## Project 0

In this assignment, you will familiarize yourself with functional programming in Standard ML by implementing functions of various types. Each function can be implemented with no more than a few lines of code, but requires a bit of thinking. So this assignment should not be painstaking programming; rather it will be great fun!

Before we start, we need to have our workspace organized.

- Download the archive

  `www.math.us.edu.pl/~pgladki/teaching/2016-2017/tke_lab0.zip`

  and read it. This assignment consists of two parts, the files are to be found in two directories: *part_one* and *part_two*.
- If you haven't done so far, go ahead and instal SML.
- In order to compile your code, run SML in the same directory where you unpacked the archive `tke_lab0.zip`, that is either in*part_one* or in *part_two*.
- Type **CM.make "sources.cm";** This should load and compile all sources for each directory.

After downloading and unpacking `tke_lab0.zip` you should find `czesc1.sml`, `czesc1-sig.sml` and `sources.cm` in `part_one` and `czesc2.sml`, `czesc2-sig.sml` and `sources.cm` in `part_two`. You will be editing `czesc1.sml` i `czesc2.sml`, don't change anything in `czesc1-sig.sml`, `czesc2-sig.sml` or in `sources.cm`. The stub file `czesc1.sml` looks like this:

```
structure foo :>PART_ONE =
struct
  exception NotImplemented
  datatype 'a tree= Leaf of 'a | Node of 'a tree * 'a * 'a tree
  fun sum _ = raise NotImplemented
  fun fac _ = raise NotImplemented
  ...
end
```

and the stub file `czesc2.sml` like that:

```
structure foo :>PART_TWO =
struct
  exception NotImplemented
  datatype 'a tree = Leaf of 'a | Node of 'a tree * 'a * 'a tree
  funfact =raiseNotImplemented
  ...
end
```

Fill the function body with your own code. This is absolutely crucial; if you leave code that does not compile, you will receive no credit. If you cannot implement a function, just leave it intact!

Spend a few good hours working hard on the assignment. Try to implement as many functions as possible. To run your program on the Standard ML interpreter, use the Compile Manager. And whenever you compile your source code, you should `open` your `structure`.

Here is a sample session using the provided file `sources.cm` in `part_one`:

```
[foo:38 ] sml
Standard ML of New Jersey v110.58 [built: Fri Mar 03 15:32:15 2006]
- CM.make "sources.cm";
[autoloading]
.....
[New bindings added.]
val it = true : bool
- open foo;
opening foo
  exception NotImplemented
  datatype 'a tree = Leaf of 'a | Node of 'a tree * 'a * 'a tree
  val sum : int ->int
  val fac : int ->int
  ...
- sum 10;
val it = 55 : int;
- fac 10;
uncaught exception NotImplemented
  raised at: hw1.sml:5.27-5.41
```

Once you decide you're done with everything, send the files `czesc1.sml` and `czesc2.sml` to your instructor. The due date is **November 14th, 2016**.

## 1. Part one

Before you start, read Chapter 1 from Benjamin C. Pierce, *Types and Programming Languages,* MIT Press, 2002 and Chapter 1, Chapter 2, Section 3.1 and Section 3.2 from Riccardo Puccella, *Notes on Programming SML/NJ*:

`http://www.cs.cornell.edu/riccardo/smlnj.html`

Your code should strictly follow the Standard ML style guide. This is not a crucial requirement, but a good programming style helps not only the reader understand your code but also yourself better develop the code.

For this assignment, **do not** use any library functions provided by Standard ML.

OK, here we go:

### 1.1. **Functions on integers.**

1.1.1. `sum` *for adding integers* 1 *to* $n$ *(inclusive).* (2 points)

(Type) `sum : int ->int`

(Description) `sum` $n$ returns $\sum_{i=1}^{n} i$

(Invariant) $n > 0$.

(Example)

```
- sum 10;
val it = 55 : int
```

1.1.2. *Factorial* `fac`. (2 points)
(Type) `fac: int ->int`
(Description) `fac` $n$ returns $\prod_{i=1}^{n} i$.
(Invariant) $n > 0$.

1.1.3. *Fibonacci sequence* `fib`. (1 point)
(Type) `fib: int ->int`
(Description) `fib` $n$ returns `fib` $(n-1)$ + `fib` $(n-2)$ if $n \geq 2$ and 1 if $n = 0$ or $n = 1$.
(Invariant) $n \geq 0$.

1.1.4. *Greatest common denominator* `gcd`. (2 points)
(Type) `gcd: int * int ->int`
(Description) `gcd` $(m, n)$ returns the greatest common denominator of $m$ and $n$ computed with the Euclidean algorithm
(Invariant) $m \geq 0$, $n \geq 0$, $m + n > 0$.

1.1.5. *Maximum* `max` *from a list.* (2 points)
(Type) `max: int list ->int`
(Description) `max` $l$ returns the greatest integer from the list $l$. If the list is empty, it returns 0.
(Example) `max [5,3,6,7,4]` returns `7`.

1.2. **Functions on binary trees.**

1.2.1. `sumTree` *for computing the sum of integers stored in a binary tree.* (2 points)
(Type) `sumTree : int tree ->int`
(Description) `sumTree` $t$ returns the sum of integers stored in the tree $t$
(Example) `sumTree (Node (Node (Leaf 1, 3, Leaf 2), 7, Leaf 4))` returns `17`.

1.2.2. `depth` *for computing the depth of tree.* (2 points)
(Type) `depth : 'a tree ->int`
(Description) `depth` $t$ returns the length of the longest path from the root to leaf
(Example) `depth (Node (Node (Leaf 1, 3, Leaf 2), 7, Leaf 4))` returns `2`

1.2.3. `binSearch` *for searching an element in a binary search tree.* (2 points)
(Type) `binSearch : int tree ->int ->bool`
(Description) `binSearch` $t$ $x$ returns `true` if $x$ is in $t$ and `false` otherwise.
(Niezmiennik) $t$ is a binary search tree: all numbers in a left subtree are smaller than the number of the root, and all numbers in a right subtree are greater than the number of the root. We further assume that all numbers are distinct.
(Example) `binSearch (Node (Node (Leaf 1, 2, Leaf 3), 4, Leaf 7)) 2` returns `true`. `binSearch (Node (Node (Leaf 1, 2, Leaf 3), 4, Leaf 7)) 5` returns `false`.

1.2.4. `preorder` *for a preorder traversal of binary trees.* (2 points)
(Type) `preorder: 'a tree ->'a list`
(Description) `preorder` $t$ returns a list of elements produced by a preorder traversal of the tree $t$.
(Example) `preorder (Node (Node (Leaf 1, 3, Leaf 2), 7, Leaf 4))` returns `[7, 3, 1, 2, 4]`.

1.3. **Functions on lists of integers.**

1.3.1. `listAdd` *or adding each pair of integers from two lists.* (2 points)
(Type) `listAdd: int list ->int list ->int list`
(Description) `listAdd [a,b,c,...] [x,y,z,...]` returns `[a+x,b+y,c+z,...]`.
(Example) `listAdd [1, 2] [3, 4, 5]` returns `[4, 6, 5]`.

1.3.2. `insert` *for inserting an element into a sorted list.* (2 points)
(Type) `insert: int ->int list ->int list`
(Description) `insert` $m$ $l$ inserts $m$ in the sorted list $l$.
(Invariant) $l$ is sorted in ascending order.
(Example) `insert 3 [1, 2, 4, 5]` returns `[1, 2, 3, 4, 5]`.

1.3.3. `insort` *for insertion sort.* (2 points)
(Type) `insort: int list ->int list`
(Description) `insort` $l$ returns a sorted list of elements in $l$.
(Example) `insort [3, 7, 5, 1, 2]` returns `[1, 2, 3, 5, 7]`.

1.4. **Higher-order functions.**

1.4.1. `compose` *for functional composition.* (2 points)
(Type) `compose: ('a ->'b) ->('b ->'c) ->('a ->'c)`
(Description) `compose` $f$ $g$ returns $g \circ f$ .
(Remark) You should use only two arguments to implement `compose`.Thus, something like this:

```
fun compose f g = ...
```

is OK, but that:

```
fun compose f g x = ...
```

is not.

1.4.2. `curry` *for currying.* (2 points)
(Type) `curry: ('a * 'b ->'c) ->('a ->'b ->'c)`
(Description) We have a choice of how to write functions of two or more arguments. Functions are in *curried* form if they take arguments one at a time. *Uncurried* functions take arguments as a pair. `curry` $f$ transforms an uncurried function $f$ into a curried version.
(Example)

```
fun multiply x y = x * y (* curried *)
fun multiplyUC (x, y) = x * y (* uncurried *)
```

Applying `curry` to `multiplyUC` yields `multiply`.

1.4.3. `uncurry` *for uncurrying.* (2 points)
(Type) `uncurry: ('a ->'b ->'c) ->('a * 'b ->'c)`
(Description) See above.
(Example) Applying `uncurry` to `multiply` yields `multiplyUC`.

1.4.4. `multifun` *for applying a function n-times.* (2 points)
(Type) `multifun : ('a ->'a) ->int ->('a ->'a)`
(Description) `(multifun f n) x` returns $\underbrace{\texttt{f(f(}\ldots\texttt{f(x)))}}_{n \text{ times}}$
(Example) `(multifun (fn x => x + 1) 3) 1` returns `4`.
`(multifun (fn x => x * x) 3) 2` returns `256`.
(Invariant) $n \geq 1$.

1.5. **Functions on `'a list`.**

1.5.1. `ltake` *for taking the list of the first i element of l.* (2 points)
(Type) `ltake: 'a list ->int ->'a list`
(Description) `ltake` $l$ $n$ returns first $n$ elements of $l$. If $n > l$, it returns $l$.
(Example) `ltake [3, 7, 5, 1, 2] 3` returns `[3,7,5]`.
`ltake [3, 7, 5, 1, 2] 7` returns `[3,7,5,1,2]`.
`ltake ["s","t","r","i","k","e","r","z" ] 5` returns `["s","t","r","i","k"]`.

1.5.2. `lall` *for examining a list.* (2 points)
(Type) `lall : ('a ->bool) ->'a list ->bool`
(Description) `lall` $f$ $l$ returns `true` if for every element $x$ of the list $l$, $fx$ returns `true`; otherwise it returns `false`.
(Example) `lall (fn x =>x >0) [1, 2, 3]` returns `true`.
`lall (fn x =>x >0) [ 1, 2, 3]` returns `false`.

1.5.3. `lmap` *for converting a list into another list.* (3 points)
(Typ) `lmap : ('a ->'b) ->'a list ->'b list`
(Opis) `lmap` $f$ $l$ applies $f$ to each element of $l$ from left to right, returning the list of results.
(Example) `lmap (fn x => x + 1) [1, 2, 3]` returns `[2, 3, 4]`.

1.5.4. `lrev` *for reversing a list.* (3 punkty)
(Type) `lrev: 'a list ->'a list`
(Description) `lrev` $l$ reverses $l$.
(Example) `lrev [1, 2, 3, 4]` returns `[4, 3, 2, 1]`.

1.5.5. `lzip` *for pairing corresponding members of two lists.* (3 points)
(Type) `lzip: ('a list * b' list) ->('a * 'b) list`
(Description) `lzip` $([x_1, \ldots, x_n], [y_1, \ldots, y_n]) \Rightarrow [(x_1, y_1), \ldots, (x_n, y_n)]$. If two lists differ in length, ignore surplus elements.
(Example) `lzip (["Rooney","Park","Scholes","C.Ronaldo"],[8,13,18,7,10,12])` returns `[("Rooney"`

1.5.6. `split` *for splitting a list into two lists.* (3 points)
(Type) `split: 'a list ->'a list * 'a list`
(Description) `split` $l$ returns a pair of two lists. The first list consists of elements in odd positions and the second consists of elements in even posistions in a given list respectively.
(Example) `split [1, 3, 5, 7, 9, 11]` returns `([1, 5, 9], [3, 7, 11])`.

1.5.7. `cartprod` *for the Cartesian product of two sets.* (3 points).
(Type) `cartprod: 'a list ->'b list ->('a * 'b) list`
(Description) `cartprod` $S$ $T$ returns the set of all pairs $(x, y)$ with $x \in S$ and $y \in T$. The order of elements matters:
`cartprod` $[x_1, \ldots, x_n]$ $[y_1, \ldots, y_n]$ $\Rightarrow$ $[(x_1, y_1), \ldots, (x_1, y_n), (x_2, y_1), \ldots, (x_n, y_n)]$.
(Example) `cartprod [1, 2] [3, 4, 5]` $\Rightarrow$ `[(1,3), (1,4), (1,5), (2,3), (2,4), (2,5)]`.

## 2. Part two

### 2.1. **Recursive functions.**

2.1.1. `lconcat` *for concatenating a list of lists.* (3 points)
(Type) `lconcat : 'a list list ->'a list`
(Description) `lconcat` $l$ concatenates all elements of $l$.
(Example) `lconcat [[1, 2, 3], [6, 5, 4], [9]]` returns `[1, 2, 3, 6, 5, 4, 9]`.

2.1.2. `lfoldl` *for left folding a list.* (3 points)
(Type) `lfoldl: ('a * 'b ->'b) ->'b ->'a list ->'b`
(Description) `lfoldl` $f$ $e$ $l$ takes $e$ and the first item of $l$ and applies $f$ to them, then feeds the function with this result and the second argument and so on.
`lfoldl` $f$ $e$ $[x_1, x_2, \ldots, x_n]$ returns $f(x_n, \ldots, f(x_2, f(x_1, e))\ldots)$ or $e$ if the list is empty

### 2.2. **Tail recursive functions.**

For each description below, give a tail recursive implementation. In all cases except for `union`, you want to introduce a tail recursive helper function; the main function is not recursive but just invokes the helper function with appropriate arguments. For example, a tail recursive implementation of `sumList` may look like

```
fun sumList inputList =
let
  fun sumList' l accum =
    ...
in
  sumList' inputList 0
end
```

where `sumList'` is tail recursive.
In the case of `union` you may want to introduce some local helper functions. The main function itself is tail-recursive and may use those helper functions.

2.2.1. `fact` *for factorial.* (1 point)
(Type) `fact: int ->int`
(Description) `fact` $n$ returns $\prod_{i=1}^{n} i$.
(Invariant) $n \geq 0$.

2.2.2. `power` *for powers.* (1 point)
(Type) `power: int ->int ->int`
(Description) `power` $x$ $n$ returns $x^n$.
(Invariant) $n \geq 0$.

2.2.3. `fib` *for Fibonacci sequence.* (1 point)
(Type) `fib: int ->int`
(Description) `fib` $n$ returns `fib` $(n-1)$ + `fib` $(n-2)$ if $n \geq 2$ and 1 if $n = 0$ or $n = 1$.
(Invariant) $n \geq 0$.

2.2.4. `lfilter` *for filtering a list.* (1 punkt)
(Type) `lfilter : ('a ->bool) ->'a list ->'a list`
(Description) `lfilter` $p$ $l$ returns all elements of $l$ that satisfies the predicate $p$.
(Example) `lfilter (fn x =>x >2) [0, 1, 2, 3, 4, 5]` returns `[3, 4, 5]`.

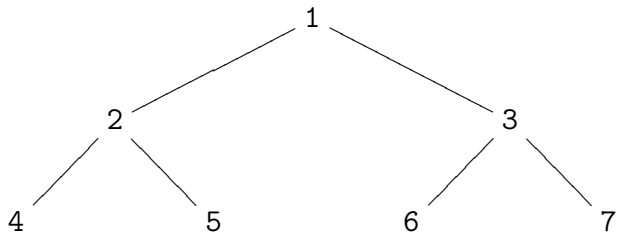
2.2.5. `ltabulate`. (1 punkt)
(Type) `ltabulate : int ->(int ->'a) ->'a list`
(Description) `ltabulate` $n$ $f$ applies $f$ to each element of the list `[0, 1, ..., n-1]`.
(Example) `ltabulate 4 (fn x =>x * x)` returns `[0, 1, 4, 9]`.
(Invariant) $n \geq 0$

2.2.6. `union` *for union of two sets.* (3 points)
(Type) `union: ''a list ->''a list ->''a list`
(Description) `union` $S$ $T$ returns a set that includes all elements of $S$ and $T$ without duplication of any element. Note that all list elements have an equality type as indicated by equality type variable `''a`. The order of elements in the return value does not matter.
(Invariant) Each set consists of distinct elements.
(Example) `union [1, 2, 3] [2, 4, 6]` returns `[3, 1, 2, 4, 6]`.

2.2.7. `inorder` *for an inorder traversal of binary trees.* (4 points)
(Type) `inorder: 'a tree ->'a list`
(Description) `inorder` $t$ returns a list of elements produced by an inorder traversal of the tree $t$
(Example) `inorder (Node (Node (Leaf 1, 3, Leaf 2), 7, Leaf 4))` returns `[1, 3, 2, 7, 4]`.
(Example) `inorder` can be implemented as follows:

```
fun inorder t =
  let
    fun inorder' (t' : 'a tree) (post : 'a list) : 'a list = ...
  in
    inorder' t [ ]
  end
```

`post` will be a list of elements to be appended to the result of an inorder traversal of `t'`. For example, when `inorder'` visits the node marked 2 in the tree below, `post` will be bound to `[1, 6, 3, 7]`.



2.2.8. `postorder` *for a postorder traversal of binary trees.* (4 points)
(Type) `postorder: 'a tree ->'a list`
(Description) `postorder` $t$ returns a list of elements produced by a postorder traversal of the tree $t$
(Example) `postorder (Node (Node (Leaf 1, 3, Leaf 2), 7, Leaf 4))` returns `[1, 2, 3, 4, 7]`.

2.2.9. `preorder` *for a preorder traversal of binary trees.* (4 points)
(Type) `preorder: 'a tree ->'a list`
(Description) `preorder` $t$ returns a list of elements produced by a preorder traversal of the tree $t$
(Example) `preorder (Node (Node (Leaf 1, 3, Leaf 2), 7, Leaf 4))` returns `[7, 3, 1, 2, 4]`.

2.3. **Sorting in the ascending order.**

2.3.1. `quicksort` *for quick sorting.* (4 points)
(Type) `quicksort: int list ->int list`
(Description) `quicksort` $l$ iimplements quick sorting by selecting the first element of $l$ as a pivot.
(Example) `quicksort [3, 7, 5, 1, 2]` selects `3` as a pivot to obtains two sublists `[1, 2]` and `[5, 7]` to be sorted independently.

2.3.2. `mergesort` *for merge sorting.* (4 points)
(Type) `mergesort: int list ->int list`
(Description) `mergesort` $l$ divides $l$ into two sublists, sorts each sublist, and then merges the two sorted sublists. If the length of $l$ is even, then the two sublists are of equal length. If not, one sublist has one more element than the other.

2.4. **Structures.** The goal of this part is to learn *modular programming* in SML – structures and signatures. We will first implement a structure for heaps. You should keep in mind that this data structure is not an ordinary heap data structure. You had better think of it as a mechanism for dynamic memory allocation. See the explanation below carefully.

2.4.1. `Heap` *for heaps.* (5 points)
The structure `Heap` conforms to the signature `HEAP`. A heap is a mechanism for dynamic memory allocation.

```
signature HEAP =
sig     exception InvalidLocation
  type loc
  type 'a heap
  val empty : unit ->'a heap
  val allocate : 'a heap ->'a ->'a heap * loc
  val dereference : 'a heap ->loc ->'a
  val update : 'a heap ->loc ->'a ->'a heap
end
```

- `loc` is the internal representation of location, which is similar to the *pointer* of C language. `type loc` is not visible to the outside of the structure;
- `'a heap` is a heap for the type `'a`;
- `empty ()` returns an empty heap;

- `allocate` $h$ $v$ allocates the given value $v$ to a fresh heap cell and returns the pair $(h', l)$ of the updated heap $h'$ and the location $l$ of this cell;
- `dereference` $h$ $l$ fetches the value $v$ stored in the heap cell at location $l$. `InvalidLocation` is raised if the $l$ is an invalid `loc`;
- `update` $h$ $l$ $v$ updates the heap cell at location $l$ with the given value $v$ and returns the updated heap $h'$. `InvalidLocation` is raised if the $l$ is an invalid `loc`.

2.4.2. *Sygnatura* `DICT`. `DICT` is a signature for dictionaries.

```
signature DICT =
sig
  type key
  type 'a dict
  val empty : unit ->'a dict
  val lookup : 'a dict ->key ->'a option
  val delete : 'a dict ->key ->'a dict
  val insert : 'a dict ->key * 'a ->'a dict
end
```

- `empty ()` returns an empty dictionary;
- `lookup` $d$ $k$ searches the key $k$ in the dictionary $d$. If the key is found, it returns the associated item. Otherwise, it returns `NONE`;
- `delete` $d$ $k$ deletes the key $k$ and its associated item in the dictionary $d$ and returns the resultant dictionary $d'$. If the key does not exist in the dictionary $d$, it returns the given dictionary $d$ without any modification;
- `insert` $d$ $(k, v)$ inserts the new key $k$ and its associated item $v$ in the dictionary $d$. If the key $k$ already exists in the dictionary $d$, it just updates its associated item with the given item $v$.

2.4.3. *Structure* `DictList`. (5 points)

Implement the structure `DictList` of signature `DICT` with the definition `'a dict = (key * 'a) list`.

The structure `DictList` uses a list of pairs as the representation of a dictionary. The implementation should be straightforward because a list of pairs itself may be thought of as a dictionary.

2.4.4. *Structure* `DictFun`. (5 points)

Implement the structure `DictFun` of signature `DICT` with the definition `'a dict = key ->'a option`.

The structure `DictFun` uses a "functional representation" of dictionaries. The idea is that we represent a dictionary as a function that, given a key, returns an associated item. The implementation of `DictFun` may be either very difficult or just a piece of cake depending on how familiar you are with "functional thinking." My advice is: forget about everything that you have learned so far about imperative programming; just "think functionally!" You will be amazed at the conciseness of your code once you figure it out.