Synopsis Lab 6

6.1 Getting start with Haskell (ghc)

- to start the interpreter:

  `ghci`

- to quit, type:

  `:q`

- source files: name ends either in `.lhs` (literate Haskell code, preferred) or in `.hs` (Haskell code); please note that indenting the code is crucial. Use SPACES for code indentation.

- to load the source file `f1.lhs`:

  `:l f1.lhs`

- to reload the last loaded file:

  `:r`

- file comments: in `.lhs` files, the CODE blocks are enclosed betweeen `\begin{code}` and `\end{code}`; everything else is a comment. In `.hs` files, pairs `{-` and `-}` delimit multiple line comments. In both cases, the pair `--` starts a comment which ends at the end of the line.

6.2 Getting start with ML (polyML)

- to start the interpreter (`rlwrap` offers access to command history, pairing parentheses etc.):

  `rlwrap poly`

- to quit, simultaneously press keys `Ctrl` and `D` (hereinafter written `Ctrl+D`)

- source files: extension `.ml`

- to load the source file `f1.ml`:

  `use "f1.ml";`

  If this does not work, replace `use` above with `PolyML.use`

- file comments start with the pair of character `(*` and end with `*)`

- to modify the number of list elements printed by the system:

  `PolyML.print_depth 500;`

6.3 Some type issues

When using `ghci`, both `1 + 2.0` and `1.0 + 2.0` are handled with no problems, as the `+` operator has type `Num a => a -> a -> a`. Now, assume you need to compute the average of some integers stored in a list. In this case, you'll need to employ `fromIntegral` to convert the type `Int` returned by `length` into a `Fractional`, as required by the division operator, because the `Int` type does not have a `Fractional` instance. Example[1]:

```
Prelude> :type (/)
(/) :: Fractional a => a -> a -> a

Prelude> :set +t -- ask ghci to display the type of each entity
Prelude> sum [1,2,3,4] / length [1,2,3,4]

<interactive>:3:1: error:
     No instance for (Fractional Int) arising from a use of /
     In the expression: sum [1, 2, 3, 4] / length [1, 2, 3, 4]
      In an equation for it:
          it = sum [1, 2, 3, ....] / length [1, 2, 3, ....]

Prelude> sum [1,2,3,4]
10
it :: Num a => a

Prelude> length [1,2,3,4]
4
it :: Int

Prelude> fromIntegral (length [1,2,3,4])
4
it :: Num b => b

Prelude> sum [1,2,3,4] / fromIntegral (length [1,2,3,4])
2.5
it :: Fractional a => a
```

In ML, `1 + 2.0;` will fail:

```
> poly: : error: Type error in function application.
   Function: + : int * int -> int
   Argument: (1, 2.0) : int * real
   Reason:
      Can't unify int (*In Basis*) with real (*In Basis*)
         (Different type constructors)
Found near 1 + 2.0
```

The following will work well: `real 1 + 2.0;` and will return `val it = 3.0: real` as expected.

[1]Here, `Prelude>` is the `ghci` prompt; you need to type only the subsequent commands

6.4 Tasks

1. Skim lectures #1, #2 and #3. Find out and test the Haskell and ML functions which perform the following operations: extract the list head and tail, test if a list is empty, compute list length, list append, take, drop, reverse, compute factorial of a number, infix operator xor. Tip: check the `src` folder for (some of) the examples in the lectures.

2. (Haskell, ML) Write a function `myplus` with 2 numbers `x` and `y` as arguments, which returns their sum. E.g. in `ghci`:

   ```
   Prelude> :l myplus.lhs
   Main> myplus 1 2
   3
   ```

   and in `poly`:

   ```
   > use "myplus.ml";
   val myplus = fn : int -> int -> int
   val it = () : unit
   > myplus 1 2;
   val it = 3 : int
   ```

3. (Haskell, ML) Write a function `sudan` which computes the values of the function of Sudan (denoted by `S`), defined as follows:

$$S\ n\ x\ y = \begin{cases} x + y & \text{if n = 0} \\ x & \text{if n > 0 and y=0} \\ S\ (n-1)\ (S\ n\ x\ (y-1))\ (y + (S\ n\ x\ (y-1))) & \text{otherwise} \end{cases}$$

4. (Haskell, ML) Define an infix operator called `!&` (in Haskell), respectively `nand` (in ML), which implements the logical function `nand` (`not and`):

$$x\ nand\ y = \begin{cases} false & \text{if x=y=true} \\ true & \text{otherwise} \end{cases}$$

5. (Haskell) Write a function `myinit` which takes the first `n-1` elements from a list of `n` elements. E.g.:

   ```
   Main> myinit [1,2,3,4,5]
   [1,2,3,4]
   ```

6. (Haskell) Write a function `lastbut2` which returns the last but 2 element of a list. E.g.:

   ```
   Prelude> lastbut2 [1,2,3,4,5,6]
   4
   ```

7. (Haskell) Write a function `palindrome` which tests if a list in palindromic. E.g.:

   ```
   Main> palindrome [1,2,3,2,1]
   True
   ```