Worksheet Ten - Functional Programming Redux

The next instalment of the functional programming exercises...

This set of exercises start off slowly but soon "ramp up" to use many of the latter features we have discussed on `C#` and FP.
The exercises marked (`**`) and (`***`) are considered "advanced" and are for "extra-credit" (and exam revision) purposes.

Some of the questions make use of the `language-ext` library which is described on the external website. You will need to use this library for the `Option`, `Bind`, ... types.

Part A

1. Write a console app that calculates a user's Body-Mass Index:
   - prompt the user for her height in metres and weight in kg
   - calculate the BMI as `weight/height^2`
   - output a message: `underweight` (`bmi < 18.5`), `overweight` (`bmi >= 25`) or `healthy weight`
   - Structure your code so that structure it so that *pure* and *impure* parts are separate
   - Unit test the pure parts
   - Unit test the impure parts using the HOF-based approach.
2. Write a generic function that takes a string and parses it as a value of an enum.
   It should be usable as follows:

   ```
   Enum.Parse<DayOfWeek>("Friday") // => Some(DayOfWeek.Friday)
   Enum.Parse<DayOfWeek>("Freeday") // => None
   ```

3. Write a `Lookup` function that will take an `IEnumerable` and a predicate, and return the first element in the `IEnumerable` that matches the predicate, or `None` if no matching element is found. Write its signature in arrow notation:

   ```
   bool isOdd(int i) => i % 2 == 1;
   new List<int>().Lookup(isOdd) // => None
   new List<int> { 1 }.Lookup(isOdd) // => Some(1)
   ```

4. Consider the extension methods defined on `IEnumerable` in `System.LINQ.Enumerable`.
   Which ones could potentially return nothing, or throw some kind of not-found exception, and would therefore be good candidates for returning an `Option<T>` instead?

5. Write implementations for the methods in the `AppConfig` class below. (For both methods, a reasonable one-line method body is possible).

   - You may assume that the settings are of type `string`, `numeric`, or `date`.
   - Can this implementation help you to test code that relies on settings in a `.config` file? Explain your answer.

```
public class AppConfig
{
    NameValueCollection source;

    public AppConfig() : this(ConfigurationManager.AppSettings)
{ }

    public AppConfig(NameValueCollection source)
    {
        this.source = source;
    }

    public Option<T> Get<T>(string name)
    {
        throw new NotImplementedException("your implementation
here...");
    }

    public T Get<T>(string name, T defaultValue)
    {
        throw new NotImplementedException("your implementation
here...");
    }
}
```

6. Implement `Map` (the function) for `ISet<T>` and `IDictionary<K, T>`.

7. Implement `Map` for an `Option` type and `IEnumerable` in terms of `Bind` and `Return`.
   Note: You may have to write the `Option` type if you do not use the

`language-ext` library.

8. Use `Bind` and an `Option`-returning `Lookup` function to implement the `GetWorkPermit` function shown below.
Then augment your implementation so that the function returns `None` if the work permit has expired.

```
static Option<WorkPermit> GetWorkPermit(Dictionary<string,
Employee> people, string employeeId)
{
    throw new NotImplementedException();
}
```

9. Use `Bind` to implement `AverageYearsWorkedAtTheCompany`, shown below (only employees who have left the company should be included):

```
static double AverageYearsWorkedAtTheCompany(List<Employee>
employees)
{
    // your implementation here...
}

//...

public struct WorkPermit
{
    public string Number { get; set; }
    public DateTime Expiry { get; set; }
}

public class Employee
{
    public string Id { get; set; }
    public Option<WorkPermit> WorkPermit { get; set; }

    public DateTime JoinedOn { get; }
    public Option<DateTime> LeftOn { get; }
}
```

Part B

We now consider further functional operations on lists.

Implement functions to work with the singly linked List from the examples repo:

1. `InsertAt` inserts an item at the given index.
2. `RemoveAt` removes the item at the given index.
3. `TakeWhile` takes a predicate, and traverses the list yielding all items until it find one that fails the predicate.
4. `DropWhile` works similarly, but excludes all items at the front of the list.
5. Now write implementations that take an `IEnumerable` rather than a List.

Now we consider tree structures.

1. Is it possible to define `Bind` for the binary tree implementation in the examples repo. If so, implement `Bind`, otherwise explain why it is not possible
2. Implement a `LabelTree` type, where each node has a label of type string and a list of subtrees; this could be used to model a typical navigation tree or a category tree in a website.
3. (**) Imagine you need to add localisation to your navigation tree and you are given:

    - a `LabelTree` where the value of each label is a key, and
    - a dictionary that maps keys to translations in one of the languages that your site must support.

    (Hint: define `Map` for `LabelTree` and use it to obtain the localised navigation/category tree.)

Part C (**)

1. Without looking at any code or documentation (or using "intellisense" - difficult I know), write the function signatures of

    - `OrderByDescending`,
    - `Take`, and
    - `Average`,

    which are used to implement `AverageEarningsOfRichestQuartile`:

    ```
    static decimal AverageEarningsOfRichestQuartile(List<Person> population)
        => population
            .OrderByDescending(p => p.Earnings)
            .Take(population.Count/4)
            .Select(p => p.Earnings)
            .Average();
    ```

Check your answer with the `MSDN` documentation; how is the signature of `Average` different to the other two methods?

2. Implement a general purpose `Compose` function that takes two unary functions and returns the composition of the two.

3. Write a `ToOption` extension method to convert an `Either` into an `Option`. Then write a `ToEither` method to convert an `Option` into an `Either`, with a suitable parameter that can be invoked to obtain the appropriate `Left` value, if the `Option` is `None`.

4. Consider the workflow where two or more functions that return an `Option` are chained using `Bind`.

   - Then change the first one of the functions to return an `Either`.
   - This should cause compilation to fail.

   Since `Either` can be converted into an `Option` write extension overloads for `Bind`, so that functions returning `Either` and `Option` can be chained with `Bind`, yielding an `Option`.

5. Write a function `Safely` of type

   ((() → R), (Exception → L)) → Either<L, R>

   that will execute the given function in a `try/catch` (from `language-ext`), returning an appropriately populated `Either`.

6. Write a function `Try` of type

   (() → T) → Exceptional<T>

   that will execute the given function in a `try/catch` block (from `language-ext`, returning an appropriately populated exception.

Part D (***)

1. These questions examine *partial application* of functions using a binary arithmetic function (you should refer to the partial function application examples from the appropriate lecture):

   1. Write a function `Remainder`, that calculates the remainder of integer division(and works for negative input values!).
      Note how the expected order of parameters is not the one that is most likely to be required by partial application (you are more likely to partially apply the divisor).
   2. Write an `ApplyR` function, that provides the rightmost parameter to

a given binary function (try to write it without looking at the implementation for `Apply`).

3. Write the signature of `ApplyR` in arrow notation, both in *curried* and *non-curried* form.
4. Use `ApplyR` to create a function that returns the remainder of dividing any number by 5.
5. Write an overload of `ApplyR` that gives the rightmost argument to a ternary function.

2. We now move onto ternary functions.

    1. Define a class `PhoneNumber` with three fields:

    ```
    number type(home, mobile, ...)
    country code('it', 'uk', ...)
    number
    ```

    `CountryCode` should be a custom type with implicit conversion *to* and *from* string.

    2. Now define a ternary function that creates a new number, given values for these fields.
    What is the signature of your factory function?

3. Functions are everywhere.

    You may still feel that objects are ultimately more powerful than functions.
    To see that this is not necessarily so, challenge yourself to write a very simple logging mechanism without defining any classes or structs.
    A *logger* object should expose methods for related operations such as `Debug`, `Info`, `Error`?

    ```
    static void ConsumeLog(Log log) => log.Info("look! no
    objects!");

    enum Level { Debug, Info, Error }
    ```

Credits

These exercises are based upon some examples and questions from the book Functional Programming in C# published by Manning.