User defined data types - part 1

Level of difficulty of this handout

This handout includes material of easy, medium, hard and advanced level. If some of the material feels difficult, it is probably because it is difficult rather than your fault. This means you have to work hard if you want to achieve a high mark in the module, as is the case for all modules.

Video lectures for this handout

The following videos are also linked at appropriate points of this handout for your convenience.

1. Introduction, the booleans revisited, isomorphisms, Weekdays and the new `Maybe` type constructor (35 min)
2. Type retracts (13 min)
3. Either and And and pairs (9 min)
4. Lists revisited (9 min)
5. Binary trees (12 min)
6. Directions, addresses and paths in binary trees (15 min)
7. Traversals in binary trees (10 min)
8. Inverting traversals (18 min)

Total 2hrs.

Experimenting with the Haskell code included here

You should experiment with the Haskell code in theses notes in order to achieve full understanding. This means running the code and adding things to it, such as solutions of exercises and puzzles, or your own brilliant ideas.

These lecture notes are in markdown format including Haskell code. To get the Haskell code out of the markdown code, we can use the program `mdtohs.hs` included in the Resources directory, as follows in a Unix/Linux terminal:

```
$ cat Data1.md | runhaskell ../../Resources/mdtohs.hs > Data1.hs
```

This means "copy the contents of the file `Data1.md` to the standard input of the Haskell program `mdtohs.hs` and store the output of the program in the file `Data1.hs` ". This can be equivalently written as

```
$ runhaskell ../../Resources/mdtohs.hs < Data1.md > Data1.hs
```

This removes all the markdown code and keeps only the Haskell code, so that we can work with it.

We have already run this for you, and the file Data1.hs is available in this GitLab repository. Make your own **copy** of this file to avoid conflicts when we update it.

How to run Data1.hs with `ghci`

The import `System.Random` will fail if we don't specify which package it comes from, which is `random` . You specify this as follows:

```
$ ghci -package random Data1.hs
```

Haskell imports in these lecture notes

Any needed library imports should be mentioned here at the top of the file. We need the following for generating random inputs for testing:

```haskell
module Data1 where

import System.Random
```

Contents

Type synonyms

A video discussing the next few sections is available on Canvas.

Sometimes, mainly for the sake of clarity, we may wish to give a new name to an existing type. For example, the Haskell prelude defines a string to be a list of characters:

```
type String = [Char]
```

Since `String` is just a type synonym, operations such as list concatenation and reverse

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

can be freely applied to strings:

```
> "abc" ++ reverse "def"
"abcfed"
```

Type synonyms can also have parameters, as in e.g.

```
type Lst a = [a]
```

User defined data types

The booleans revisited

The booleans are defined as follows in Haskell, in the prelude:

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

This defines a new type, called `Bool`, with two elements (or *constructors*), called `False` and `True`:

```
   False :: Bool
   True  :: Bool
```

Functions over a data type can be conveniently defined by **pattern-matching** on its constructors. For example, in the prelude, the conjunction operation

```
(&&) :: Bool -> Bool -> Bool
```

is defined as follows:

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

A slightly subtle aspect of the semantics of pattern-matching in Haskell is that:

1. the different pattern-matching clauses are tried in order from top to bottom, and
2. the input arguments of the function are only evaluated to the extent needed to check whether they match the current pattern.

A consequence of this semantics is that the above definition of conjunction implements [short-circuit evaluation](): if the first argument is `False`, then the function returns `False` without even evaluating the second argument.

In contrast, consider the following alternative definition of conjunction:

```
conj :: Bool -> Bool -> Bool
conj False False = False
conj False True  = False
conj True  False = False
conj True  True  = True
```

This version does *not* implement short-circuit evaluation: the second argument will always be evaluated regardless of the value of the first. We can observe the difference between these two versions of conjunction by running the following experiment in the GHC interpreter:

```
> False && undefined
False
> False `conj` undefined
*** Exception: Prelude.undefined
#CallStack (from HasCallStack):
  error, called at libraries/base/GHC/Err.hs:79:14 in base:GHC.Err
  undefined, called at <interactive>:28:11 in interactive:Ghci5
```

Type isomorphisms

Let's introduce another data type `BW` defined by

```
data BW = Black | White
```

This type is *isomorphic* to the type `Bool`, via the type-conversion functions

```
bw2bool :: BW -> Bool
bw2bool Black = False
bw2bool White = True

bool2bw :: Bool -> BW
bool2bw False = Black
bool2bw True  = White
```

That the pair of functions `(bw2bool,bool2bw)` is an isomorphism means that they are mutually inverse, in the sense that

```
   bw2bool(bool2bw b) = b
```

for all `b :: Bool`, and

```
   bool2bw(bw2bool c) = c
```

for all `c :: BW`.

Type isomorphisms should *not* be confused with type synonyms. For example, if we try to directly use a value of type `BW` where a value of type `Bool` is expected, we get a type error:

```
> let test = Black && True

<interactive>:39:1: error:
    • Couldn't match expected type ‘Bool’ with actual type ‘BW’
    • In the first argument of ‘(&&)’, namely ‘Black’
      In the expression: Black && True
```

```
    In an equation for 'it': it = Black && True
```

On the other hand, if we wrap up the values using the explicit coercions `bw2bool` and `bool2bw`, then everything checks:

```
> let test = bool2bw (bw2bool Black && True)
```

Of course, the names `Black` and `White` are arbitrary, and there is another isomorphism between `BW` and `Bool` that swaps `Black` with `True` and `White` with `False` instead.

```
bw2bool' :: BW -> Bool
bw2bool' Black = True
bw2bool' White = False

bool2bw' :: Bool -> BW
bool2bw' False = White
bool2bw' True  = Black
```

And both of the types `Bool` and `BW` are of course isomorphic (again in two different ways each) to the type

```
data Bit = Zero | One
```

of binary digits. One of the isomorphisms is the following:

```
bit2Bool :: Bit  -> Bool
bool2Bit :: Bool -> Bit

bit2Bool Zero  = False
bit2Bool One   = True

bool2Bit False = Zero
bool2Bit True  = One
```

Another one is the following:

```
bit2Bool' :: Bit  -> Bool
bool2Bit' :: Bool -> Bit

bit2Bool' Zero  = True
bit2Bool' One   = False

bool2Bit' False = One
bool2Bit' True  = Zero
```

Note: The syntax rules of Haskell require that both type names (here `Bool`, `BW`, `Bit`) and constructor names (here `False`, `True`, `Black`, `White`, `Zero`, `One`) should start with a capital letter.

Weekdays

Another example of a data type is

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

We can ask Haskell to do some jobs for free for us (there are alternative ways of doing them ourselves with our own sweat, using type class instances, which we will discuss later):

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

This automatically adds the type `WeekDay` to the type classes with these five names, which give functions

```
   show :: WeekDay -> String
   read :: String -> WeekDay
   (==) :: WeekDay -> WeekDay -> Bool
   (<), (>), (<=), (>=) :: WeekDay -> WeekDay -> Bool
   succ, pred :: WeekDay -> WeekDay
```

Look this up in our adopted textbook. Notice that `show` is the counterpart of Java's `toString`, and `read` does the opposite. Some examples are:

```
> show Tue
"Tue"
> read "Tue" :: WeekDay  -- (the type annotation tells Haskell to try to parse the string as a We
Tue
> read "Dog" :: WeekDay
*** Exception: Prelude.read: no parse
> Mon == Tue
False
> Mon < Tue
True
> succ Mon
Tue
> pred Tue
Mon
> [Mon .. Fri]
[Mon,Tue,Wed,Thu,Fri]
```

Monday doesn't have a predecessor, and Sunday doesn't have a successor:

```
> pred Mon
*** Exception: pred{WeekDay}: tried to take `pred' of first tag in enumeration
CallStack (from HasCallStack):
  error, called at Data1.hs:20:47 in main:Main
> succ Sun
*** Exception: succ{WeekDay}: tried to take `succ' of last tag in enumeration
CallStack (from HasCallStack):
  error, called at Data1.hs:20:47 in main:Main
```

Notice that the choice of names in the type of weekdays is arbitrary. An equally good, isomorphic definition is

```
data WeekDay' = Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday
```

Some important type constructors

The `Maybe` type constructor

Sometimes a function may not be able to give a result, in which case we would like it to say explicitly that it cannot. We use the `Maybe` type from the prelude for that purpose:

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

Here `a` is a type parameter, and we have the following types for `Nothing` and `Just`:

```
   Nothing :: Maybe a
   Just    :: a -> Maybe a
```

This means that the constructor `Just` is a function. It converts an element of type `a` into an element of type `Maybe a`. So this function `Just` is a so-called *type coercion*, also known as a *type cast*.

For example:

```
   Just 17 :: Maybe Integer
```

In summary, the only possible elements of the type `Maybe a` are `Nothing` and `Just x` where `x` has type `a`.

In Java the `Maybe` type constructor is called `Optional`.

Example: integer computations that may give errors

In the following definition of division, if the denominator is zero, then division is impossible, and so the result is `Nothing`. If it is possible, we simply perform the division, and we convert the resulting `Int` to `Maybe Int` using the type conversion function `Just`, which then gives the result of the function `dive`:

```
dive :: Int -> Int -> Maybe Int
x `dive` y = if y == 0 then Nothing else Just (x `div` y)
```

For example, we get

```
> 10 `dive` 2
Just 5
> 10 `dive` 0
Nothing
```

But now suppose you want to do `3 + (10 `dive` 0)`. You would expect `Nothing` but instead this expression doesn't even type check:

```
> 3 + (10 `dive` 0)

<interactive>:11:1: error:
    • No instance for (Num (Maybe Int)) arising from a use of ‘+’
    • In the expression: 3 + (5 `dive` 0)
      In an equation for ‘it’: it = 3 + (5 `dive` 0)
```

What this is saying is that `(10 `div` 0)` is expected to be an `Int`, because `+` expects an `Int` as its right argument, but it isn't. That's because it is a `Maybe Int`. So we need a version of addition that can cope with errors as its possible inputs:

```
adde :: Maybe Int -> Maybe Int -> Maybe Int
adde Nothing  Nothing  = Nothing
adde Nothing  (Just y) = Nothing
adde (Just x) Nothing  = Nothing
adde (Just x) (Just y) = Just (x + y)
```

Now to fix `3 + (5 `dive` 0)` we replace `+` by `adde`, but also we need to convert the number `3` to the type `Maybe Int` with the *type coercion* or *type cast* `Just`.

```
> Just 3 `adde` (10 `dive` 2)
Just 8
> Just 3 `adde` (10 `dive` 0)
Nothing
```

A more concise definition of `adde`:

```
adde' :: Maybe Int -> Maybe Int -> Maybe Int
adde' (Just x) (Just y) = Just (x+y)
adde' _        _        = Nothing
```

This works because the execution of Haskell programs tries patterns from top to bottom, and the last pattern catches all remaining possibilities.

A definition using `cases` is also possible:

```
adde'' :: Maybe Int -> Maybe Int -> Maybe Int
adde'' xm ym = case xm of
                Nothing -> Nothing
                Just x  -> case ym of
                            Nothing -> Nothing
                            Just y  -> Just (x+y)
```

Later we will see that there is a much more concise way of making such definitions using *monads*. But for now we will stick to pattern matching and cases.

Example: find the first position an element occurs in a list

Since the first position is undefined if the element doesn't occur in the list, in that case we answer `Nothing`:

```
firstPosition :: Eq a => a -> [a] -> Maybe Int
firstPosition x []     = Nothing
firstPosition x (y:ys)
           | x == y    = Just 0
           | otherwise = case firstPosition x ys of
                           Nothing -> Nothing
                           Just n  -> Just (n+1)
```

For example:

```
> firstPosition 'a' ['a'..'z']
Just 0
```

```
> firstPosition 'b' ['a'..'z']
Just 1
> firstPosition 'z' ['a'..'z']
Just 25
> firstPosition '!' ['a'..'z']
Nothing
```

which we summarize as

```
    firstPosition 'a' ['a'..'z'] = Just 0
    firstPosition 'b' ['a'..'z'] = Just 1
    firstPosition 'z' ['a'..'z'] = Just 25
    firstPosition '!' ['a'..'z'] = Nothing
```

A precise specification of `firstPosition` is that if `firstPosition x ys = Just n` then `ys !! n = x`, and if `firstPosition x ys = Nothing` then `ys !! i ≠ x for all` `i` `in the list` [0..length ys-1]`. We can actually use this specification to test our implementation for correctness:

```
testFirstPosition :: Eq a => a -> [a] -> Bool
testFirstPosition x ys =  case firstPosition x ys of
                           Nothing -> and [ ys !! i /= x | i <- [0 .. length ys - 1]]
                           Just n  -> ys !! n == x
```

Here are some tests:

```
> testFirstPosition 'a' ['a'..'z']
True
> testFirstPosition 'b' ['a'..'z']
True
> testFirstPosition 'z' ['a'..'z']
True
> testFirstPosition '!' ['a'..'z']
True
```

All tests are successful, and so we get some confidence about the correctness of our implementation. Of course, it is not possible to prove correctness by testing all cases, as there is an infinite amount of them.

You are required to use the book to find out what `case` is and how it works in general, but in this example it should be clear. You are also required to use the book to find out about conditional definitions using `|` to indicate *guards* for equations.

We will use the `Maybe` type constructor very often, because there are many occasions in which some inputs are *invalid*.

Task. Define a function `allPositions :: Eq a => a -> [a] -> [Int]` that finds all positions in which an element element occurs in a list. For example, we should have `allPositions 17 [13,17,17,666] = [1,2]` and `allPositions 17 [1,2,3] = []`.

Type retracts

A video discussing this section is available.

This section may be rather hard at your current level. Consider it as a challenge. If you get lost, feel free to skip it at a first reading, go to the next section, and come back to it later. This is important because it deals with data coding, which is a crucial, aspect of Computer Science. What we say here is in Haskell, but it actually applies to any programming language.

We have already seen examples of type *isomorphisms*. For example, the type `Bool` is isomorphic to the type `BW` of black-and-white colours, and also to the type `Bit` of binary digits `Zero` and `One`.

More generally, an isomorphism of types `a` and `b` is a pair of functions

```
f :: a -> b
g :: b -> a
```

such that

- `f (g y) = y` for all `y :: b`, and
- `g (f x) = x` for all `x :: a`.

We summarize these two equations by saying that these two functions are *mutually inverse*. This means that we can convert back and forth between elements of the type `a` and elements of the type `b`, like we did when we converted `False` to `Zero` and `True` to `One`. You can think of `f` and `g` as *renaming* functions: the function `f = bit2Bool` renames a bit to a boolean, and the function `g = bool2Bit` renames a boolean to a bit.

In practice, this means that it doesn't matter whether we work with the type `Bool` with elements `True` and `False`, or the type `Bit` with elements `Zero` and `One`. In fact, computers work by exploting this identification of booleans with binary digits.

There is another relationship between types that is also useful in practice: a type `b` can "live" inside another type `a`, in the sense of having a "copy" in the type `a`. A simple example is that the type `Bool` has a copy inside the type `Int`:

```
bool2Int :: Bool -> Int
bool2Int False = 0
bool2Int True  = 1
```

Not only do we have a copy of `Bool` inside `Int`, but also we can go back, so that we get `False` and `True` from `0` and `1`:

```
int2Bool :: Int -> Bool
int2Bool n | n == 0    = False
           | otherwise = True
```

However, notice that not only `1` is converted back to `True`, but also everything other than `0` is converted to `True`.

We have

```
    int2Bool (bool2Int y) = y
```

for every `y :: Bool`, but we don't have `bool2Int (int2Bool x) = x` for all `x :: Int`, as this fails for e.g. `x = 17` because `bool2Int (int2Bool 17)` is `1` rather than `17`.

We can say that there is enough room in the type integers for it to host a copy of the type of booleans, but there isn't enough room in the type of booleans for it to host a copy of the type of integers.

When there are functions

```
f :: a -> b
g :: b -> a
```

such that

- `f (g y) = y` for all `y :: b`,

but not necessarily `g (f x) = x` for all `x :: a`, we say that the type `b` is a *retract* of the type `a`

Our discussion above shows that the type `Bool` is a retract of the type `Int`. This retraction is the same as that performed in the programming language `C`, where the integer `0` codes `False` and everything else codes `True`.

But notice that there are other ways in which the type `Bool` lives inside the type `Int` as a retract: for example, we can send `False` to `13` and `True` to `17`, and then send back everything bigger than `15` to `True` and everything else to `False`. We are free to code things as we wish.

Task. Show that the type `Maybe a` is a retract of the type `[a]`. The idea is that `Nothing` corresponds to the empty list `[]` and that `Just x` corresponds to the one-element list `[x]`. Make this idea precise by writing back and forth functions between these types so that they exhibit `Maybe a` as a retract of `[a]`. Our adopted textbook exploits such a retraction often, albeit without making it explicit. In fact, what the book does very often is to avoid the type `Maybe a` and instead work with the type `[a]`, considering only the list `[]` (corresponding to `Nothing`) and singleton lists `[x]` (corresponding to `Just x`), and ignoring lists of length `2` or greater. (The reason the book does that is to avoid monads (before they are taught) in favour of list comprehensions (which are taught early on), as list comprehensions happen to accomplish the same thing as "do notation" for monads, in the particular case of the list monad. So this coding is done for pedagogical purposes in this case.)

If we have a type retraction `(f,g)` as above, then:

- `f` is a **surjection**.

  This means that for every `y :: b` there is **at least one** `x :: a` with `f x = y`.

  For example, in the case of the booleans as a retract of the integers, this means that every boolean is coded by at least one integer.

- `g` is an **injection**.

  This means that for every `x :: a` there is **at most one** `y :: b` with `g y = x`.

  In the first example of the booleans as a retract of the integers, this is the case:

  - For `x = 0` we have exactly one `y` with `bool2Int y = x`, namely `y=False`.

- For `x = 1` we have exactly one `y` with `bool2Int y = x`, namely `y=True`.
- For `x` different from `0` and `1` we have no `y` with `bool2Int y = x`.

So for every `x` there is at most one such `y` (i.e. exactly one or none).

Task. Define

```
data WorkingWeekDay = Mon' | Tue' | Wed' | Thu' | Fri'
```

We add primes to the names because the names without prime are already used as elements of the type `WeekDay` defined above. Show that the type `WorkingWeekDay` is a retract of the type `WeekDay`. Arbitrary choices will need to be performed in one direction, like e.g. the language `C` arbitrarily decides that any integer other than `0` codes `true`.

Puzzle. Consider the function

```
g :: Integer -> (Integer -> Bool)
g y = \x -> x == y
```

We can visualize `g` in the following table:

	...	-5	-4	...	-1	0	1	...	4	5	...
g(-5)=	...	True	False	...	False	False	False	...	False	False	...
g(-4)=	...	False	True	...	False	False	False	...	False	False	...
...											
g(-1)=	...	False	False	...	True	False	False	...	False	False	...
g(0)=	...	False	False	...	False	True	False	...	False	False	...
g(1)=	...	False	False	...	False	False	True	...	False	False	...
...											
g(4)=	...	False	False	...	False	False	False	...	True	False	...
g(5)=	...	False	False	...	False	False	False	...	False	True	...

That is, the function `g` codes the integer `y` as the function `h` such that `h y = True` and `h x = False` for `x` other than `y`. Convince yourself that the function `g` is an injection. In this sense, the type `Integer` lives inside the function type `Integer -> Bool`. Do you think `g` has a companion `f : (Integer -> Bool) -> Integer` that "decodes" functions `Integer -> Bool` back to integers, such that for any code `g y` of the integer `y` we get the integer back as `f (g y) = y`? If yes, then give a Haskell definition of such an `f` and convince yourself that indeed `f (g y) = y` for all integers `y`. If not, why? This puzzle is rather tricky, and none of the possible answers "yes" or "no" to the question is obvious.

The `Either` type constructor

A video discussing the next few sections is [available on Canvas](#).

It is defined in the prelude as follows:

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

Then we have

```
    Left  :: a -> Either a b
    Right :: b -> Either a b
```

For example:

```
    Left 17     :: Either Integer String
    Right "abd" :: Either Integer String
```

The idea is that the type `Either a b` is the *disjoint union* of the types `a` and `b`, where we tag the elements of `a` with `Left` and those of `b` with `Right` in the union type. An example of its use is given below.

The `And` type constructor, defined by ourselves

The following has an isomorphic version predefined in the language, as we shall see soon:

```
data And a b = Both a b
```

This is a type constructor with two parameters, and with an element constructor `Both`, which is a function

```
   Both :: a -> b -> And a b
```

For example, assuming we have defined types `MainDish`, `Dessert`, `Drink`,

```
data MainDish = Chicken | Pasta | Vegetarian
data Dessert = Cake | IceCream | Fruit
data Drink = Tea | Coffee | Beer
```

we can define:

```
type SaverMenu = Either (And MainDish Dessert) (And MainDish Drink)
```

which can be equivalently written

```
type SaverMenu = Either (MainDish `And` Dessert) (MainDish `And` Drink)
```

(Choose which form of definition you like better. Haskell accepts both.)

So what is available in the saver menu is either a main dish and a dessert, or else a main dish and a drink. It should be intuitively clear that this is isomorphic to

```
type SaverMenu' = And MainDish (Either Dessert Drink)
```

meaning that you can have a main dish and either dessert or a drink. This intuition is made precise by the isomorphism

```
prime :: SaverMenu -> SaverMenu'
prime (Left (Both m d)) = Both m (Left  d)
prime (Right(Both m d)) = Both m (Right d)

unprime :: SaverMenu' -> SaverMenu
unprime (Both m (Left  d)) = Left (Both m d)
unprime (Both m (Right d)) = Right(Both m d)
```

So, as a software developer, you can choose either `SaverMenu` as your implementation, or else `SaverMenu'`. They are different, but essentially equivalent.

We actually don't need to define `And`, because an equivalent type constructor is already available in Haskell, namely the type of pairs. We have an isomorphism as follows:

```
and2pair :: And a b -> (a,b)
and2pair (Both x y) = (x,y)

pair2and :: (a,b) -> And a b
pair2and (x,y) = Both x y
```

And so we have further isomorphic versions of the saver menu type:

```
type SaverMenu''  = Either (MainDish, Dessert) (MainDish, Drink)
type SaverMenu''' = (MainDish, Either Dessert Drink)
```

Lookup the type of pairs (tuple types) in the book and read about it.

Lists revisited

A video discussing the next few sections is [available on Canvas](#).

With a pinch of salt, the type of lists is predefined by

```
data [a] = [] | a : [a]  -- not quite a Haskell definition
```

which says that a list of `a`'s is either empty, or else an element of the type `a` followed (indicated by `:`) by a list of a 's. This is an example of a *recursive* data type definition. We have the following types for the list constructors:

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

Although the above not-quite-a-Haskell-definition is semantically correct, it is syntactically wrong, because Haskell (unfortunately) doesn't accept this kind of syntactical definition. If we don't care about syntax, we can define an isomorphic version as follows:

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

Then the types of the constructors are

```haskell
   Nil  :: List a
   Cons :: a -> List a -> List a
```

For example, the native list `[1,2,3]` is written `Cons 1 (Cons 2 (Cons 3 Nil))` in our isomorphic version of the type of lists. Let's define the isomorphism to make this clear:

```haskell
nativelist2ourlist :: [a] -> List a
nativelist2ourlist []     = Nil
nativelist2ourlist (x:xs) = Cons x (nativelist2ourlist xs)

ourlist2nativelist :: List a -> [a]
ourlist2nativelist Nil         = []
ourlist2nativelist (Cons x xs) = x:ourlist2nativelist xs
```

Notice that these coercions are defined recursively, corresponding to the fact that the data type itself is defined recursively.

Implementing some basic operations on lists

Let's write our own versions of the list concatenation ("append") and reverse operations from the prelude:

```haskell
append :: List a -> List a -> List a
append Nil         ys = ys
append (Cons x xs) ys = Cons x (append xs ys)

rev :: List a -> List a
rev Nil         = Nil
rev (Cons x xs) = rev xs `append` (Cons x Nil)
```

We can try to test that these do the right thing by comparing them to the implementations of list concatenation and reversal in the Haskell prelude, using the isomorphism between `List a` and `[a]`. Indeed, we expect that

```haskell
ourlist2nativelist (append (nativelist2ourlist xs) (nativelist2ourlist ys)) == xs ++ ys
```

and

```haskell
ourlist2nativelist (rev (nativelist2ourlist xs)) == reverse xs
```

should evaluate to `True` for all native lists `xs, ys :: [a]`. Let's test these properties:

```
> let xs = [1..5]
> let ys = [6..10]
> ourlist2nativelist (append (nativelist2ourlist xs) (nativelist2ourlist ys)) == xs ++ ys
True
> ourlist2nativelist (rev (nativelist2ourlist xs)) == reverse xs
True
```

Of course, here we have only tested on a couple examples, but it is true in general. (Question: how would you *prove* this?)

Although our definitions are functionally correct, there is a more subtle problem with our implementation of `rev`. By inspection of the code, `append xs ys` computes the concatenation of two lists in time O(n), where n is the length of `xs`, since each recursive call to `append` decreases the length of `xs` by one, and the calls to `Cons` are constant time. On the other hand, `rev` is O(n^2) by the same argument, since each recursive call to `rev` decreases the length of `xs` by one, and each call to `append` is O(n).

This is not just a theoretical problem — we quickly bump into it if we compare reversing a reasonably large list using the native `reverse` function versus the implementation `rev` above.

```
> let xs = [1..10^5]
> length (reverse xs)  -- this is fast (we return the length of the reversed list in order to kee
100000
> length (ourlist2nativelist (rev (nativelist2ourlist xs)))  -- this is really slow, so we give u
  C-c C-cInterrupted.
```

There's a much more efficient way of implementing reversal by introducing a helper function with an extra argument:

```
fastrev :: List a -> List a
fastrev xs = revapp xs Nil
  where
    revapp :: List a -> List a -> List a
    revapp (Cons x xs) ys = revapp xs (Cons x ys)
    revapp Nil         ys = ys
```

One way to think of the second argument of the helper function `revapp` is as a stack, initially set to be empty (`Nil`). The function recursively scans the input from the first argument, pushing each element onto the stack (second argument). When there are no more input elements, the stack is simply popped directly to the output, with all of the elements of the original list now in reverse order.

Here's a concrete illustration of how this works, unrolling the definitions of `fastrev` and `revapp` to reverse a four-element list:

```
  fastrev (Cons 1 (Cons 2 (Cons 3 (Cons 4 Nil))))
= revapp (Cons 1 (Cons 2 (Cons 3 (Cons 4 Nil)))) Nil
= revapp (Cons 2 (Cons 3 (Cons 4 Nil))) (Cons 1 Nil)
= revapp (Cons 3 (Cons 4 Nil)) (Cons 2 (Cons 1 Nil))
= revapp (Cons 4 Nil) (Cons 3 (Cons 2 (Cons 1 Nil)))
= revapp Nil (Cons 4 (Cons 3 (Cons 2 (Cons 1 Nil))))
= Cons 4 (Cons 3 (Cons 2 (Cons 1 Nil)))
```

Another way of thinking of the function `revapp` is suggested by its name: given two lists `xs` and `ys` , we have that `revapp xs ys` computes the reversal of `xs` *appended* with `ys` . It's not hard to see that this binary operation `revapp` is *more general* than the original unary reversal operation: the latter can be recovered by taking `ys = Nil` . On the other hand, `revapp` is much more efficient than our original function `rev` , being only O(n) in the length of its first argument `xs` .

This pattern — where we manage to solve a problem or solve it more efficiently by replacing it with a more general and seemingly more difficult problem — happens again and again in functional programming.

An aside on accumulators

The extra argument `ys` that we used in the helper function `revapp` is sometimes called an "accumulator", since it accumulates a value that is eventually passed to the output. Above we saw how an accumulator could be used to turn an $O(n^2)$ algorithm into an O(n) algorithm for list reversal. For an even starker example, consider the problem of computing the [Fibonacci numbers](#) F_n.

The mathematical definition of the Fibonacci sequence in Wikipedia may be translated directly into the following Haskell code:

```
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

But although this definition is correct, it is extremely inefficient!

We can already see this if we try to use the above definition to compute, say, the first 32 Fibonacci numbers:

```
> :set +s -- ask ghci to print time and space usage
> [fib n | n <- [0..31]]
[0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584,4181,6765,10946,17711,28657,46368,750
(10.23 secs, 4,086,282,024 bytes)
```

Over ten seconds to compute 32 Fibonacci numbers! Indeed, the running time of `fib n` is roughly $O(2^n)$, since the recursive case makes two calls to `fib` while only decreasing `n` by 1 or 2.

Here is an alternative, much more efficient implementation using a pair of accumulators `x` and `y` :

```
fastfib n = fibAcc n 0 1
  where
```

```
    fibAcc 0 x y = x
    fibAcc 1 x y = y
    fibAcc n x y = fibAcc (n-1) y (x+y)
```

With this implementation we have no trouble computing, say, the first 100 Fibonacci numbers in a fraction of a second:

```
> [fastfib n | n <- [0..99]]
[0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584,4181,6765,10946,17711,28657,46368,750
(0.02 secs, 3,057,552 bytes)
```

Again to see what is going on it is helpful to unroll the definitions on a concrete example:

```
  fastfib 7
= fibAcc 7 0 1
= fibAcc 6 1 1
= fibAcc 5 1 2
= fibAcc 4 2 3
= fibAcc 3 3 5
= fibAcc 2 5 8
= fibAcc 1 8 13
= 13
```

We can see that this functional implementation of the Fibonacci numbers using a pair of accumulator arguments `x` and `y` works very similarly to the way one might compute the Fibonacci numbers in Java, by updating a pair of variables `x` and `y` inside a loop:

```
static int fastfib(int n) {
  int x = 0, y = 1;
  while (n > 1) {
     int z = x+y;
     x = y;
     y = z;
     n = n-1;
  }
  return (n == 0 ? x : y);
}
```

In addition to this low-level view of what `fastfib` and in turn `fibAcc` is doing, there is also a more high-level view (as we saw before in the case of `revapp`). Can you identify a sense in which the helper function `fibAcc n x y` computes a *more general* function than `fib n`?

Binary trees

A video discussing the next few sections is [available on Canvas](#).

A binary tree over elements of a type `a` is either empty, or consists of a root labelled by an element of the type `a` followed by two binary trees, called the left and right subtrees:

```
data BT a = Empty | Fork a (BT a) (BT a)
```

- We have the empty tree, to be called `Empty`. We take the convention that empty trees are drawn as dangling leaves.

```
                              |
```

- Given two trees `l` and `r` and an element `x::a`, we have the new tree

```
                              x
                             / \
                            /   \
                           l     r
```

  written `Fork x l r`.

For example, the tree

```
                       8
                      / \
                     /   \
```

```
                            4      16
                           / \    / \
                          2         20
                         / \       /  \
```

is written, in this notation, as

```haskell
btexample = Fork 8 (Fork 4 (Fork 2 Empty Empty) Empty) (Fork 16 Empty (Fork 20 Empty Empty))
```

We can ask Haskell to do some work for us by deriving things as above.

```haskell
data BT a = Empty
          | Fork a (BT a) (BT a) deriving (Show, Read, Eq, Ord)
```

We have that

```haskell
   Empty :: BT a
   Fork  :: a -> BT a -> BT a -> BT a
```

Puzzle. It should be clear what the automatically derived `Show`, `Read` and `Eq` do. But what do you think the order on trees derived with `Ord` should be? *Hint.* This is a non-trivial question. So examine it first for the type of lists. In that case, the automatically derived order is the [lexicographic order](), which is like the dictionary order.

Basic functions on binary trees

To get started, let's mirror trees, so that e.g. from the above we get

```
                              8
                             / \
                            /   \
                           16     4
                          / \   / \
                         20        2
                        / \       / \
```

We do this as follows:

```haskell
mirror :: BT a -> BT a
mirror Empty = Empty
mirror (Fork x l r) = Fork x (mirror r) (mirror l)
```

Running this on the above example we get

```haskell
    mirror btexample = Fork 8 (Fork 16 (Fork 20 Empty Empty) Empty) (Fork 4 Empty (Fork 2 Empty E
```

This notation for trees is not very good for visualizing them, as you can see, but is very good for computation.

We define the *size* of a tree as its total number of nodes:

```haskell
size :: BT a -> Integer
size Empty        = 0
size (Fork x l r) = 1 + size l + size r
```

Since we are considering binary trees, the size (i.e., the number of nodes) is also equal to the number of leaves minus one:

```haskell
leaves :: BT a -> Integer
leaves Empty        = 1
leaves (Fork x l r) = leaves l + leaves r
```

We define the *height* of a tree to be the length of the longest path from the root, measured in number of nodes:

```haskell
height :: BT a -> Integer
height Empty        = 0
height (Fork x l r) = 1 + max (height l) (height r)
```

A balanced binary tree has height approximately log of its size, whereas a binary tree which is very unbalanced, such as

```
                    20
                   / \
                  16
                 / \
                8
               / \
              4
             / \
            2
           / \
```

```
btleft = Fork 20 (Fork 16 (Fork 8 (Fork 4 (Fork 2 Empty Empty) Empty) Empty) Empty) Empty
```

has height approximately equal to its size.

Directions, addresses and paths in binary trees

A video discussing the next few sections is available on Canvas.

To pick a subtree of a binary tree, we go left or right successively, until we find it. But a wrong list of directions, called an address here, may be given, and hence we need the `Maybe` type for the output:

```
data Direction = L | R deriving (Show)
type Address   = [Direction]

subtree :: Address -> BT a -> Maybe(BT a)
subtree []     t            = Just t
subtree (_:_)  Empty        = Nothing
subtree (L:ds) (Fork _ l _) = subtree ds l
subtree (R:ds) (Fork _ _ r) = subtree ds r
```

Following the above pattern, we can define a function that checks whether an address in a given tree is valid:

```
isValid :: Address -> BT a -> Bool
isValid []     _            = True
isValid (_:_)  Empty        = False
isValid (L:ds) (Fork _ l _) = isValid ds l
isValid (R:ds) (Fork _ _ r) = isValid ds r
```

The list of valid addresses for subtrees can be computed as follows:

```
validAddresses :: BT a -> [Address]
validAddresses Empty        = [[]]
validAddresses (Fork _ l r) = [[]]
                           ++ [L:ds | ds <- validAddresses l]
                           ++ [R:ds | ds <- validAddresses r]
```

List comprehensions can always be eliminated. In this example they become

```
validAddresses' :: BT a -> [Address]
validAddresses' Empty        = [[]]
validAddresses' (Fork _ l r) = [[]]
                            ++ (map (L:) (validAddresses' l))
                            ++ (map (R:) (validAddresses' r))
```

We expect that

```
    isValid ds t = ds `elem` (validAddresses t)
```

Or, in words, an address is valid if and only if it is an element of the list of valid addresses. Should this be intuitively clear? The statement, yes. But the fact, given our definitions, I don't think so. I would say that it requires a convincing argument. In any case, intuition is something we develop based on convincing arguments we learn.

The list of all paths from the root to a leaf has a similar definition:

```
btpaths :: BT a -> [[a]]
btpaths Empty        = [[]]
btpaths (Fork x l r) = [x:xs | xs <- btpaths l]
                    ++ [x:xs | xs <- btpaths r]
```

Proofs on binary trees by induction

If we have a property `P` of trees, and we want to show that `P(t)` holds for all trees `t`, we can do this by *induction on trees* as follows:

- Argue that `P(Empty)` holds.
- Argue that if `P(l)` and `P(r)` hold for given trees `l` and `r`, then it holds for `P(Fork x l r)` where `x` is arbitrary.

We are not going to emphasize proofs in this module, but we will indicate when some claims genuinely require proofs, and, moreover, we will try to be precise regarding the specifications of the programs we write.

It is often the case that somebody shows us a clever algorithm and we feel stupid because we don't understand it. But this feeling is wrong. If we don't understand an algorithm, what is missing is a proof. A proof is an explanation. This is what proof means. In order to understand an algorithm we need

- the algorithm itself,
- a precise description of what it is intended to do, and
- a convincing explanation that the algorithm does do what it is intended to do.

Programs alone are not sufficient. We need to know what they are intended to accomplish, and we want to know an explanation justifying that they accomplish what we promise. This promise is called the *specification* of the algorithm / program. Program correctness means "the promise is fulfilled". One way to attempt to prove that the promise is fulfilled is to *test* the program. But actually, all that testing can do it to show that the promise is *not* fulfilled, by finding counterexamples. When the good examples work, we have some kind of evidence that the algorithm works, but not full confidence, because we may have missed examples of inputs that give wrong outputs. Full confidence can only be given by a convincing explanation, also known as *proof*. If you ever asked yourself what "proof" really means, the ultimate answer is "convincing argument".

Functional proofs

The dependently typed language Agda allows to write functional programs *and* their correctness proofs, where the proofs themselves are written as functional programs. As an example, here is a computer-checked proof of the above relation between the functions `isValid` and `validAddresses` in Agda. This is not examinable, and is included here for the sake of illustration only.

Traversals in binary trees

A video discussing the next few sections is available on Canvas.

We now define the standard in-order and pre-order traversals.

```
treeInOrder :: BT a -> [a]
treeInOrder Empty = []
treeInOrder (Fork x l r) = treeInOrder l ++ [x] ++ treeInOrder r

treePreOrder :: BT a -> [a]
treePreOrder Empty = []
treePreOrder (Fork x l r) = [x] ++ treePreOrder l ++ treePreOrder r
```

For instance, for the trees `btexample` and `btleft` considered above,

```
                             8
                            / \
     btexample =           /   \
                          4     16
                         / \   / \
                        2        20
                       / \      /  \
```

```
                            20
                           / \
                          16
                         / \
        btleft =        8
                       / \
                      4
                     / \
                    2
                   / \
```

we get:

```
> (treeInOrder btexample, treePreOrder btexample)
([2,4,8,16,20],[8,4,2,16,20])
> (treeInOrder btleft, treePreOrder btleft)
([2,4,8,16,20],[20,16,8,4,2])
```

Breadth-first traversal is trickier. We first define a function that given a tree, produces a lists of lists, with the nodes of level zero (just the root), then the nodes of level one (the successors of the root), then the nodes of level two, and so on:

```
levels :: BT a -> [[a]]
levels Empty        = []
levels (Fork x l r) = [[x]] ++ zipappend (levels l) (levels r)
  where
    zipappend []       yss      = yss
    zipappend xss      []       = xss
    zipappend (xs:xss) (ys:yss) = (xs ++ ys) : zipappend xss yss
```

(Compare `zipappend` to the prelude function zipWith.) For example:

```
> levels btexample
[[8],[4,16],[2,20]]
> levels btleft
[[20],[16],[8],[4],[2]]
```

With this we can define

```
treeBreadthFirst :: BT a -> [a]
treeBreadthFirst = concat . levels
```

where `.` stands for function composition (look it up in our textbook), and the prelude function `concat :: [[a]] -> [a]` concatenates a list of lists, for example getting `[8,4,16,2,20]` from `[[8],[4,16],[2,20]]`. For further discussion about breadth-first search, see The under-appreciated unfold (a free version is at the authors' web page), but this is probably beyond your current level for most of you.

Inverting traversals (generating trees)

A video discussing the next few sections is available on Canvas.

Many different trees can have the same (in-order/pre-order/breadth-first) traversal, as we saw above with `btexample` and `btleft`, which have the same in-order traversal. In other words, all of the functions

```
treeInOrder, treePreOrder, treeBreadthFirst :: BT a -> [a]
```

are *non-injective* and hence non-invertible. Nonetheless, an interesting and tractable problem is to try to construct a binary tree with a given (in-order/pre-order/breadth-first) traversal, or even to generate *all possible binary trees* with a given traversal.

As an example, the following will produce a *balanced* binary tree given its in-order traversal (which will be a binary *search* tree if the input is sorted):

```
balancedTree :: [a] -> BT a
balancedTree [] = Empty
balancedTree xs = let (ys, x:zs) = splitAt (length xs `div` 2) xs in
                  Fork x (balancedTree ys) (balancedTree zs)
```

(The prelude function `splitAt` splits a list in two lists at a given position.) This satisfies the equation

```
    treeInOrder (balancedTree xs) = xs
```

for all `xs :: [a]`. In the other direction, it is certainly **not** the case that

```
    balancedTree (treeInOrder t) = t
```

for all `t :: BT a`, for instance

```
balancedTree (treeInOrder btleft) = Fork 8 (Fork 4 (Fork 2 Empty Empty) Empty) (Fork 20 (Fork 16
```

which is not equal to `btleft`. Indeed, the composite function

```
balance :: BT a -> BT a
balance = balancedTree . treeInOrder
```

which applies `treeInOrder` to a tree followed by `balancedTree` to the resulting list can be seen as an operation for rebalancing a binary tree.

Now, using list comprehensions, it is a small step from the function `balancedTree` above to a function generating *all* binary trees with a given in-order traversal.

```
inOrderTree :: [a] -> [BT a]
inOrderTree [] = [Empty]
inOrderTree xs = [Fork x l r | i <- [0..length xs-1],
                               let (ys, x:zs) = splitAt i xs,
                               l <- inOrderTree ys, r <- inOrderTree zs]
```

This satisfies the property that

```
elem t (inOrderTree xs)
```

if and only if

```
treeInOrder t = xs
```

for all `t :: BT a` and `xs :: [a]`. For example, running

```
> inOrderTree [1..3]
[Fork 1 Empty (Fork 2 Empty (Fork 3 Empty Empty)),Fork 1 Empty (Fork 3 (Fork 2 Empty Empty) Empty
```

successfully computes all five binary search trees whose in-order traversal is `[1,2,3]`:

```
   1
  / \
     2
    / \
       3
      / \
Fork 1 Empty (Fork 2 Empty (Fork 3 Empty Empty))

   1
  / \
     3
    / \
   2
  / \
Fork 1 Empty (Fork 3 (Fork 2 Empty Empty) Empty)

     2
    / \
   /   \
  1     3
 / \   / \
Fork 2 (Fork 1 Empty Empty) (Fork 3 Empty Empty)

    3
   / \
  1
 / \
    2
   / \
Fork 3 (Fork 1 Empty (Fork 2 Empty Empty)) Empty

      3
     / \
    2
   / \
  1
 / \
Fork 3 (Fork 2 (Fork 1 Empty Empty) Empty) Empty
```

Task: write a function `preOrderTree :: [a] -> [BT a]`, with the property that `elem t (preOrderTree xs)` if and only if `treePreOrder t = xs` for all `t :: BT a` and `xs :: [a]`.

Very hard task: write a function `breadthFirstTree :: [a] -> [BT a]`, with the property that `elem t (breadthFirstTree xs)` if and only if `treeBreadthFirst t = xs` for all `t :: BT a` and `xs :: [a]`. solution)