```
data Base = A | T | C | G
   deriving (Show)
```

We can encode this enumeration type by two bits, in which each bit is represented by the type `Either () ()`. The function `toBase` converts each constant to our encoding.

```
type BASE = (Either () (), Either () ())

toBase A = (Left (), Left ())
toBase T = (Left (), Right ())
toBase C = (Right (), Left ())
toBase G = (Right (), Right ())
```

The instance of class `Key` for type `Base` is defined similar to the instance for lists.

```
instance Key Base where
   data Map Base val = B (Map BASE val)

   empty              = B empty

   lookup k (B m)     = lookup (toBase k) m

   insert k f (B m) = B (insert (toBase k) f m)
```

An example call:

```
⋙ lookup [A,C,T] (insert [A,C,C] (const 1) (insert [A,C,T] (const 3) empty))
Just 3
```

Remark: the type of digital search trees, `Map key val`, takes two type arguments: `key` and `val`. These are treated quite differently: `Map` is defined by case analysis on the type of keys, but it is (fully) parametric (i.e. polymorphic) in the type of values. In other words, `Map key` is a functor. The system behind the case analysis can be seen more clearly if we write `Map K V` as an exponential i.e. V^K. The definition of `Map K V` is based on the *laws of exponents*:

$$V^{A+B} = V^A * V^B$$
$$V^{A*B} = (V^B)^A$$

A map for a sum type is a pair of maps, one for keys of the form `Left a` and one for elements of the form `Right b`. A map for a product type is given by a nested map: to look up `(a, b)` we first look up `a`, which yields a finite map, in which we look up `b`.

Exercise 12.4 (Programming, `Printf.hs`). Fancy some type hacking? This exercise discusses an alternative implementation of C's `printf` that is both type-safe and extensible. The basic idea is the same as for the implementation given in the lectures—the format directives are elements of singleton types so that the type of `printf` can *depend* on the type of the format directive—but the details differ. To illustrate, given these format directives

```
data D  =  D
data F  =  F
data S  =  S
```

```
infixr 4 &
(&) :: a → b → (a, b)
a & b  =  (a, b)
```

the type and class magic allows us to invoke `printf` with a variable number of arguments:

```
 ⋙ printf ("I am " & D & " years old.") 51
"I am 51 years old."
 ⋙ printf ("I am " & D & " " & S & " old.") 1 "year"
"I am 1 year old."
 ⋙ fmt = "Color " & S & ", Number " & D & ", Float " & F
 ⋙ :type fmt
fmt :: (String, (S, (String, (D, (String, F)))))
 ⋙ printf fmt "purple" 4711 3.1415
"Color purple, Number 4711, Float 3.1415"
```

Turning to the implementation, ideally we would like to define a type family and a type class,

```
type family Arg dir res :: *

class Format dir where
  printf :: dir → Arg dir String
```

with instances

```
type instance Arg D res =  Int → res

instance Format D where
  printf D = \ i → show i
...
```

Sadly, this approach only works for primitive directives such as `D` or `F`, but not for compound ones such as `D & F`. (Have a go. If you can make it work, I'd really like to know!)

One way out of this dilemma is to define `printf` in terms of a more complicated function.

```
printf :: (Format dir) ⇒ dir → Arg dir String
printf dir = format dir id ""

class Format dir where
  format :: dir → (String → a) → String → Arg dir a

instance Format D where
  format D cont out = \ i → cont (out ++ show i)
```

The helper function `format` takes two additional arguments in addition to the format directive: a so-called continuation and an accumulating string. The instance declaration shows how to use the arguments: the textual representation of `i` is appended to the accumulator `out`, the result of which is then passed to the continuation `cont`.

Fill in the missing bits and pieces i.e. define type instances and class instances for `F`, `S`, `String`, and `(dir1, dir2)`. *Hint:* do use the skeleton file provided as it includes pragmas to enable the necessary type and class system extensions.

13 Duality: folds and unfolds

Exercise 13.1 (Warm-up: programming, `FoldUnfold.hs`). In the lectures we have re-defined `foldr` in order to exhibit the duality between folds and unfolds more clearly.

```
type LIST a = Maybe (a,[a])

out :: [a] → LIST a
out []      = Nothing
out (x:xs)  = Just (x,xs)

inn :: LIST a → [a]
inn Nothing       = []
inn (Just (x,xs)) = x:xs

foldR   :: (Maybe (a,b) → b) → ([a] → b)
foldR al    = consume where
   consume = al ∘ fmap (fmap consume) ∘ out
```

But, are the two definitions, `foldr` and `foldR`, actually equivalent?

1. Define `foldr` in terms of `foldR`.

2. Conversely, define `foldR` in terms of `foldr`.

The type `Maybe(a,b)` was used to represent one layer of the list type. The `inn` and `out` functions are conversions between the *isomorphic* types `Maybe(a,[a])` and `[a]`. Suppose we would have represented a layer by the following data type

```
data LISTB elem res = NIL | CONS elem res
```

3. Show that `Maybe(a,b)` and `LISTB a b` are isomorphic too. You can do this by defining two conversion functions:

   ```
   mbP2LB :: Maybe (elem,res) → LISTB elem res

   lB2MbP :: LISTB elem res → Maybe(elem,res)
   ```

4. Also show that `LISTB a [a]` and `[a]` are isomorphic. The *witnesses* of this isomorphism have type

   ```
   outB :: [a] → LISTB a [a]
   innB :: LISTB a [a] → [a]
   ```

   Define these functions in terms of `inn`, `out`, `mbP2LB` and `lB2MbP`

5. Make `LISTB a` an instance of class `Functor`.

6. Consider the following definition for

   ```
   foldRB al    = consume where
      consume = al ∘ fmap consume ∘ outB
   ```

What is the type of `foldRB`. Define also the dual version `unfoldRB`

7. Now, take the following alternative mapping function for type `LISTB a b`

   ```
   emap :: (a → b) → LISTB a c → LISTB b c
   emap f NIL        = NIL
   emap f (CONS a b) = CONS (f a) b
   ```

   Now re-define the original `Prelude` function `map` in terms of `foldRB` and `emap`

8. Dually, re-define `map` in terms of `unfoldRB` and `emap`

Exercise 13.2 (Programming, `Minimax2.hs`). Reconsider the *multiway trees* of Exercise 9.4.

```
data Tree elem  =  Node elem [Tree elem]
```

1. Introduce a base functor for `Tree` and provide suitable class instances of `Functor` and `Base`.

2. Re-define the functions

   ```
   size, depth :: Tree elem → Integer
   ```

   using `fold`. The function `size` computes the size of a multiway tree i.e. the number elements of type `elem`; the function `depth` computes the length of the longest path from the root to a leaf.

3. Re-implement the function

   ```
   gametree :: (position → [position]) → (position → Tree position)
   ```

   that constructs a game tree in terms of `unfold`.

4. Re-implement the function

   ```
   winning  :: Tree position → Bool
   ```

   using `fold`. The function determines whether the root of a game tree is labelled with a winning position.

Exercise 13.3 (Worked example: algorithmic duality, `Sorting.hs`).

⟨...⟩ we can see that sorting is worth of serious study,
as a practical matter.
Even if sorting were almost useless, there would be plenty of rewarding reasons for studying it anyway! The ingenious algorithms that have been discovered show that sorting is an extremely interesting topic to explore in its own right. Many fascinating unsolved problems remain in this area, as well as quite a few solved ones.
From a broader perspective we will find also that sorting algorithms make a valuable case study *of how to attack computer programming problems in general.*
The Art of Computer Programming, Volume 3—Donald E. Knuth

We have used sorting as a running example to illustrate the idea of algorithmic duality. This exercise continues the journey looking at a family of sorting algorithms that are based on search trees, either implicitly or explicitly. These algorithms work in two phases:

- first phase: create a search tree from an unordered list;
- second phase: flatten the search tree to an ordered list.

In order to be able to apply the machinery of higher-order operators such as folds and unfolds, we assume the following definition of binary trees and its associated base functor.

```
data Tree elem      =  Empty | Node (Tree elem) elem (Tree elem)
data TREE elem tree =  EMPTY | NODE tree elem tree
```

During the lectures, the first phase has already been discussed. The (non-recursive) core algorithm that served as the basis for two dual variants of phase one, was defined as follows:

```
growCore :: (Ord a) ⇒ LIST a (x,TREE a x) → TREE a (Either x (LIST a x))
growCore NIL = EMPTY
growCore (CONS a (et, EMPTY)) = NODE (Left et) a (Left et)
growCore (CONS a (nt, NODE l b r))
  | a < b       =  NODE (Right (CONS a l)) b (Left r)
  | otherwise   =  NODE (Left l) b (Right (CONS a r))
```

The dual functions that create a search tree are:

```
grow1, grow2 :: (Ord elem) ⇒ [elem] → Tree elem
grow1  =  unfold  (para  (fmap (joinRight inn) ∘ growCore))
grow2  =  fold    (apo  (growCore ∘ fmap (splitRight out)))
```

1. Define functions that flatten a search tree.

   ```
   flatten1, flatten2 :: (Ord elem) ⇒ Tree elem → [elem]
   flatten1  =  fold    (apo   (flattenCore ∘ fmap (splitRight out)))
   flatten2  =  unfold  (para  (fmap (joinRight inn) ∘ flattenCore))
   ```

   Again, you only have to define the *algorithmic core* of these functions:

   ```
   flattenCore :: (Ord a) ⇒ TREE a (x, LIST a x) → LIST a (Either x (TREE a x))
   ```

   Be careful to preserve the relative order of elements so that a search tree is flattened to an ordered list (inorder traversal).

   Try to describe the workings of `flatten1` (which focusses on the input) and `flatten2` (which focusses on the output) in your own words.

2. Combine the phases to form sorting algorithms—there are four combinations altogether. Do you recognize any of the algorithms? One of them implements a version of Quick Sort, where the search tree structure is made explicit.

Exercise 13.4 (Pencil and paper: uniqueness and fusion). In the lectures we have introduced a *generic* definition of fold that works for arbitrary recursive datatypes.

```
fold :: (Base f) ⇒ (f a → a) → (Rec f → a)
fold a  =  a ∘ fmap (fold a) ∘ out
```

The generic definitions enjoys generic properties! First of all, folds enjoy the following *uniqueness property*:

$$\texttt{f = fold a} \iff \texttt{f} \circ \texttt{inn = a} \circ \texttt{fmap f} \tag{22}$$

The law states that `fold` is the unique solution of its defining equation.

1. Show that the uniqueness property implies the *computation law*:

   $$\texttt{fold a} \circ \texttt{inn} = \texttt{a} \circ \texttt{fmap (fold a)} \tag{23}$$

2. Show that the uniqueness property implies the *reflection law*:

   $$\texttt{id} = \texttt{fold inn} \tag{24}$$

   Note that `inn :: f (Ref f) → Rec f` is an algebra.

3. Finally, show that the uniqueness property implies the *fusion law*:

   $$\texttt{h} \circ \texttt{a = b} \circ \texttt{fmap h} \Longrightarrow \texttt{h} \circ \texttt{fold a = fold b} \tag{25}$$

   The fusion law states a condition for fusing a function with a fold to form another fold.

4. Use the properties above to show that `inn` and `fold (fmap inn)` are inverses of each other:

   $$\texttt{inn} \circ \texttt{fold (fmap inn)} = \texttt{id} \tag{26a}$$
   $$\texttt{fold (fmap inn)} \circ \texttt{inn} = \texttt{id} \tag{26b}$$

   In other words, `fold (fmap inn) = out`. *Hint:* first show (26a), then use (26a) to establish (26b).

5. *Optional:* can you dualize the laws to unfolds?

Modules

Haskell has a relatively simple module system which allows programmers to create and import modules, where a *module* is simply a collection of related types and functions.

Declaring modules

Most projects begin with something like the following as the first line of code:

```haskell
module Main
where
```

This declares that the current file defines functions to be held in the `Main` module. Apart from the `Main` module, it is recommended that you name your file to match the module name. So, for example, suppose you were defining a number of protocols to handle various mailing protocols, such as POP3 or IMAP. It would be sensible to hold these in separate modules, perhaps named `Network.Mail.POP3` and `Network.Mail.IMAP`, which would be held in separate files. Thus, the POP3 module would have the following line near the top of its source file.

```haskell
module Network.Mail.POP3
where
```

This module would normally be held in a file named

`src/Network/Mail/POP3.hs` .

Note that while modules may form a hierarchy, this is a relatively loose notion, and imposes nothing on the structure of your code.

By default, all of the types and functions defined in a module are exported. However, you might want certain types or functions to remain private to the module itself, and remain inaccessible to the outside world. To achieve this, the module system allows you to explicitly declare which functions are to be exported: everything else remains private. So, for example, if you had defined the type `POP3` and functions `send :: POP3 → IO ()` and `receive :: IO POP3` within your module, then these could be exported explicitly by listing them in the module declaration:

```haskell
module Network.Mail.POP3 ( POP3 (..) , send , receive )
```

Note that for the type `POP3` we have written `POP3 (..)`. This declares that not only do we want to export the *type* called `POP3`, but we also want to export all of its constructors too.

Importing modules

The `Prelude` is a module which is always implicitly imported, since it contains the definitions of all kinds of useful functions such as `map :: (a → b) → (f a → f b)`. Thus, all of its functions are in scope by default. To use the types and functions found in other modules, they must be imported explicitly. One useful module is the `Data.Maybe` module, which contains useful utility functions:

```haskell
maybe      :: b → (a → b) → Maybe a → b
catMaybes  :: [Maybe a] → [a]
```

Importing all of the functions from `Data.Maybe` into a particular module is done by adding the following line below the module declaration, which imports every entity exported by `Data.Maybe`

```
import Data.Maybe
```

It is generally accepted as good style to restrict the imports to only those you intend to use: this makes it easier for others to understand where some of the unusual definitions you might be importing come from. To do this, simply list the imports explicitly, and only those types and functions will be imported:

```
import Data.Maybe ( maybe, catMaybes )
```

This imports `maybe` and `catMaybes` in addition to any other imports expressed in other lines.

Qualifying and hiding imports

Sometimes, importing modules might result in conflicts with functions that have already been defined. For example, one useful module is `Data.Map`. The base datatype that is provided is `Map` which efficiently stores values indexed by some key. There are a number of other useful functions defined in this module:

```
empty   ::  Map k v
insert  ::  (Ord k) ⇒ k → v → Map k v → Map k v
update  ::  (Ord k) ⇒ k → Map k v → Maybe v
```

It might be tempting to import `Map` and these auxiliary functions as follows:

```
import Data.Map ( Map (..), empty, insert, lookup )
```

However, there is a catch here! The `lookup` function is initially always implicitly in scope, since the `Prelude` defines its own version. There are a number of ways to resolve this. Perhaps the most common solution is to qualify the import, which means that the use of imports from `Data.Map` must be prefixed by the module name. Thus, we would write the following instead as the import statement:

```
import qualified Data.Map
```

To actually use the functions and types from `Data.Map`, this prefix would have to be written explicitly. For example, to use `lookup`, we would actually have to write `Data.Map.lookup` instead.

These long names can become somewhat tedious to use, and so the qualified import is usually given as something different:

```
import qualified Data.Map as M
```

This brings all of the functionality of `Data.Map` to be used by prefixing with `M` rather than `Data.Map`, thus allowing you to use `M.lookup` instead.

Another solution to module clashes is to hide the functions that are already in scope within the module by using the `hiding` keyword:

```
import Prelude hiding ( lookup )
```

This will override the `Prelude` import so that the definition of `lookup` is excluded.