The Barcode Reader
Informatics 1 – Functional Programming: Tutorial 6

Due: The tutorial of week 8 (12/13 Nov.)

Please attempt the entire worksheet in advance of the tutorial, and bring with you all work, including (if a computer is involved) printouts of code and test results. Tutorials cannot function properly unless you do the work in advance.

You may work with others, but you must understand the work; you can't phone a friend during the exam.

Assessment is formative, meaning that marks from coursework do not contribute to the final mark. But coursework is not optional. If you do not do the coursework you are unlikely to pass the exams.

Attendance at tutorials is obligatory; please let your tutor know if you cannot attend.

The barcode reader

In this tutorial we will take a look at a barcode scanner. Of course, we will not be doing the actual scanning, but what we *will* do is search a database for the item that belongs to a scanned barcode. We will read the database from a file and store it in different shapes, to see which gives the fastest retrieval times.

The Haskell files that come with this tutorial are `tutorial6.hs`, `KeymapList.hs`, and `KeymapTree.hs`. There is also the database itself: `database.csv` (`csv` stands for 'comma-separated values').

Let's start by opening `KeymapList.hs`. This file defines an abstract data type for a keymap. The file starts as follows:

```
module KeymapList ( Keymap,
                    size,
                    ...
                  )
where
```

This declaration means that `KeymapList.hs` is a *module* that can be used by other Haskell files, just like `Data.Char` and `Test.QuickCheck`. The functions and constructors mentioned in parentheses (`Keymap`, `size`, etc.) are the ones that are *exported* by the module, i.e. the ones that can be used when the module is imported somewhere else.

Next, let's take a look at the type declaration for `Keymap`.

```
newtype Keymap k a = K [(k,a)]
```

This defines the polymorphic data type `Keymap`. The first argument, `k`, is what's used as the *key* in the keymap, the second (`a`) is the type of the *values* stored. For instance, a keymap of type `Keymap Int String` associates keys of type `Int` with values of type `String`.

Finally, there is the definition itself, `K [(k,a)]`. As you see, a keymap is simply stored as a list of key–value pairs. The type-constructor `K` is just a wrapper that prevents us from using normal list functions on keymaps. This is precisely the idea: we want the type to be *abstract*, so we can hide the fact that we are using lists underneath. This ensures that we don't use ordinary list functions on keymaps and accidentally invalidate the representation. And it frees us to change our representation whilst guaranteeing that our users' code still works.

Now, let's look at the functions in the file. The *constraints* `Eq k` mean that whatever the type `k`, it must support *equality testing*. In other words, if we want to use a type as a key, we have to be able to use the function `(==)` on values of that type—otherwise, we would have no way to identify the right key in the keymap.

- `size :: Eq k => Keymap k a -> Int`

  This function gives the number of entries in a `Keymap`.

- `get :: Eq k => k -> Keymap k a -> Maybe a`

  Given a key, this function looks up the associated value in a `Keymap`. In this function keys are matched using `(==)`, which is why the constraint `Eq k` is needed. The value returned is not just of type `a`, but `Maybe a`, because a key might not occur in a `Keymap`. We will get back to this later.

- `set :: Eq k => k -> a -> Keymap k a -> Keymap k a`

  Given a key and a value, this function sets the value associated with the key to the given value in a `Keymap`. If the key already had a value, this value is replaced; otherwise the key is newly added to the keymap.

- `del :: Eq k => k -> Keymap k a -> Keymap k a`

  This function deletes an entry from a keymap.

- `select :: Eq k => (a -> Bool) -> Keymap k a -> Keymap k a`

  This function narrows a keymap down to those values that satisfy a given predicate.

- `tolist :: Eq k => Keymap k a -> [(k,a)]`

  This function exports the keymap as a list.

- `fromList :: Eq k => [(k,a)] -> Keymap k a`

  This function builds a keymap from a list.

Now that we know what `KeymapList` is like, we can start working on `tutorial6.hs`. Just below the top, you will find the declarations:

```
import KeymapList

type Barcode = String
type Product = String
type Unit    = String

type Item    = (Product,Unit)

type Catalogue = Keymap Barcode Item
```

Firstly, we are importing the `KeymapList` module. Next, there are type aliases `Barcode`, `Product` and `Unit`, whose values are strings, and `Item` which is a pair `(Product,Unit)`. Finally, we are using the type alias `Catalogue` whose values are keymaps that associate a `Barcodes` with an `Item`.

Below that, you will find a little test database.

Exercises

1. Before we can work on the database, we need some way of viewing it. If you try `testDB` or `show testDB` on the GHCi prompt, you will find it refuses to print. Instead, try:

   ```
   *Main> toList testDB
   ```

   However, this looks rather cluttered.

   (a) Write a function `longestProductLen` that finds the length of the longest product name in a list of `(Barcode,Item)`-pairs.

   (b) Write a function `formatLine` that, given a number (the desired length of the product name) prints the barcode, product and unit information, separated by dots, as a single line. For example:

   ```
   *Main> formatLine 7 ("0001",("product","unit"))
   "0001...product...unit"
   *Main> formatLine 7 ("0002",("thing","unknown"))
   "0002...thing.....unknown"
   ```

   You may assume that the product name is never longer than the desired length for it.

   (c) Write a function `showCatalogue` that pretty-prints a `Catalogue`. You will need to use `toList` (from the `KeymapList` module). Test your function by writing at the prompt:

   ```
   *Main> putStr (showCatalogue testDB)
   ```

2. Next, we will start using the `get` function.

   Firstly, try the following at the interpreter:

   ```
   *Main> :t get
   ```

   You will see that the result type is `Maybe a`. This data-type is defined as follows:

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

   A value of type `Maybe a` is either the value `Nothing`, or it is a value of the form `Just x`. In this way, values of type `Maybe a` can be used to represent ordinary values that have the option of *failing*. The type is just like the type `a`, but it contains the extra value `Nothing` meaning "there is no value."

   So by saying that `get` returns a value of type `Maybe a`, we are saying that it will either return a value of the form `Just x` if `x` appears in the database, or otherwise will return `Nothing`.

   Now try the following expressions at the interpreter:

   ```
   *Main> get "9780201342758" testDB

   *Main> get "000" testDB
   ```

   What did you get?

   (a) When you apply your function `get` with a certain key to the test database, what is the type it returns? What are the possible values (hint: there are five)?

   (b) Write a function `maybeToList :: Maybe a -> [a]` that returns the empty list if the input is `Nothing`, and returns a singleton list otherwise.

   (c) Write another function `listToMaybe :: [a] -> Maybe a`. You can think of this function as a safe version of `head`. It should return `Nothing` if the input is the empty list, and return just the head otherwise.

(d) Write the function `catMaybes :: [Maybe a] -> [a]`. This function should drop all elements of the form `Nothing`, and strip the `Just` constructor from all the other elements.

3. Using the functions from the previous exercise, Write a function `getItems` that, given a list of `Barcode`s, returns a list of `Item`s. Test your code for errors by typing:

   ```
   *Main> getItems ["0001","9780201342758","0003"] testDB
   ```

   It should return a list containing only the item for the textbook.

The real database

Your file `tutorial6.hs` contains a few functions that we haven't shown yet. First of all, it can read in the database file `database.csv`. You can do this by writing:

```
*Main> theDB <- readDB
```

(it might take a little while). The database is now loaded and assigned to the variable `theDB`, and will remain in the computer's memory until you reload your file.

The database is pretty large, so it's not a good idea to try to print it on the screen. But you can ask for the size:

```
*Main> size theDB
```

Another thing that is provided is the function `getSample`. This will give you a random barcode from the database; try:

```
*Main> getSample theDB
```

To find the `Item` for that barcode, follow the previous command with:

```
*Main> get it theDB
```

('`it`' is a GHCi-special that refers to the value returned by the last expression it evaluated).

We will see how fast our implementation of keymaps in `KeymapList` is. First, we need to turn on the timekeeping feature of GHCi. Type this at the prompt:

```
*Main> :set +s
```

It may seem as if nothing has changed, but GHCi will now give you time (and memory use) information for each expression you ask it to evaluate.

Exercises

4. (a) Reload your file and load up the database again. Take a note of how much time it took.
   (b) Take at least ten samples from the database, and record how much time it takes to find an item with `get`. (The time it takes to find a random sample is not really relevant here.)
   (c) Think about the different values you get. If the database was 2 times bigger, how much longer would it take (on average) to find an item? How many items does the `get` function from `KeymapList` look at before it finds the right item, if it happens to be the last one?

Keymaps as trees

In this part of the exercise we will build a different implementation of keymaps, based on trees rather than lists. In the file `KeymapTree.hs` you will find a different declaration of the `Keymap` data type, as well as the skeletons of the functions as in `KeymapList.hs`.

In `KeymapTree` we will implement the same functions and data type as we had in `KeymapList`, so from the outside they will look the same. However, internally they will be very different, and so will their performance.

The idea behind the tree implementation is explained in section 16.7 (page 395) of Thompson or pages 135-137 of Lipovaca. Basically, the data is stored in the nodes of a tree. The left branch of a node only stores data that is *smaller* than the data at the node itself, while the right branch stores data that is *larger*.

First, look at the data type for `Keymap`. It is a lot more complicated than before:

```
data Ord k => Keymap k a = Leaf
                         | Node k a (Keymap k a) (Keymap k a)
```

The data type again defines a keymap storing keys of type `k` and values of type `a`. To sort the keys into larger and smaller ones, we need the constraint `Ord k`. This means that the keys used in a keymap can be *ordered*; in practice, the means we can always use the functions `(==)`, `(>=)` and `(<=)` on keys.

The constructors for `Keymaps` then are `Leaf` and `Node`. The value of `Leaf` is just a leaf. It stores no data, like the empty list `[]`. The second one does all the work. It carries (in order):

- a key of type `k`
- the associated value of type `a`
- a subtree on the left, which is a tree of type `Keymap k a`
- a subtree on the right, also a tree of type `Keymap k a`

When building a keymap in the shape of a tree, we want to make sure that the tree remains sorted. That is, for any node with a certain key, the keys in the left subtree should all be smaller than that key, and the keys in the right subtree should all be larger. To ensure this, we make sure a user of these keymaps can only access them through functions that are safe.

Exercises

5. In `tutorial6.hs` change the line:

   ```
   import KeymapList
   ```

   to

   ```
   import KeymapTree
   ```

   and load `tutorial6.hs` up in GHCi. Think of what the following expression should return:

   ```
   size ( Node "0001" "just some item" Leaf Leaf )
   ```

   If you try it out, what does it say?

What happens is that by not exporting the constructors `Node` and `Leaf` themselves, we prevent people from writing recursive functions on our trees—at least outside of `KeymapTree.hs`. Now, we will complete the functions in `KeymapTree.hs`.

Exercises

6. (a) Look at the function `size`. How does it work, and how can we recurse over trees?
   (b) Define the function `depth`, which takes a tree and returns the *maximal* depth of the tree, i.e. the length of the longest path from its root to any of its leaves. A leaf should have depth 0.
   (c) Load up `KeymapTree.hs` into GHCi and try the functions `size` and `depth` on the little test tree `testTree` (it should have size 4 and depth 3).
7. Define the function `toList`, which takes a tree and returns the corresponding list of pairs. Try it on `testTree`. Can you make it so that the returned list is sorted?

8. Take a look at the function `set`. The function defines a helper function `f` to do the recursion, to avoid repeating the variables `key` and `value` too often in the definition.

   (a) Explain what the function `f` does when it encounters a leaf.

   (b) Explain what the function `f` does when it looks at a node and it encounters the key it was looking for? Complete the definition of this function. Hint: the last two cases will need to recurse down an appropriate branch.

9. Complete the function `get`. Remember that you should return a `Maybe`-value. When should it return `Nothing`, and when should it return `Just` a value? Test your function on `testTree` first, and then use QuickCheck to verify `prop_set_get`.

10. Write a function `fromList` to turn a list into a tree. You should use the function `set` to add each element to the tree. Think about what the tree is that you should start out with. For this question you can use recursion over the input list, but you could also try to use `foldr` and `uncurry`.

    Use the test property `prop_toList_fromList` to test your solutions. If you managed to return sorted lists with `toList`, you can also test with `prop_toList_fromList_sorted`.

At this point we have added enough functions to `KeymapTree` to start evaluating its performance.

Exercises

11. Save `KeymapTree.hs` and open up `tutorial6.hs`

    (a) Load up the database again by entering `theDB <- readDB` at the prompt. This time, it will be constructed as a tree. How much time did it take?

    (b) Try on at least 10 examples how fast your `get` function is now. Remember:

    ```
    *Main> getSample theDB
    ```

    gives you a random barcode, and then

    ```
    *Main> get it theDB
    ```

    looks up the associated item.

    (c) How many barcodes does our `get` function inspect, at most, when searching the database?

Optional material

Exercises

12. Define two functions `filterLT` and `filterGT`, such that `filterLT k t` is the tree that results from removing all elements in `t` whose key is greater than `k`. For example:

    ```
    *Main> filterLT "0900000000000" testDB
    [("0042400212509",("Universal deep-frying pan","pc"))
    ,("0265090316581",("The Macannihav'nmor Highland Single Malt","75ml bottle"))

    *Main> filterLT "0" testDB
    []
    ```

    The function `filterGT k t` is the tree that results from removing all elements in `t` whose key is less than `k`. So:

    ```
    *Main> filterGT "0900000000000" testDB
    [("0903900739533",("Bagpipes of Glory","6-CD Box"))
    ,("9780201342758",
        ("Thompson - \"Haskell: The Craft of Functional Programming\"","Book"))]

    *Main> filterGT "0" testDB
    ```

```
[("0042400212509",("Universal deep-frying pan","pc"))
,("0265090316581",("The Macannihav'nmor Highland Single Malt","75ml bottle"))
,("0903900739533",("Bagpipes of Glory","6-CD Box"))
,("9780201342758",
    ("Thompson - \"Haskell: The Craft of Functional Programming\"","Book"))]
```

We will need these functions for the next part of the exercise.

13. Define the function `merge`, which takes two trees and produces a single tree of all their elements. Write a suitable `quickCheck` test for your function.
14. Define the function `del`, which takes a key and a tree, and returns a tree identical to its argument except with any entry for the given key deleted. You will need the function `merge` here.
15. Define the function `select`, which takes a predicate and a tree, and returns a new tree that contains only entries where the value satisfies the predicate.