PROPLISTS

A proplist is a list of key value pairs. Although a proplist is a list, the order of the key value pairs is irrelevant in these exercises: a proplist is treated as a map. Usually, a record would be more suitable in Haskell but the proplist enables a more functional “state of mind”. In other languages, like Erlang for instance, proplists are very common.

The files needed and start of the exercises including solutions can be found at:

https://github.com/tty/joy-of-coding

Given the following type definition and test instances of a proplist:

```
module Proplists where

import Data.List
import Data.Char

type Proplist = [ (String,PropVal) ]

data PropVal = Int Integer
             | Str String
             | Undefined
             | Obj Proplist
             | Arr [PropVal]
             deriving (Show, Eq)

testList :: Proplist
testList = [("a",Int 1), ("b", Str "Foo")]

testList2 :: Proplist
testList2 = [("b", Str "Bar"),("c", Undefined)]

fromArr a = Arr a
fromStr s = Str s
fromObj o = Obj o
fromInt i = Int i
```

1. Deletion

Write a function named `del` that removes a key (if present) from a proplist:

```
> del "b" testList
[("a",Int 1)]

> del "c" testList
[("a",Int 1),("b",Str "Foo")]
```

2. Setter

Create a function `set` that sets or replaces a given key of a proplist:

```
> set "a" (Str "Bar") testList
[("a",Str "Bar"),("b",Str "Foo")]

> set "c" (Str "Bar") testList
[("c",Str "Bar"),("a",Int 1),("b",Str "Foo")]
```

3. Getter

Create a function `get` that, given a key and a proplist, retrieves the value from the proplist, or `Undefined` otherwise:

```
> get "a" testList
Int 1

> get "c" testList2
Undefined

> get "z" testList
Undefined
```

4. Merge proplists

Now a bit more challenging, create a function `merge x y` that merges two proplists x y by replacing or adding all values from y into x:

```
> merge testList testList2
[("c",Undefined),("b",Str "Bar"),("a",Int 1)]
```

or

```
[("b",Str "Bar"),("a",Int 1),("c",Undefined)]
```

Remember, order is irrelevant.

JSON

5. toJSON

Extend “Proplists.hs” with a function `toJSON` that converts a `PropVal` to a `String`, where a Proplist is encoded as a JSON object.

```
> putStr $ toJSON (Str "test")
"test"

> putStr $ toJSON (Obj testList)
{"a":1,"b":"Foo"}

> putStr $ toJSON (Arr [Int 3, Str "test"])
[3,"test"]
```

Bonus: pretty print the output with tabs/spaces/newlines.

Parsing JSON is a bit harder and out of scope for these exercises, therefore we have included a JSON parser, written with Parsec which we will use in the next exercises. The module can be found in “JSON.hs”. Parsec is a very elegant parser, take a look if you like and try to understand.

Note that this parser is not yet fully JSON compliant, it does not parse Bools nor Floats.

The important and only exported function from module `JSON` is `parseJSON`. It takes a string and parses it to a `PropVal`. The toplevel can only contain be an object or an array.

DATABASE

Given the following functions and data definitions in “Db.hs”, this should compile if the previous exercises are completed successfully:

```
module Db where

import JSON
import Proplists
import Data.List

type DB = [ Proplist ]
type Record = Proplist

data Mp3 = Mp3 { song :: String, artist :: String, rating :: Integer }
deriving (Show)

mp31 = Mp3 { song = "Street Spirit", artist = "Radiohead", rating = 9}
mp32 = Mp3 { song = "We Will Rock You", artist = "Queen", rating = 3}
mp33 = Mp3 { song = "Bohemian Rhapsody", artist = "Queen", rating = 4}

testDb = addRecord (mp3ToProplist mp31) create
testDb2 = addRecord (mp3ToProplist mp32) testDb
testDb3 = addRecord (mp3ToProplist mp33) testDb2

makeMp3 s a r = mp3
  where
    s' = set "song" (fromStr s) []
    a' = set "artist" (fromStr a) s'
    mp3 = set "rating" (fromInt r) a'

mp3ToProplist (Mp3 {song = s, artist = a, rating = r}) = makeMp3 s a r

create :: DB
create = []

addRecord :: Record -> DB -> DB
addRecord r db = r:db
```

6. Import JSON module

Create a new file “Db.hs”, import `JSON` and `Proplists modules`. Test that you can convert the above database to JSON.

```
> dbToJSON testDb3
```

Check that the parser works:

```
> fromJSON (dbToJSON testDb3)
```

7. Search

Write a function `search` that searches through a database using a given function. The type of this function is:

```
search :: (Record -> Bool) -> DB -> D

> search (\x -> (get "artist" x) == (fromStr "Queen")) testDb3
```

```
[[("rating",Int 3),("artist",Str "Queen"),("song",Str "Bohemian Rhapsody")],
[("rating",Int 4),("artist",Str "Queen"),("song",Str "We Will Rock You")]]
```

This anonymous function used here, can be seen as the `WHERE` clause of a select statement.

8. Search functions

Now we can write selector functions that can be used instead of the anonymous function. Create a function that can be used to search for all records that have a rating higher than 4.

```
> search ratingHigherThan4 testDb3
[[("rating",Int 9),("artist",Str "Radiohead"),("song",Str "Street Spirit")]]
```

9. Select

Write a function `select` that selects all Records of a DB based on a given `Propval`. The type of this function is:

```
select :: (Record -> PropVal) -> PropVal -> DB -> DB

> select (get "artist") (fromStr "Queen") testDb3

[[("rating",Int 3),("artist",Str "Queen"),("song",Str "Bohemian Rhapsody")],
[("rating",Int 3),("artist",Str "Queen"),("song",Str "We Will Rock You")]]
```

Now we can create selector functions using currying:

```
artistSelector artist = select artistEq (fromStr artist)

artistEq = (get "artist")

> artistSelector "Queen" testDb3

[[("rating",Int 3),("artist",Str "Queen"),("song",Str "Bohemian Rhapsody")],
 [("rating",Int 3),("artist",Str "Queen"),("song",Str "We Will Rock You")]]
```

10. Update

Write a function `update` that is similar to `select`, but with an extra argument, a `Record`. All rows matching the selection are updated using the given record. You can use the `merge` function for this.

The type of this function is

```
update :: (Record -> PropVal) -> PropVal -> Record -> DB -> DB

> update (get "artist") (fromStr "Queen") [("rating", Int 10000)] testDb3

[[("rating",Int 10000),("artist",Str "Queen"),("song",Str "Bohemian Rhapsody")],
 [("rating",Int 10000),("artist",Str "Queen"),("song",Str "We Will Rock You")],
 [("rating",Int 9),("artist",Str "Radiohead"),("song",Str "Street Spirit")]]
```

11. Beers!!

If you pulled the exercise files from GitHub you probably noticed the function readDB which uses IO to read a JSON database of beers. It takes one argument: a function to apply to the read database.

Give it a try:

```
readDB (\x -> printDB (select (get "name") (Str "Innovation") x))
```

Write a (selector) function that finds the beer with the highest alcohol by volume (abv).