T-501-FMAL Programming languages, Practice class 3
Spring 2021

1. We considered the following F# datatype of node-labelled trees.

```
type 'a tree =
    | Lf
    | Br of 'a * 'a tree * 'a tree
```

Here is a variation, a datatype of nonempty trees, that excludes (on the level of typing) the empty tree `Lf`, which does not contain any label.

```
type 'a netree =
    | N of 'a * ('a netree) option * ('a netree) option
```

E.g., the tree
`Br (34, Br (23, Lf, Br (78, Lf, Lf)), Br (54, Lf, Lf))`
can be represented as a nonempty tree by
`N (34, Some (N (23, None, Some (N (78, None, None)))), Some (N (54, None, None)))`

Code a function `tree2netree : 'a tree -> ('a netree) option` that converts a tree to a nonempty tree (returning `None` in the "undefined" case of the empty tree).

Code a function `netree2tree: 'a netree -> 'a tree` that converts a nonempty tree to a tree.

2. Discussing F# record types, we considered the following type of persons.

```
type person =
   { name : string
   ; father : person option
   ; mother : person option
   }
```

Persons like this are really just a different representation of nonempty trees of strings. Provide evidence for this claim by coding functions `person2netree : person -> string netree` and `netree2person : string netree -> person` for conversion between persons and nonempty trees of strings.

3. Code a function `truncate : int -> 'a tree -> 'a tree` that truncates a tree at a given depth.

```
> truncate 0 (Br (34, Br (23, Lf, Br (78, Lf, Lf)), Br (54, Lf, Lf)));;
val it : int tree = Lf
> truncate 2 (Br (34, Br (23, Lf, Br (78, Lf, Lf)), Br (54, Lf, Lf)));;
val it : int tree = truncate 2 (Br (34, Br (23, Lf, Lf), Br (54, Lf, Lf)))
```

Hint: `truncate` is easily coded by direct recursion and is similar to the function `take` for lists.

4. Code a function `prettyprint : 'a tree -> unit` that prints a tree on the screen using indentation (similar to structured code).

```
> prettyprint (Br (34, Br (23, Lf, Br (78, Lf, Lf)), Br (54, Lf, Lf)));;
34
   23
      .
      78
         .
         .
   54
      .
      .
val it : unit = ()
```

Use the F# function `printf` that takes as arguments a format string and then the values to be printed. The newline symbol is `'\n'`.

The function `prettyprint` is naturally coded by direct recursion, don't try anything more complicated than so.

5. Code a function `breadthfirst : 'a tree -> 'a list` lists the labels of a tree in the breadth-first order.

```
> breadthfirst (Br (34, Br (23, Lf, Br (78, Lf, Lf)), Br (54, Lf, Lf)));;
val it : int list = [34; 23; 54; 78]
```

This problem is a little harder. It may be a good idea to first convert the tree into a list of list of labels where the labels in each layer of the tree are kept in a separate inner list. The final result can then be obtained by "flattening" this list of lists.

6. Before decimalization in 1971, the British pound (£) was 20 shillings (s), a shilling was 12 pence (d) and a penny was 4 farthings.

Amounts in this currency can be represented as an F# datatype

```
type oldCurrency =
    | Lsdf of (int * int * int * int)
```

where the four arguments of the data constructor `Lsdf` correspond to pounds, shillings, pence and farthings.

Code a function `normalize : oldCurrency -> oldCurrency` that normalizes a given amount so that the shilling, pence and farthing amounts lie in the intervals 0..19, 0..11 and 0..3 respectively.

```
> normalize (Lsdf (0, 24, 27, 5));;
val it : oldCurrency = Lsdf (1, 6, 4, 1)
```

Code a function `(+++) : oldCurrency -> oldCurrency -> oldCurrency` that adds two amounts and also normalizes the result.

There were 10 types of coin in circulation.

```
type oldCoin =
    | Farthing
    | Halfpenny
    | Penny
    | Threepence
    | Sixpence
    | Shilling
    | Florin
    | HalfCrown
    | Crown
    | DoubleFlorin

let value c =
    match c with
    | Farthing      -> Lsdf (0, 0, 0, 1)
    | Halfpenny     -> Lsdf (0, 0, 0, 2)
    | Penny         -> Lsdf (0, 0, 1, 0)
    | Threepence    -> Lsdf (0, 0, 3, 0)
    | Sixpence      -> Lsdf (0, 0, 6, 0)
    | Shilling      -> Lsdf (0, 1, 0, 0)
    | Florin        -> Lsdf (0, 2, 0, 0)
    | HalfCrown     -> Lsdf (0, 2, 6, 0)
    | DoubleFlorin  -> Lsdf (0, 4, 0, 0)
    | Crown         -> Lsdf (0, 5, 0, 0)
```

Code a function `totalValue : oldCoin list -> oldCurrency` that calculates the (normalized) value of a bag of coins represented as a list.

```
> totalValue [Crown; Crown; Crown; HalfCrown; HalfCrown; Florin; Florin; Shilling;
    Sixpence; Sixpence; Threepence; Halfpenny; Farthing; Farthing; Farthing];;
> val it : oldCurrency = Lsdf (1, 6, 4, 1)
```

On the Decimal Day, the pound kept its old value, but the smaller units were replaced with the new pence (p), one pound equalling 100 new pence.

```
type decCurrency =
    | Lp of (int * float)              // p amounts rounded to the nearest half
```

Code a function `old2dec : oldCurrency -> decCurrency` that converts an old currency amount to a (normalized) new currency amount to the precision of 0.5 p.

```
> old2dec (Lsdf (1, 6, 4, 1));;
val it : decCurrency = Lp (1, 32.0)
```

7. Complex numbers can be represented as an F# datatype

```
type cmplx =
    | C of float * float
```

Code addition and multiplication of two complex numbers as functions
`(.+) : cmplx -> cmplx -> cmplx` and `(.*) : cmplx -> cmplx -> cmplx`.

```
> C (0.0, 1.0) .* C (0.0, 1.0);;
val it : cmplx = C (-1.0, 0.0)
```

Alternatively, complex numbers can also be represented as a record type

```
type cmplx = { re : float; im : float }
```

Code addition and multiplication also for this representation.