly, the constructors `[]` and `:` can be used to build lists of *arbitrary* type. They are therefore said to be *polymorphic*
- To express this in type definitions, we use a *type variable*, which is an identifier beginning with a lower-case letter
- For example, the types of the two list constructors are:

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

- Here `a` is a type variable; the second line reads: "`(:)` is a function which takes an object of *any* type `a` and a list of objects of (the same) type `a` and delivers a list of objects of (the same) type `a`"

- A variable in a type (e.g. `a` above) stands for any type (*for all a*, or $\forall a$), but once determined each `a` in the type must be the same
- For example, `Int -> [Int] -> [Int]` is a valid *instance* of type `a -> [a] -> [a]`, but `Char -> [Int] -> [Int]` is not
- Many other Haskell prelude functions are polymorphic, e.g.

`id :: a -> a`	The identity function
`fst :: (a, b) -> a`	Pair index
`snd :: (a, b) -> b`	Pair index

- Note that the type of `fst` and `snd` involve two type variables since pair elements can be of any type

- `[]` and `:` can be used in function definitions to build lists
- As an exercise, let's write a recursive function that computes the sequence `[n, n-1 .. 1]` for a given `n`

```
ints :: Int -> [Int]
-- Pre: n >= 0
ints n
  | n == 0 = []
  | n > 0  = n : ints (n - 1)
```

- What about generating the list in increasing order, `[1 .. n]`?
  - We could use Haskell's "append" function (`++`) and replace the RHS `n : ints (n - 1)` with `ints (n - 1) ++ [n]`
  - We could use a helper function that counts *up* from 1 to `n`
  - We could use a helper function that carries an accumulating parameter...

- Here are the second and third versions; in the second, `ks` is the *accumulating parameter*, initially `[]`...

```
ints n
  = ints' 1
  where
    ints' k
      | k > n     = []
      | otherwise = k : ints' (k + 1)
ints n
  = ints' n []
  where
    ints' k ks
      | k == 0    = ks
      | otherwise = ints' (k - 1) (k : ks)
```

- What about functions which *consume* lists?

Method 1: `null`, `head` and `tail`

- Example: a variant of Haskell's built-in `sum` function: this version sums the elements of a list of `Int`s using the built-in functions:
  - `null :: [a] -> Bool` asks whether a list is empty;
  - `head :: [a] -> a` returns the head of a given list
  - `tail :: [a] -> [a]` returns the tail of a given list
- This works fine, but DO NOT DO IT THIS WAY!:

```
sum :: [Int] -> Int
sum xs
  = if null xs
    then 0
    else head xs + sum (tail xs)
```

- For example:

```
sum [10, 20, 30]
-> if null [10, 20, 30]
   then 0
   else head [10, 20, 30] + sum (tail [10, 20, 30])
-> head [10, 20, 30] + sum (tail [10, 20, 30])
-> 10 + sum [20, 30]
-> 10 + if null [20, 30] then ... else ...
-> 10 + head [20, 30] + sum (tail [20, 30])
-> 10 + 20 + sum [30]
-> 10 + 20 + if null [30] then ... else ...
-> 10 + 20 + head [30] + sum (tail [30])
-> 10 + 20 + 30 + if null [] then 0 else ...
-> 10 + 20 + 30 + 0
-> 60
```

Method 2 (much better): Pattern Matching

- Note that there are *exactly* two ways to build a list (`[]` and `:`) and hence *exactly* two ways to take them apart
- If we need to take a list apart then, when we look at the list, either:
    1. The list is empty, i.e. "of the form" `[]`
    2. The list is non-empty, i.e. "of the form" `(x : xs)` for some `x` and `xs`
- There are *no* other possibilities
- Here, "of the form" means "matches the pattern"
- Another way to define `sum` is by *pattern matching*...

- There are two possible patterns, so we have two rules:

```
sum :: [Int] -> Int
sum []       =  0
sum (x : xs) =  x + sum xs
```

- Think of the whole of each left-hand side as a *pattern*; patterns are tested from left to right and from top to bottom
- If the pattern matches the expression we are trying to evaluate, we return the result of evaluating the right-hand side
- Note: the pattern `(x : xs)` also serves to name the two 'things' attached to the first `:`, namely the head and tail of the given list
- You should use pattern matching in preference to the use of `null`, `head` and `tail`

- Pattern matching simplifies how we think about reduction (although it's actually implemented similarly to the previous version above), e.g.

```
sum [10, 20, 30]
-> 10 + sum [20, 30]
-> 10 + (20 + sum [30])
-> 10 + (20 + (30 + sum []))
-> 10 + (20 + (30 + 0))
-> 60
```

Aside: Case expressions

- It's pos-
sible to do pattern matching in expressions using `case` syntax, e.g.

```
sum :: [Int] -> Int
sum xs
  = case xs of
      []       -> 0
      (x : xs) -> x + sum xs
```

- Note that this has a single rule with a single right hand side (RHS), cf. two rules each with patterns on the left hand side (LHS); they both compile to the same code, however

- There is no "right" way, but pattern matching rules are arguably more idiomatic, so we'll (mostly) stick with them

More on the Haskell Prelude

- As an exercise, we'll build our own versions of some of Haskell's predefined functions (we've simplified the types in some cases):

```
null   :: [a] -> Bool
head   :: [a] -> a
tail   :: [a] -> [a]
length :: [a] -> Int
elem   :: Eq a => a -> [a] -> Bool
(!!)   :: [a] -> Int -> a
(++)   :: [a] -> [a] -> [a]
take   :: Int -> [a] -> [a]
drop   :: Int -> [a] -> [a]
zip    :: [a] -> [b] -> [(a, b)]
unzip  :: [(a, b)] -> ([a], [b])
```

Aside: back to type classes

- Recall the `Num` class:

```
Prelude> :type 3
3 :: Num t => t
```

- Another type class is `Eq` – the set of types for which there is a notion of (in)equality:

```
Prelude> :type (==)
(==) :: Eq a => a -> a -> Bool
Prelude> :type (/=)
(/=) :: Eq a => a -> a -> Bool
```

- E.g. our version of `elem` (list membership) will only work with types `a` for which `==` (and `/=`) are defined, hence: `elem :: Eq a => a -> [a] -> Bool` – we'll see why later

- Recall: The function `null` delivers `True` if a given list is empty (`[]`); `False` otherwise

- The function `null` is defined using pattern matching...

```
null :: [a] -> Bool
null []         = True
null (x : xs) = False
```

- Alternatively, as there is no need to name the head and tail,

```
null :: [a] -> Bool
null []   = True
null any = False
```

- In fact, we don't even need to name the list in the second rule; the special *wildcard* pattern `_` can be used to save inventing a name, viz. `null _ = False`

- Recall: The function `head` selects the first element of a list, and `tail` selects the remaining portion

- The prelude versions report an *error* if you try to take the head or tail of an empty list; let's do something similar...

```
head :: [a] -> a
head []       = error "Prelude.head: empty list"
head (x : xs) = x

tail :: [a] -> [a]
tail []       = error "Prelude.tail: empty list"
tail (x : xs) = xs
```

- Note that `error :: String -> a` – it can return an object of *arbitrary type*

? What is `tail [1]`

- The function `length`
  returns the length of a list (i.e. the number of elements it contains):

```
Prelude> length "brontosaurus"
12
Prelude> length [(True, True, False)]
1
Prelude> length []
0
```

- A recursive definition using pattern matching...

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

- The function `elem` determines whether a given element is a member of a given list:

```
Prelude> elem 'c' "hatchet"
True
Prelude> elem (1,2) [(3,4), (5,6)]
False
```

- One of many possible definitions using pattern matching...

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

- Note: the RHS of the second rule could have been written `if x == y then True else elem x ys`, but the use of `||` is much neater

- The `!!` operator (sometimes pronounced “at”) performs list *indexing* (the head is index 0):

```
Prelude> [11, 22, 33] !! 1
22
Prelude> "Tea" !! 0
'T'
Prelude> "Tea" !! 5
*** Exception: Prelude.(!!): index too large
Prelude>  "Error" !! (-1)
*** Exception: Prelude.(!!): negative index
```

- Here is a recursive definition using pattern matching

```
infixl 9 !!
(!!) :: [a] -> Int -> a
(x : xs) !! n
  | n == 0 = x
  | n > 0  = xs !! (n - 1)
  | n < 0  = error "Prelude.(!!): negative index"
[] !! n = error "Prelude.(!!): index too large"
```

- Note the syntax for introducing new left- (`infixl`) and right- (`infixr`) associative operators with a given precedence (here 9)

- Operator names must be unique and built from the symbols

  `! # $ % . + * @ | > < ~ - : ^ \ = / ? &`

- The binary operator `++` (pronounced "concatenate" or "append") joins two lists of the same type to form a new list e.g.

```
Prelude> [1, 2, 3] ++ [1, 5]
[1, 2, 3, 1, 5]
Prelude> "" ++ "Rest"
"Rest"
Prelude> [head [1, 2, 3]] ++ tail [2, 8]
[1, 8]
```

- The recursive definition...

```
infixr 5 ++
(++) :: [a] -> [a] -> [a]
[] ++ ys        = ys
(x : xs) ++ ys = x : (xs ++ ys)
```

- `take n xs` returns the first `n` elements of `xs`
- `drop n xs` returns the remainder of the list after the first `n` elements have been removed

```
Prelude> take 4 "granted"
"gran"
Prelude> drop 2 [True, False, True]
[True]
Prelude> take 1 "away" ++ drop 1 "away"
"away"
Prelude> drop 8 "letters"
""
```

? How would you define `take` and `drop`?

- `zip` takes two lists and forms a single list of pairs by combining the elements pairwise
- `unzip` does the opposite
- Note: for `zip` if one
  list is longer than the other then the surplus elements are discarded

```
Prelude> zip [5, 3] [4, 9, 8]
[(5,4),(3,9)]
Prelude> zip "it" "up"
[('i','u'),('t','p')]
Prelude> unzip [("your", "jacket")]
(["your"],["jacket"])
```

? How would you define `zip` and `unzip`?

A bigger example: insertion sort

- The following function takes an `Int` and an *ordered* list of `Int`s and returns a new ordered list with the `Int` in the right place

```
insert :: Int -> [Int] -> [Int]
-- Pre:  The given list is ordered
insert x []     = [x]
insert x (y : ys)
    | x < y     = x : (y : ys)
    | otherwise = y : (insert x ys)
```

- For example: `insert 3 [1, 4, 9]` proceeds as follows:

```
-> 1 : (insert 3 [4, 9])
-> 1 : (3 : [4, 9])
-> [1, 3, 4, 9]
```

- If we repeatedly insert (unsorted) items into a sorted list, the final list will also be sorted, hence:

```
iSort :: [Int] -> [Int]
iSort []       =  []
iSort (x : xs) =  insert x (iSort xs)
```

- This 'algorithm' is called *(linear) insertion sort*

- An example:

```
iSort [4, 9, 1]
-> insert 4 (iSort [9, 1])
-> insert 4 (insert 9 (iSort [1]))
-> insert 4 (insert 9 (insert 1 (iSort [])))
-> insert 4 (insert 9 (insert 1 []))
-> insert 4 (insert 9 [1])
-> insert 4 [1, 9]
-> [1, 4, 9]
```

$\boxed{?}$ How many calls to '`:`' are made on average to sort a list with n items?

Another sorting function - merge sort

- Haskell's `splitAt` function takes an `Int` index n and a list of type `[a]` and splits the list at position n

- How does Haskell implement it? A quick solution:

```
splitAt :: Int -> [a] -> ([a], [a])
-- Pre: n >= 0
splitAt n xs
  = (take n xs, drop n xs)
```

- However, this traverses the first `n` elements of `xs` *twice*

- We can avoid this but the resulting function is more complicated...

```haskell
splitAt :: Int -> [a] -> ([a], [a])
-- Pre: n >= 0
splitAt n []
  = ([], [])
splitAt n (x : xs)
  | n == 0    = ([], x : xs)
  | otherwise = (x : xs', xs'')
  where
    (xs', xs'') = splitAt (n - 1) xs
```

- We need one more ingredient: the function `merge` takes two ordered lists of `Int`s and merges them to produce a single ordered list

```
merge :: [Int] -> [Int] -> [Int]
-- Pre: both argument lists are ordered
merge [] []
  = []
merge [] (x : xs)
  = x : xs
merge (x : xs) []
  = x : xs
merge (x : xs) (y : ys)
  | x < y       = x : merge xs (y : ys)
  | otherwise = y : merge (x : xs) ys
```

- Now, a beautiful rendition of a classic sorting algorithm:

```
mergeSort :: [Int] -> [Int]
mergeSort []
  = []
mergeSort xs
  = merge (mergeSort xs') (mergeSort xs'')
  where
    (xs', xs'') = splitAt (length xs `div` 2) xs
```

? This definition isn't quite right. What's wrong and how would you fix it?

? Explicitly calculating the length each time is madness: add a helper function (`mergeSort'`?) which carries round the list to be sorted and its length.

? Which is the best sorting function: `iSort` or `mergeSort`? Why?

Aside: Enumerated Types

- Enumerated types are special forms of *user-defined data types*–see later

- They introduce a new type and an associated set of elements, called *constructors*, of that type, e.g.

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

- This says that `Day` is a new type and that objects of type `Day` may either be `Mon`, `Tue`, ..., `Sun`

- Constructor names must be unique within a program

- When Haskell spots a constructor it knows immediately its type, e.g. `Fri` is immediately recognisable as an object of type `Day`

- Type and constructor names must begin with a capital letter but are otherwise completely arbitrary

- Some more examples:

```
data Switch = Off | On

data Colour = Black | Blue    | Green  | Cyan
            | Red   | Magenta | Yellow | White

data Kerrrpowwww = Plink | Plank | Plonk
```

- It's up to you to choose type and constructor names that make sense to you

- Functions can be defined on objects of type `Day`, `Kerrrpowwww`, `Switch` etc. using pattern matching, e.g.

```
bothOn :: Switch -> Switch -> Bool
bothOn On On   = True
bothOn On Off  = False
bothOn Off On  = False
bothOn Off Off = False
```

$\boxed{?}$ Can we replace the final three rules by a single rule, e.g.: `bothOn s s' = False`?

- Note: Don't be tempted to use `==` instead of pattern matching, e.g. `bothOn s s' = s == On && s' == On`, or `if xs == [] then ...` etc. – these are HORRID!

- Secret: internally, constructors are encoded 0, 1, 2..., e.g. `Off` and `On` are encoded 0, and 1, but the type system ensures there is no confusion between 0 for `Off`, for example, and 0 for `Plink`
- (We can arrange for a function `fromEnum` to tell us the internal code of a constructor – if you want to play with this check out the `Enum` type class)
- Note that the type `Bool` is just a data type with two constructors:

```
data Bool = False | True
```

- Haskell's boolean functions can be straightforwardly defined using pattern matching, e.g.

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

infixr 3 &&
True && True = True
b && b'      = False

etc.
```

$\boxed{?}$ What if we wrote four equations: `True && True = True`, `True && False = False`, `False && True = False`, `False && False = False`. Is this the same as the above?

4. Higher-order functions

- *Higher-order* functions take other functions as arguments, e.g.

```
map               :: (a -> b) -> [a] -> [b]
filter            :: (a -> Bool) -> [a] -> [a]
zipWith           :: (a -> b -> c) -> [a] -> [b] -> [c]
foldr             :: (a -> b -> b) -> b -> [a] -> b
foldl             :: (a -> b -> a) -> a -> [b] -> a
foldr1, foldl1 :: (a -> a -> a) -> [a] -> a
scanr             :: (a -> b -> b) -> b -> [a] -> [b]
scanl             :: (a -> b -> a) -> a -> [b] -> [a]
takeWhile, dropWhile
                  :: (a -> Bool) -> [a] -> [a]
iterate           :: (a -> a) -> a -> [a]
```

- (Again, we've simplified some of the types)

- The function `map` applies a given function (passed as a parameter) to every element of a given list, e.g.

```
Prelude> map succ [1, 2, 3, 4]
[2,3,4,5]
Prelude> map head ["Random", "Access", "Memory"]
"RAM"
```

- Note that the expression `map f xs` is equivalent to the list comprehension `[f x | x <- xs]` so one way to define `map` would be

```
map :: (a -> b) -> [a] -> [b]
map f xs
  = [f x | x <- xs]
```

- Recall that juxtaposition (here of `f` and `x`) means function application in Haskell

- We can define `map` recursively using pattern matching as well...

```
map :: (a -> b) -> [a] -> [b]
map f []
  = []
map f (x : xs)
  = f x : map f xs
```

- Let's see how it works with an example:

```
map not [True, False, False]
-> not True : map not [False, False]
-> not True : not False : map not [False]
-> not True : not False : not False : map not []
-> not True : not False : not False : []
-> [False, True, True]
```

- The function `filter` filters out elements of a list using a given *predicate* (a function that returns a `Bool`)
- The predicate is applied to each element of a given list; if the result is `False` the element is excluded from the result, e.g.

```
Prelude> filter even [1 .. 10]
[2,4,6,8,10]
Prelude> let f x = x > 6 in filter f [4,7,6,9,1]
[7,9]
Prelude> let f x = x /= 's' in filter f "scares"
"care"
```

- The expression `filter f xs` is equivalent to the list comprehension `[x | x <- xs, f x]`

? How would you define `filter` recursively?

- The function `zipWith` applies a given function pairwise to the elements of two given lists, e.g.

```
Prelude> zipWith (+) [1, 2, 3] [9, 5, 8]
[10,7,11]
Prelude> zipWith elem "abp" ["dog", "cat", "pig"]
[False,False,True]
Prelude> zipWith max [(2,1),(4,9)] [(1,1),(8,5)]
[(2,1),(8,5)]
```

- Recall: `(op)` is the prefix version of `op`; also, `elem e xs` is `True` iff `e` is an element of list `xs`

- The expression `zipWith f xs ys` is equivalent to the list comprehension `[f x y | (x, y) <- zip xs ys]`

? How would you define `zipWith` recursively?

- The functions `foldr` and `foldl` *reduce* a list to a value by 'inserting' a given function between each element:

$$foldr\ f\ u\ []\longrightarrow u$$
$$foldr\ f\ u\ [x_1, x_2, ..., x_n]\longrightarrow f\ x_1\ (f\ x_2\ (...(f\ x_n\ u)...))$$
$$foldl\ f\ u\ []\longrightarrow u$$
$$foldl\ f\ u\ [x_1, x_2, ..., x_n]\longrightarrow f\ (...(f\ (f\ u\ x_1)\ x_2)...)\ x_n$$

- The additional parameter, u, is the *unit* of f – computationally, it defines what to do when the given list is empty
- The types of `foldr` and `foldl` are:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldl :: (b -> a -> b) -> b -> [a] -> b
```

- Some examples:

```
Prelude> foldr (+) 0 [3, 5, 7, -3, 9]
21
Prelude> foldl (+) 0 [3, 5, 7, -3, 9]
21
Prelude> foldl (*) 1 [2, 4, 6, -1]
-48
Prelude> foldr (++) [] ["a", "bb", "ccc"]
"abbccc"
Prelude> foldr (:) [] [3, 5, 7, -3, 9]
[3,5,7,-3,9]
```

$\boxed{?}$ How would you define `foldr`? What about `foldl`?

- The functions `foldr1` and `foldl1` are versions of `foldr` and `foldl` without the unit:

$$foldr1\ f\ [x_1, x_2, ..., x_n] \longrightarrow f\ x_1\ (f\ x_2\ (...(f\ x_{n-1}\ x_n)...))$$
$$foldl1\ f\ [x_1, x_2, ..., x_n] \longrightarrow f\ (...(f\ (f\ x_1\ x_2)\ x_3)...)\ x_n$$

- Crucially, `foldr1` and `foldl1` are *undefined* for empty lists

- The types are:

```
foldr1, foldl1 :: (a -> a -> a) -> [a] -> a
```

- Some examples:

```
Prelude> let f x y = y in foldr1 f [1, 2, 3, 4]
4
Prelude> foldr1 min [4,7,6,2,7,3]
2
Prelude> foldr1 min []
*** Exception: Prelude.foldr1: empty list
```

? Is there a sensible answer to `foldr1 min []`?

? Under what conditions is `foldr1 op xs = foldl1 op xs`?

? How would you define `foldr1` and `foldl1`?

- Sometimes it's useful to know each intermediate result computed during a fold, e.g. when summing the elements of the list `[3, 2, 9, 5]` we may want each of:

```
0
0 + 3
0 + 3 + 2
0 + 3 + 2 + 9
0 + 3 + 2 + 9 + 5
```

as *partial sums* or *prefix sums*

- The functions `scanr` and `scanl` (and their '`1` variants) do exactly this:

```
scanr :: (a -> b -> b) -> b -> [a] -> [b]
scanl :: (a -> b -> a) -> a -> [b] -> [a]
```

- Some examples:

```
Prelude> scanl (+) 0 [3,2,9,5]
[0,3,5,14,19]
Prelude> scanr (+) 0 [3,2,9,5]
[19,16,14,5,0]
Prelude> scanr (:) []  "gobi"
["gobi","obi","bi","i",""]
Prelude> scanl1 min [4, 7, 3, 2, 5, 1, 8, 7]
[4,4,3,2,2,1,1,1]
```

? How would you define `scanr` and `scanl`?

- Three more built-in higher-order functions...

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

dropWhile :: (a -> Bool) -> [a] -> [a]
-- Exercise: write it

iterate :: (a -> a) -> a -> [a]
iterate f x
  = x : iterate f (f x)
```

- For example,

```
Prelude> take 10 (iterate succ 0)
[0,1,2,3,4,5,6,7,8,9]
Prelude> take 6 (iterate tail "suffix")
["suffix","uffix","ffix","fix","ix","x"]
Prelude> takeWhile even [2, 4, 7, 6]
[2,4]
Prelude> dropWhile isSpace "     Begin"
"Begin"
```

- Note that lazy evaluation is essential for evaluating expressions involving `iterate`

Binary Operator Extensions

- Some binary operators have corresponding generalisations over lists, for example:

Operator	Generalisation
`+`	`sum :: Num a => [a] -> a`
`*`	`product :: Num a => [a] -> a`
`&&`	`and :: [Bool] -> Bool`
`||`	`or :: [Bool] -> Bool`
`++`	`concat :: [[a]] -> [a]`
`max`	`maximum :: Ord a => [a] -> a`
`min`	`minimum :: Ord a => [a] -> a`

- Note: see later for a proper explanation of the types

- Examples...

```
Prelude> sum [1..6]
21
Prelude> product [2, 4, 1, 6]
48
Prelude> and [True, False, True]
False
Prelude> or [x < 3 | x <- [5, 4, 8, 1, 9]]
True
Prelude> concat ["Three ", "small ", "lists"]
"Three small lists"
Prelude> maximum [1, 4, 3, 1, 9]
9
```

- Note: these operators can be defined recursively using pattern matching, e.g.

```
product :: [Int] -> Int
product []       = 1
product (x : xs) = x * product xs

and :: [Bool] -> Bool
and []       = True
and (b : bs) = b && and bs
```

- However, in each case all we're doing is 'inserting' a binary operator in between each list element and using an appropriate unit element

- But this is just what `fold` does

- So, alternatively:

```
product xs = foldr (*) 1 xs
and xs     = foldr (&&) True xs
```

- Subtle point: It is easy to see that for all lists (of numbers) `xs`, `foldr (*) 1 xs` $\equiv$ `foldl (*) 1 xs`, so which is best?
  - `foldl (*) 1 xs` is *tail recursive*, which means that it's more efficient to implement
  - However, because `&&` works "left to right" it's better to use `foldr` for `and`

$\boxed{?}$ If we write `foldl (&&) True [False, b2, ..., bn]` are `b2`, ..., `bn` evaluated?

- By picking the most efficient implementation in each case we have:

```
sum xs     = foldl (+) 0 xs
product xs = foldl (*) 1 xs
and xs     = foldr (&&) True xs
or xs      = foldr (||) False xs
concat xs  = foldr (++) [] xs
maximum xs = foldl1 max xs
minimum xs = foldl1 min xs
```

- Note: `maximum []` and `minimum []` are not defined - hence the use of `foldl1`.

$\boxed{?}$ Could we use `foldr1` instead?

Currying and Partial Application

- Functions can also return other functions as their result (another type of higher-order function)
- Q: How can we evaluate something and end up with a new function? A: Partial application...
- Consider the function

```
plus :: Int -> Int -> Int
plus x y
  = x + y
```

- Why do we write `Int -> Int -> Int`?
- The answer is that `plus` actually introduces *two* single-argument functions...

1. `plus` is really a single-argument function whose type signature is `Int -> (Int -> Int)`

2. If `a :: Int` then `plus a` is a *function* of type `Int -> Int`

- So, `plus 4` is a perfectly meaningful expression—it is the function *which adds 4 to things*

- This suggests we can map partial applications over lists; let's try:

```
Prelude> map (plus 4) [1, 3, 8]
[5, 7, 12]
Prelude> map (elem 'e') ["No", "No again", "Yes"]
[False,False,True]
```

- An application which only partially completes the arguments of a function `f` is called a *partial application* of `f`

- The idea of treating all multi-argument functions "one argument at a time" is called *currying* after the mathematician **HASKELL B. Curry**!!

- Partial applications of operators are called *sections* and Haskell has some special notation to help. For example:

`(1/)`	is the 'reciprocal'	function
`(/2)`	is the 'halving'	function
`(^3)`	is the 'cubing'	function
`(+1)`	is the 'successor'	function
`(!!0)`	is the 'head'	function
`(==0)`	is the 'is-zero'	function

```
Prelude> map (==0) [4, 0, 8, 0]
[False,True,False,True]
Prelude> map (^2) [1..4]
[1,4,9,16]
Prelude> map (!!2) ["one", "two", "three"]
"eor"
Prelude> takeWhile (<20) (iterate (+3) 1)
[1,4,7,10,13,16,19]
Prelude> filter (/=0) (map (`mod` 2) [1..10])
[1,1,1,1,1]
```

- So functions really are 'first-class' citizens in Haskell. Indeed, even function composition can be expressed as a higher-order function:

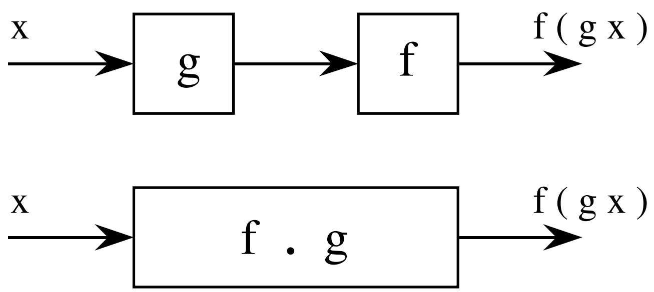
```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = h where h x = f (g x)
```

- Note: there are several other ways we can write this, e.g.

```
(f . g) x = f (g x)

f . g = \x -> f (g x)
```

Diagrammatically:



- Example: here are two equivalent definitions of functions `notNull` and `allZero`

```
notNull :: [a] -> Bool
notNull xs = not (null xs)

-- Or...
notNull xs = (not . null) xs

allZero :: [Int] -> Bool
allZero xs = and (map (==0) xs)

-- Or...
allZero xs = (and . map (==0)) xs
```

Extensionality

- A useful rule for simplifying some definitions is the *extensionality* rule from mathematics:

$$\text{if } \forall x,\ f\ x = g\ x \text{ then } f = g$$

- This means, for example, that in our `notNull` and `allZero` functions we can instead *cancel* the `xs` and write

```
notNull = not . null

allZero = and . map (==0)
```

- Similarly for our earlier examples, e.g.

```
sum      = foldl (+) 0
product  = foldl (*) 1
and      = foldr (&&) True
or       = foldr (||) False
concat   = foldr (++) []
maximum  = foldl1 max
minimum  = foldl1 min
```

- PING!

- These exploit the fact that function application associates to the left, i.e. $f\ x\ y\ z \equiv (\ (\ f\ x\ )\ y\ )\ z$

`squareRoot` as a function "pipeline"

- Here is an alternative definition of `squareRoot`:

```
squareRoot :: Float -> Float
-- Pre: x >= 0
squareRoot x
  = (head . dropWhile isBad . iterate next . (/2)) x
  where
    next a  = (a + x / a) / 2
    isBad a = abs (x - a^2) / x > 0.0001
```

- The `x` is passed from one function to the next, from right to left, cf. a pipeline

? Can you cancel the `x` in the LHS and RHS of `squareRoot`?

6. User-defined data types

- We have already seen *enumerated* types, for example:

```
data Bool = False | True


data Color = Black | Blue    | Green  | Cyan
           | Red   | Magenta | Yellow | White
```

- In general, constructors can also take arguments and both they (and the associated type) can be polymorphic

- These data types are sometimes called *user-defined types* or *algebraic data types*, or quite often just “data types”

Example...

- Haskell's built-in `Maybe` type:

```
data Maybe a = Nothing | Just a
```

- This is very handy for computations that might fail in some recoverable way (`Nothing` $\equiv$ "fail", `Just x` $\equiv$ "success with answer `x`")
  - `Nothing` is a constructor of type `Maybe a` (i.e. `Nothing :: Maybe a`)
  - `Just` is a constructor of type `a -> Maybe a` (i.e. `Just :: a -> Maybe a`)
- Constructors are thus functions with no rules
- They are defined implicitly when they appear in a `data` definition

- Haskell's `lookup` function returns a `Maybe` – it's a *really* useful function...

```
lookup key []
  = Nothing
lookup key ((k, v) : table)
  | key == k  = Just v
  | otherwise = lookup key table
```

  (We'll work out its exact type later.)

- For example:

```
Prelude> lookup 'x' [('a', 9), ('c', 2)]
Nothing
Prelude> lookup 4 [(9, "nine"), (4, "four"), (1, "one")]
Just "four"
```

- Uses of `lookup` are ubiquitous – we might use them like this:

```
f x table
  = ... f' (lookup x table) ...
  where
    f' Nothing
      = b
    f' (Just res)
      = g res
```

- Notice how we have used the constructors on the *left* of the `=` to form patterns, c.f. enumerated types

- However, we can make this type of code *much* shorter by using the functions in `Data.Maybe`...

```
import Data.Maybe
...
```

- Two commonly-used functions in `Data.Maybe`:

```
maybe :: b -> (a -> b) -> Maybe a -> b
maybe x f Nothing
  = x
maybe x f (Just y)
  = f y

fromJust :: Maybe a -> a
fromJust Nothing
  = error ...
fromJust (Just x)
  = x
```

- For example, `f` above can be written:

```
import Data.Maybe

f x table
  = ... maybe b g (lookup x table) ...
```

- If we happen
  to know that there will always be a binding for `x` in the `table` then:

```
f x table
  = ... g (fromJust (lookup x table)) ...
```

- For a bit of homework,
  check out the type `Either` which implements a *union* of two types:

```
data Either a b = Left a | Right b
```

  Note that it is parameterised by *two* type variables, as you'd
  expect

Recursive data types

- User-defined types can also be recursive, e.g. a “hand-made” list type:

```
data List a = Nil | Cons a (List a)
```

- The type `List a` is isomorphic to Haskell’s list type `[a]`

- Indeed, Haskell’s prelude essentially has this:

```
data [a] = [] | a : [a]
```

although this is *not* legal syntax

- If we stick with our own definition of lists (`List a`) we'll need to use `Nil` and `Cons` instead of `[]` and '`:`' e.g.

```
Nil
Cons 6 Nil
Cons "this" (Cons "that" Nil)
```

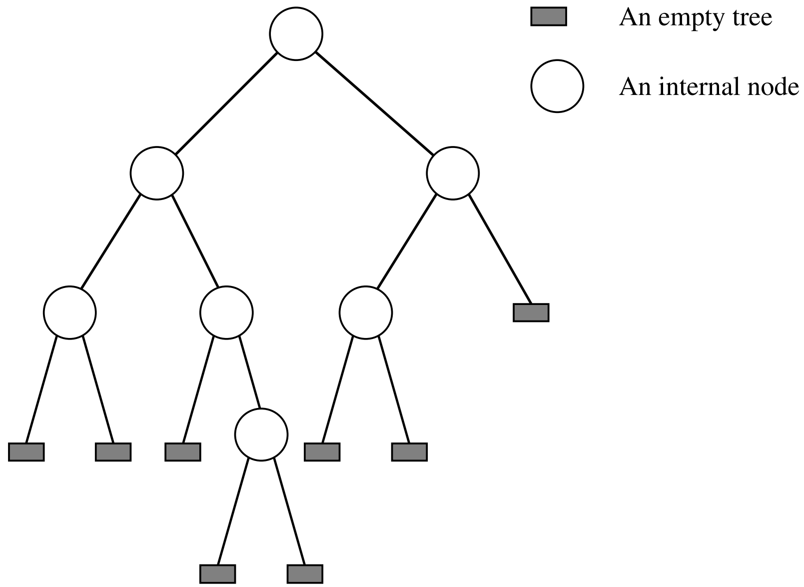
generate objects of type `List Int` and `List String` respectively

- We can also pattern match on terms involving `Nil` and `Cons` in the obvious way, e.g.

```
length :: List a -> Int
length Nil
  = 0
length (Cons x xs)
  = 1 + length xs
```

Trees

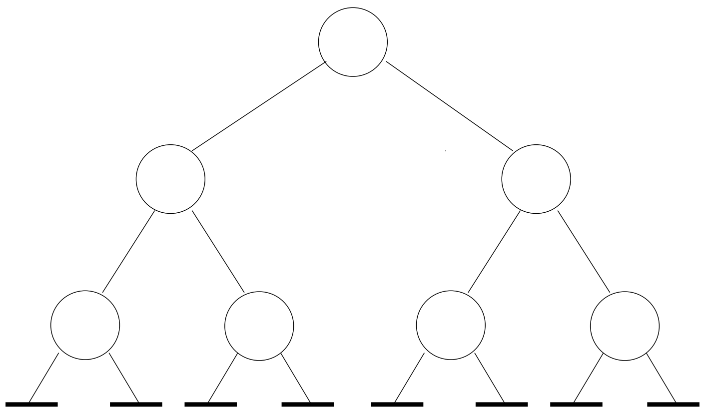
- Trees are powerful generalisations of lists and have a two-dimensional branching structure
- An example: a *binary* tree where each node has two subtrees



- Usually (but not always) elements are stored at the nodes, as assumed here

Why trees are important

- If a tree is *perfectly balanced*, i.e. every node has identical-sized subtrees then a tree with n items has $log_2(n+1)$ depth, e.g.



Balanced tree, n = 7, depth = 3

- It is thus possible to implement common operations on trees in *logarithmic* time, cf. linear time, as for a list
- Even if a tree is unbalanced it is often more efficient than using a list (it depends...)

- We can describe the structure of trees using a data type
- E.g. for tress like the above we'll call the constructor for an empty tree `Empty` and that for an internal node `Node`
- We'll allow any type of object to sit in the nodes, so we'll make the trees polymorphic:

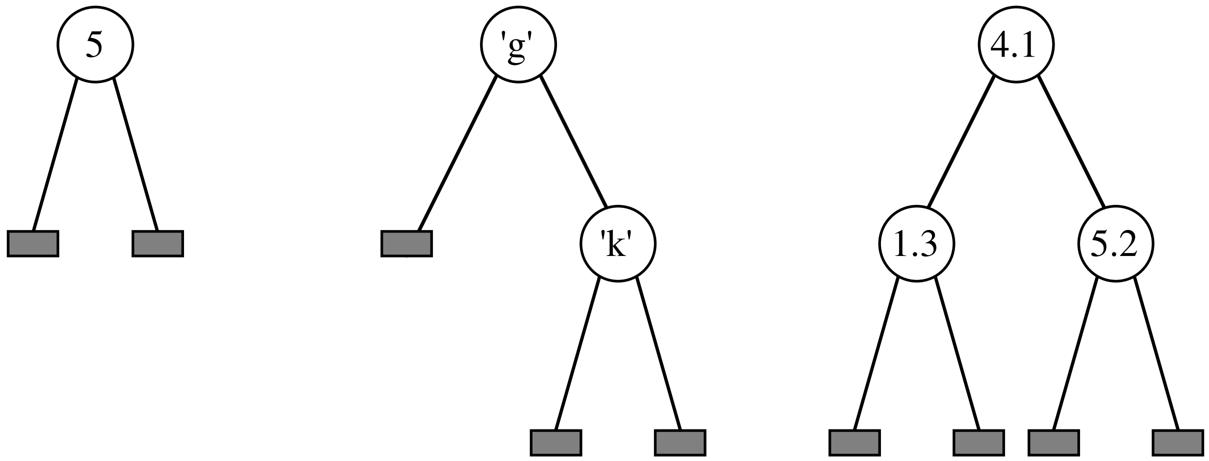
```
data Tree a = Empty | Node (Tree a) a (Tree a)
```

- Note we could rearrange the arguments of `Node`, e.g.

```
data Tree a = Empty | Node a (Tree a) (Tree a)
```

- It doesn't matter so long as we are consistent; we'll use the former

- Some example trees



(a) (b) (c)

- (a) is a `Tree Int`, (b) is a `Tree Char` and (c) is a `Tree Float` element is smaller than every right subtree element

- We can write Haskell expressions that represent these, e.g. (b) corresponds to `Node Empty 'g' (Node Empty 'k' Empty)` ?
What about (a) and (c)?

- As with lists we can write functions on `Trees` using pattern matching
- There are two types of tree, hence two types of pattern to consider
- Example: `size` for computing the number of nodes in a tree

```
size :: Tree a -> Int
size Empty
  = 0
size (Node l x r)
  = 1 + size l + size r
```

- Compare this with `length` for lists; here `size` has *two* “sub-trees” to explore beneath each `Node` hence *two* recursive calls to `size`

- Another example: `flatten` which will reduce a `Tree a` to a list of type `[a]` by performing an *in-order* traversal of the tree
- In-order traversal visits the nodes from left to right

```
flatten :: Tree a -> [a]
flatten Empty
  = []
flatten (Node t1 x t2)
  = flatten t1  ++  (x : flatten t2)
```

- Note that the flattened version of `t1` is the leftmost argument of `++` and therefore will appear *first* in the resulting list; hence the left-to-right order

$\boxed{?}$ How many calls to '`:`' are required to flatten a prefectly balanced tree containing $n = 2^k - 1$ elements? How would you redefine `flatten` so that exactly *one* '`:`' is required for each element?

- Example: a function to insert a number into an ordered tree
- An ordered tree satisfies the property that for every node:
  - Every element of the left subtree is smaller than (or equal to) the element at the node
  - The node element is smaller than every element in the right subtree

```
insert :: Int -> Tree Int -> Tree Int
-- Pre: The tree is ordered
insert n Empty
  = Node Empty n Empty
insert n (Node t1 x t2)
  | n <= x     = Node (insert n t1) x t2
  | otherwise = Node t1 x (insert n t2)
```

? Note that `insert` as defined is *not* polymorphic. Why?

- Example: a function to construct an ordered tree from an unordered list of numbers:

```
build :: [Int] -> Tree Int
build
  = foldr insert Empty
```

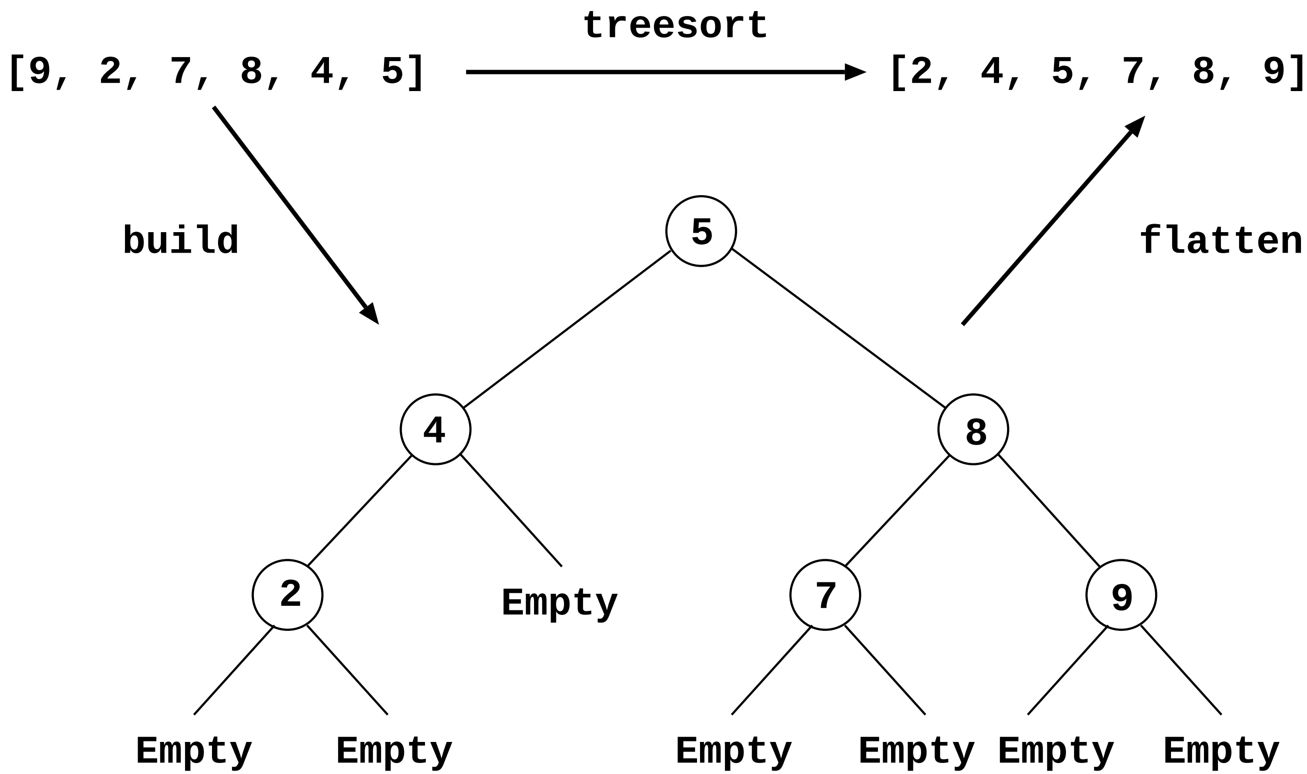
- Hence, yet another program for sorting a list of numbers:

```
treeSort :: [Int] -> [Int]
treeSort
  = flatten . build
```

$\boxed{?}$ On average, how many nodes have to be visited for each insertion?

$\boxed{?}$ If you use the optimised version of `flatten` (see above) how many constructor calls (list and tree constructors) are required on average to sort a list of n elements?

- The composition of `build` and `flatten` can be seen clearly with a pretty picture:



6. Type classes

- In contrast to polymorphic functions such as `length`, some functions, e.g. `==`, are *overloaded*:
  - they can be used at more than one type
  - their definitions are different at different types
- The collection of types over which a function is defined is called a *class*
- The set of types over which `==` is defined is called the *equality class*, `Eq`
- We say that `==` is a *member function* of `Eq`
- Note that `/=` is also a member of `Eq`
- The Haskell equality class is defined by:

```
class Eq t where
  (==), (/=) :: t -> t -> Bool
  x /= y = not (x == y)
  x == y = not (x /= y)
```

- Note the *default* definition of `==` and `/=`; as we add new types to the equality class, `/=` will de defined automatically (in terms of `==`)

- The member *types* of a class (the types that `t` can be above) are called *instances* of that class; for `Eq` the instances include `Int, Float, Bool, Char`

- It is this which enables use to write things like `True == True || 'a' == 'b' && 13 /= 7`

Extending a Class

- Recall from earlier:

```
data Switch = Off | On
```

- Suppose we wish to check whether two `Switch` values are equal. We could define a new function, e.g.

```
eqSwitch :: Switch -> Switch -> Bool
eqSwitch On On    = True
eqSwitch Off Off  = True
eqSwitch s1 s2    = False
```

- This is fine, but it would be much more convenient if we could use `==` instead, as in `Off == On`, for example

- Problem: `Switch` is *not* a member type of `Eq`, but we can add it in one of two ways:

1 Explicitly by adding a definition of '`==`' on values of type `Switch`:

```
instance Eq Switch where
  On  == On  = True
  Off == Off = True
  s1  == s2  = False
```

(Note that `/=` is defined in terms of `==` by default in the class definition but we could *override* it here if we wanted)

2 Implicitly using the keyword `deriving` in the `data` definition:

```
data Switch = On | Off
            deriving (Eq)
```

- The use of `deriving` saves us a lot of work—the system builds the definition of `==` over `Switch` values automatically
- To appreciate the benefits, try writing `==` for type `Day`, defined earlier

Puzzle: Given the `Eq` class definition:

```
class Eq t where
  (==), (/=) :: t -> t -> Bool
  x /= y =  not (x == y)
  x == y =  not (x /= y)
```

and the following data type and instance declaration:

```
data D = C Int

instance Eq D
```

(i.e. `D` uses `Eq`'s default definitions for `==` and `/=`), what happens if we try to compute `C 1 == C 2`?

Contexts

- Some function types need to be *restricted* to reflect the operations that they perform on their arguments
- Here is a valid definition

```
equals :: Int -> Int -> Bool
equals x y
  = x == y
```

- However, if we try to make this polymorphic, as in

```
equals :: t -> t -> Bool
equals x y
  = x == y
```

  we get an error
- A type variable `t` in a type means (literally) “for all `t`”, but `equals` will only work if values of type `t` are *comparable*

- To make the type of `equals` as *general* as possible, we need to give `t` a *context* thus

```
equals :: Eq t => t -> t -> Bool
equals x y = x == y
```

- `Eq t => ...` now means "any `t` that is a member of `Eq`" rather than "for *all* `t`"

- Example: Haskell provides a built-in function `elem` for testing membership of a list, e.g.

```
*Main> elem 1 [2, 4, 9]
False
*Main> elem 'a' "Harry"
True
*Main> elem True []
False
```

- So, what is the type of `elem`?
- The basic type structure is clearly of the form
  `a -> [a] -> Bool`
- However, the list elements (the things of type `a`) *must* be comparable, i.e. `a` must be an instance of `Eq`
- In the Haskell standard prelude, we find:

```
elem :: Eq a => a -> [a] -> Bool
```

- Q: What is the most general type of Haskell's `lookup` function?

```
lookup key []
  = Nothing
lookup key ((k, v) : table)
  | key == k  = Just v
  | otherwise = lookup key table
```

Derived Classes

- Some classes may restrict their instance types to belong to certain other classes
- The simplest example is another built-in class called `Ord` representing the *ordered* types
- For a type to be a member of `Ord` it must also be a member of the *superclass* `Eq`
- Given the data type:

```
data Ordering = LT | EQ | GT
```

`Ord` can be defined using a context thus (see over):

```haskell
class (Eq a) => Ord a where
  compare                  :: a -> a -> Ordering
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max, min                 :: a -> a -> a
  compare x y
    | x == y     =  EQ
    | x <= y     =  LT
    | otherwise  =  GT
  x <= y = compare x y /= GT
  x <  y = compare x y == LT
  x >= y = compare x y /= LT
  x >  y = compare x y == GT
```

- We say that `Ord` *inherits* the operations of `Eq`

- Note that we can only compute `x <= y` if we can also compute `x == y`
- The basic types `Int, Float, Bool, Char` are all instances of `Ord`
- This enables us to write, e.g. `4 <= 9`, `'d' > 't'`, `max True False`
- If necessary, we can add new types
  to `Ord` in the same way that we added new types to `Eq`, for example

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

$\boxed{?}$ Because `Ord` inherits from `Eq` do we always have to derive both `Eq` and `Ord`?

- The automatically-generated definitions of `<, <= , >, ...` assume the constructors to be ordered as they are written
- Thus

```
More> Tue <  Mon
False
More> Thu >= Mon
True
More> Sun <= Sun
True
More> Fri == Sat
False
```

- Recursive data types can also be added to classes `Eq` and `Ord`, e.g. for our `List` type above:

```
data List a = Nil | Cons a (List a)
            deriving (Eq, Ord)
```

- We can compare two `List a` values *provided* that `a` is also an instance of `Ord`, e.g.

```
More> Cons 8 Nil > Cons 7 Nil
True
More> Cons Mon Nil >= Nil
True
More> Cons False (Cons True Nil) < Nil
False
```

- Note, however, that `Cons Off Nil > Nil` is an error because the type `Switch` is not an instance of `Ord`

- Another important class is called `Show` which has a member function `show :: a -> String` for converting an object of instance type `a` into a string

- Haskell uses this to print the value of an expression at the terminal, e.g.

```
Prelude> 74.6
74.6
Prelude> show 74.6
"74.6"
Prelude> show "74.6"
"\"74.6\""
```

- If a type `t` is not a member of `Show` then we can't display objects of type `t`
- For example,
  at the moment there is no instance of `Show` for the `Day` data type so:

```
*Main> Mon


ERROR: Cannot find "show" function for:
*** expression : Mon
*** of type    : Day
```

- We can fix it by deriving `Show` in the definition of `Day`...

- This will define `Show`'s member function `show` automatically; the constructor name will then be used as its representation:

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

- Whereupon:

```
*Main> Mon
Mon
*Main> (Mon, Fri)
(Mon,Fri)
```

- Alternatively, we might want to display values of type `Day` differently:

```
instance Show Day where
  show Mon = "Monday"
  show Tue = "Tuesday"
  show Wed = "Wednesday"
  show Thu = "Thursday"
  show Fri = "Friday"
  show Sat = "Saturday"
  show Sun = "Sunday"
```

- For example,

```
*Main> (Mon, Fri)
(Monday,Friday)
```

Multiple Constraints

- Contexts can include an arbitrary number of constraints, for example

```
showSmaller x y = if x < y then show x else show y
```

- Both `x` and `y` must be comparable by `<` and valid arguments to `show`, i.e. instances of *both* `Ord` and `Show`

```
Prelude> :t showSmaller
showSmaller :: (Ord a, Show a) => a -> a -> String
```

- Multiple constraints can occur in instance declarations
- For example, the pair type `(t, u)` is already defined to be an instance of `Eq`
- For two pairs to be comparable using `==` their components must also be comparable
- Hence, Haskell implements something like this:

```
instance (Eq t, Eq u) => Eq (t, u) where
  (a, b) == (c, d)  =  if a == c
                       then b == d
                       else False
```

- Finally... suppose we have a data type and evaluator for expressions:

```
data Exp = Const Int | Add Exp Exp | Sub Exp Exp |
                Mul Exp Exp
              deriving (Show, Eq)

eval :: Exp -> Int
eval (Const x)
  = x
eval (Add e e')
  = eval e + eval e'
eval (Sub e e')
  = eval e - eval e'
eval (Mul e e')
  = eval e * eval e'
```

- For example, `3 * 6 + 7` would be represented by:

```
Add (Mul (Const 3) (Const 6)) (Const 7)
```

- We can evaluate expressions like this using `eval`, e.g.:

```
*Main> eval (Add (Mul (Const 3) (Const 6)) (Const 7))
25
```

- This is all a bit of a mouthful. It would be much better if we could just type `3 * 6 + 7` and have it interpreted as `Add (Mul (Const 3) (Const 6)) (Const 7)` automatically

- We can do this by making `Exp` an instance of the `Num` class and defining `+`, `*` etc. accordingly

- This essentially defines a new "language" for expressions – we are *embedding* one language within another

- Here are the member functions of `Num`...

```
class Num a where
  (+) :: a -> a -> a
  (*) :: a -> a -> a
  (-) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
```

  Predefined member types: `Integer`, `Int`, `Float`, `Double`

- Note that we must define at least one of `-` and `negate` as they are mutually defined by default

- To make the instance, note that `e + e'` needs to be interpreted as `Add e e'`, i.e. `(+) = Add`; similarly the other functions:

```
instance Num Exp where
  (+) = Add
  (-) = Sub
  (*) = Mul
  abs x = error "abs not implemented"
  signum x = error "signum not implemented"
  fromInteger = Const . fromInteger
```

- Note that for `fromInteger` a definition like `fromInteger n = Const n`, for example, would attempt to apply `Const` to an `Integer` rather than an `Int`

- Amazingly, we can now use the type system to build the *representation* of an expression, rather than evaluate it...

- For example:

```
*Main> 3*6+7
25
*Main> 3*6+7 :: Exp
Add (Mul (Const 3) (Const 6)) (Const 7)
*Main> sum [8*9, 4]
76
*Main> sum [8*9, 4] :: Exp
Add (Add (Const 0) (Mul (Const 8) (Const 9))) (Const 4)
*Main> eval (sum [8*9, 4])
76
```

- Note that integer constants on the command line are automatically translated to calls to `fromInteger`
- Note also that the *default* instance for the `Num` class is `Integer`, e.g. `7::Exp` essentially overrides the default `7::Integer`

7. I/O and monads

Q: How can we implement I/O in a pure functional language, which has no side effects?

- To understand why Haskell's I/O is the way it is, it's useful to try to construct the problem 'bottom up'
- Doing I/O necessarily changes the state of the 'world', e.g. by reading characters from a keyboard or writing data to a file
- To do this purely functionally means that we somehow have to carry round the state of the world
- To understand the issues, we'll begin by thinking of the world state as being something *explicit*, i.e. a data structure, that is passed to and from functions explicitly

- Let's focus on input from and output to a user terminal – when we get this right it's easy to see how it can be generalised
- Assume that we magically know all the user's keyboard input in advance (a `String`) and keep track of the state of the screen output (another `String`), e.g.

```
type World = (String, String)

-- Some magical initial world...
w0 :: World
w0 = ("abcdefgh", "")
```

- In practice, the `World` will be much more complicated and will need to encode the current status of all I/O devices maintained by the operating system

- Using our very simple ‘terminal’ model of I/O, we can now write a function to read a character from the keyboard:

```
getChar :: World -> (Char, World)
getChar (c : cs, out)
  = (c, (cs, out))
```

- What about printing a character to the screen? We’ll see in a moment that it’s convenient to think of this returning a void result, `()`, in addition to the modified world:

```
putChar :: Char -> World -> ((), World)
putChar c (cs, out)
  = ((), (cs, out ++ [c]))
```

- For example, assuming our initial world, `w0`

```
*Main> getChar w0       -- Read 'a' from the keyboard
('a',("bcdefgh",""))
*Main> putChar 'X' w0   -- Write 'X' to the screen
((), ("abcdefgh","X"))
```

- We see the effect of the I/O in the form of a pair, but this is only a model: in practice, the I/O functions will invoke the operating system to *side effect* physical devices, e.g. the keyboard reader and screen in this case

- Notice that `getChar` and `putChar` have similar type signatures, so we can tidy things up by using a type synonym:

```
type IO a = World -> (a, World)
```

- Whereupon...

```
getChar :: IO Char
getChar (c : cs, out)
  = (c, (cs, out))

putChar :: Char -> IO ()
putChar c (cs, out)
  = ((), (cs, out ++ [c]))
```

- Note, for example, that in the case of `putChar`, `Char -> World -> ((), World)` is the same as `Char -> (World -> ((), World))`, which is the same as `Char -> IO ()`

- Q: How can we perform two or more I/O actions in sequence?
- A: Take the (possibly modified) world generated by the first and then execute the second with respect to it
- However, all we have at the moment are just I/O actions (`IO a`, for some `a`) so the only way we can do this is to provide some additional 'sequencing' functions to help:
- We'll "invent" two useful generic functions – actually, we'll make them infix operators:

  `>>` Perform two independent actions, one after the other

  `>>=` Perform an action that produces some result, `r` say, and then pass `r` to a second action
- Here are their definitions...

```haskell
infixl 1 >>, >>=
(>>)  :: IO a -> IO b -> IO b
(a1 >> a2) w
  = let (r1, w1) = a1 w
        (r2, w2) = a2 w1
    in (r2, w2)
```

Alternatively (and we could, of course, use `where`):

```haskell
(a1 >> a2) w
  = let (_, w1) = a1 w
    in a2 w1
```

```haskell
(>>=) :: IO a -> (a -> IO b) -> IO b
(a1 >>= a2) w
  = let (r1, w1) = a1 w
        (r2, w2) = a2 r1 w1
    in (r2, w2)
```

Alternatively:

```haskell
(a1 >>= a2) w
  = let (r1, w1) = a1 w
    in a2 r1 w1
```

- For example:

```
printHI :: IO ()
printHI
  = putChar 'H' >> putChar 'I'
```

- Short aside: In order to run a program with type `IO`, we have to give it the hidden (initial) world, `w0` say
- In practice, this will be done by the run-time system at the topmost level, but we can mock this up:

```
runIO :: IO a -> (a, World)
runIO p
  = p w0
```

- So, `runIO printHI` returns `((),("abcdefgh","HI"))`, which, using our simple I/O model, corresponds to printing `"HI"` on the screen

- Another example – read a character and then print it (`c` is the character read by `getChar`):

```
readAndPrint :: IO ()
readAndPrint
  = getChar >>= \c -> putChar c
```

Alternatively:

```
readAndPrint
  = getChar >>= putChar
```

- E.g., `runIO readAndPrint` returns `((),("bcdefgh","a"))`

- So far so good, but we have a problem: if we expose the representation of the world then we can sneakily read input as a side effect of pattern matching, e.g. for our simple world model:

```
peek :: World -> Char
peek (c : cs, out)
  = c
```

- Much more serious, any function that does I/O can 'corrupt' the world:

```
hack :: IO ()
hack (in, out)
  = (in, "Nasty message: I hate you!")
```

- The solution is essentially to *hide* the representation of the world from the user and just publish the types of the I/O primitives as an "interface" to the hidden world:

```
getChar :: IO Char
putChar :: IO ()
```

- Important: *there is only one world*
  - Each interface function only ever interacts with the single, hidden world
  - Users cannot see the world and so cannot duplicate it
  - There is no 'duplicate world' function in the interface

- In Haskell, `IO` is a built-in type and `getChar` and `putChar` are defined in a low-level I/O module (`System.IO`); interaction with the "real" world is via low-level I/O operations
- The functions `>>` and `>>=` are member functions of a class called `Monad` describing "sequencable" types:

```
class Applicative m => Monad (m :: * -> *) where
  (>>=)  :: m a -> (a -> m b) -> m b
  (>>)   :: m a -> m b -> m b
  return :: a -> m a
  fail   :: String -> m a
```

- The '`m`' here represents a *type constructors* (cf. a type), e.g. `[]`, `Maybe`...
- More details will be provided in the in Advanced Programming, including an explanation of `Applicative`, `* -> *` etc.

- `return` allows a sequence, e.g. of I/O actions, to deliver a result which is something other than the result of the final action, e.g.

```
checkInput :: IO Bool
checkInput
  = getChar >>= \c -> return (c == '\n')
```

- Note that we can't just "return" a result in the traditional sense, because we'd get a type error, e.g.

```
getChar >>= \c -> (c == '\n')    -- TYPE ERROR
```

  In this case, `Bool` does not match `IO Bool`

- `fail` describes what to do in the event of failure when implementing the '`do`' syntax – more of that later

Important notes

- All I/O has to happen at the topmost level, because the only way to propagate the world is via `>>` and `>>=`
- For an I/O program to type check, we must *always* use functions like `>>` and `>>=` for the result to be an `IO a`, for some `a`
- Turning it around, if you try to do anything other than I/O at the topmost level, or try to use I/O
  anywhere other than the topmost level, you'll get a type error, e.g.

```
*Main> getChar >>= \c -> (c == '\n')  -- TYPE ERROR


*Main> getChar : "hello"  -- TYPE ERROR
```

- Thus, amazingly, the **the type checker prevents you from violating referential transparency**!

‘`do`’ notation

- Haskell has a ‘`do`’ notation to make it easier to write ‘monadic’ code
- The following are equivalent – indeed, the ‘`do`’ syntax on the left is just sugar for the expression on the right:

```
do a1 ; a2                a1 >> a2
do x <- a1; a2            a1 >>= \x -> a2
```

- The semicolons (`;`) can also be replaced with new lines, provided we indent appropriately

- An example (`putStrLn` and `putStr` print strings with and without new lines):

```
ioTest :: IO ()
ioTest
  = do
      putStr "Please enter a character: "
      c <- getChar
      putStrLn ("\nYou entered '" ++ [c] ++ "'")
```

- This looks very much like I/O code in a conventional programming language; however, it's shorthand for

```
putStr "Please enter a character: " >> getChar >>=
  \c -> putStrLn ("\nYou entered '" ++ [c] ++ "'")
```

- Haskell has many other I/O primitives, e.g. `getLine`, `readFile` etc. Explore...!

Lots of things are monads

- `Maybe`s are monads, e.g.

```
lookupTwice x y table1 table2
  = do
      v  <- lookup x table1
      v' <- lookup y table2
      return (v, v')
```

- A `Nothing` in either lookup gets propagated, e.g.

```
*Main> lookupTwice 1 2 [] []
Nothing
*Main> lookupTwice 1 2 [] [(2,'a')]
Nothing
*Main> lookupTwice 1 2 [(1,5)] []
Nothing
*Main> lookupTwice 1 2 [(1,5)] [(2,'a')]
Just (5,'a')
```

? What does the `Monad` instance of `Maybe` look like?

- Lists are monads too, e.g.

```
*Main> do x <- "abc"; return x
"abc"
*Main> [x | x <- "abc"]
"abc"
*Main> do x <- [1,2,3]; y <- [4,5]; return (x, y)
[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
*Main> [(x, y) | x <- [1,2,3], y <- [4,5]]
[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
```

- Yes - list comprehensions are just shorthands for monadic `do` expressions!

? What does the `Monad` instance of `[]` look like?

- Remark: the `fail` function in the `Monad` class is used to handle pattern matching failure on the left of a `<-`, e.g.

```
*Main> fail "Oops" :: IO ()
*** Exception: user error (Oops)
*Main> fail "Oops" :: Maybe Int
Nothing
*Main> fail "Oops" :: [Int]
[]
*Main> do 'a' <- Just 'z'; return 'a'
Nothing
```

- Thus, when translating `do` blocks, the expression `do pat <- a1; a2` must be mapped to something like:

```
a1 >>= \arg -> case arg of
                 pat = a2
                 _   = fail "Pattern match fail..."
```

- See `GHC.Base` for the details