Foundations of Computer Science	**Homework 1**
ENGR 3520 Spring 2015	**Due Date: Thursday, Feb 5, 2015**

Note on integrity: This homework is meant to be done individually. You may discuss problems with fellow students, but all work must be entirely your own, and should not be from any other course, present, past, or future. If you use a solution from another source you must cite it, including if that source is other people who help you.

Every source file you submit should begin with a block comment that lists

- Your name
- Your email address
- Any remarks that you wish to make to the instructor

Notes

- All programming to be done in OCaml v4.
- Please code your answers by modifying the file `hw1-code.ml` provided on the website.
- This homework has both compulsory questions and discretionary questions.
- Questions that do not require coding (if any) can either be answered in the `hw1-code.ml` file as a *clearly indicated comment*, or on paper handed in at the beginning of class on the due date.
- Electronic submission instructions to be determined in the next few days. Stay tuned.

Compulsory Exercises

Please complete all the following questions.

(1) Code the following functions:[1]

(a) A function `append` of type `'a list * 'a list -> 'a list` that takes two lists and appends them together.

```
# append([], []);;
- : 'a list = []
# append([1;2;3], [3;4;5]);;
- : int list = [1; 2; 3; 3; 4; 5]
# append(["a";"b";"c"], []);;
- : string list = ["a"; "b"; "c"]
# append([], [1.0; 2.0; 3.0]);;
- : float list = [1.; 2.; 3.]
```

(b) A function `flatten` of type `'a list list -> 'a list` that takes a list of lists and flattens it into a single list.

```
# flatten [];;
- : 'a list = []
# flatten [[1;2;3]];;
- : int list = [1; 2; 3]
# flatten [[1;2;3];[4;5;6]];;
- : int list = [1; 2; 3; 4; 5; 6]
# flatten [["a"];["b";"c"];[];["d";"e";"f"]];;
- : string list = ["a"; "b"; "c"; "d"; "e"; "f"]
```

(c) A function `double` of type `int list -> int list` that takes a list of integers and returns the list of all the elements in the original list but doubled.

```
# double [];;
- : int list = []
# double [1];;
- : int list = [2]
# double [1;2;3];;
- : int list = [2; 4; 6]
# double [10;20;30;40;50];;
- : int list = [20; 40; 60; 80; 100]
```

[1] Code them up from scratch, even if similar functions already exist in the libraries.

(d) A function `last` of type `'a list -> 'a option` that takes a list of elements and returns the last element, if one exists.

The return value of the function should be of type `'a option`, which is a *built-in type*[2] defined as follows:

```
type 'a option = Some of 'a | None
```

A value of type `'a option` is therefore either `Some v` for some value `v` of type `'a`, or `None`. This type is often used as the return type of functions that sometimes return a value and sometimes don't.

```
# last [];;
- : 'a option = None
# last [1];;
- : int option = Some 1
# last [1;2;3;4];;
- : int option = Some 4
# last ["a";"b";"c"];;
- : string option = Some "c"
```

[2]meaning: you don't have to define it!

(2) Mathematically, a *set* is a collection of elements where repetition and order is irrelevant: the sets $\{1, 1, 2, 3\}$ and $\{2, 3, 3, 1\}$ are considered to be the same set.

In this question, we use lists to represent sets, and define set operations on those lists where the lists are interpreted as sets (meaning that order and repetition of elements is irrelevant).

(a) Code a function `setIn` of type `('a * 'a list) -> bool` where `setIn(a,S)` returns `true` if and only if element `a` is an element of list `S`.

```
# setIn (1,[]);;
- : bool = false
# setIn (1,[2;3]);;
- : bool = false
# setIn (1,[1;2;3]);;
- : bool = true
# setIn (1,[3;4;4;1;1]);;
- : bool = true
```

(b) Code a function `setSub` of type `('a list * 'a list) -> bool` where `setSub(S,T)` returns `true` if and only if `S` is a subset of `T` (where `S` and `T` are interpreted as sets).

```
# setSub ([],[]);;
- : bool = true
# setSub ([],[1;1;1]);;
- : bool = true
# setSub ([1],[1;1;1]);;
- : bool = true
# setSub ([1;1],[1;1;1]);;
- : bool = true
# setSub ([1;1],[1;2;3]);;
- : bool = true
# setSub ([1;1],[2;3]);;
- : bool = false
# setSub ([1],[]);;
- : bool = false
```

(c) Code a function `setEqual` of type `('a list * 'a list) -> bool` where `setEqual(S,T)` returns `true` if and only if `S` is equal to `T` *as sets*.

```
# setEqual ([],[]);;
- : bool = true
# setEqual ([1],[1]);;
- : bool = true
```

```
# setEqual ([1],[1;1;1]);;
- : bool = true
# setEqual ([1;1;1],[1;1]);;
- : bool = true
# setEqual ([1;2],[1;2;3]);;
- : bool = false
# setEqual ([1;2],[2;1]);;
- : bool = true
# setEqual ([1;1;2],[2;2;1]);;
- : bool = true
```

(d) Code a function `setUnion` of type `('a list * 'a list) -> 'a list` where `setUnion(`S`,`T`)` returns a list representing a set equal to the union of S and T interpreted as sets.

```
# setEqual ([],[]);;
- : bool = true
# setEqual ([1],[1]);;
- : bool = true
# setEqual ([1],[1;1;1]);;
- : bool = true
# setEqual ([1;1;1],[1;1]);;
- : bool = true
# setEqual ([1;2],[1;2;3]);;
- : bool = false
# setEqual ([1;2],[2;1]);;
- : bool = true
# setEqual ([1;1;2],[2;2;1]);;
- : bool = true
# setEqual(setUnion([1;2;3],[4;5;6]),[1;2;3;4;5;6]);;
- : bool = true
# setEqual(setUnion([1;2],[2;3;3]),[1;2;3]);;
- : bool = true
# setEqual(setUnion([1;2],[2;1]),[1;2]);;
- : bool = true
# setEqual(setUnion([],[]),[]);;
- : bool = true
# setEqual(setUnion([],[1;1]),[1]);;
- : bool = true
# setEqual(setUnion([1;2],[]),[2;1]);;
- : bool = true
```

(e) Code a function `setInter` of type `('a list * 'a list) -> 'a list` where

`setInter(`S`,`T`)` returns a list representing a set equal to the intersection of S and T interpreted as sets.

```
# setEqual(setInter([],[]),[]);;
- : bool = true
# setEqual(setInter([1;2],[1]),[1]);;
- : bool = true
# setEqual(setInter([1;2],[2;3]),[2]);;
- : bool = true
# setEqual(setInter([1;2;3],[3;3;2;2]),[2;3]);;
- : bool = true
# setEqual(setInter([],[1;2;3]),[]);;
- : bool = true
# setEqual(setInter([1;2;3],[]),[]);;
- : bool = true
```

(f) (Challenging) Code a function `setSize` of type `'a list -> int` where `setSize(`S`)` returns the number of elements in S interpreted as a set.

```
# setSize ([]);;
- : int = 0
# setSize ([1]);;
- : int = 1
# setSize ([1;2;3]);;
- : int = 3
# setSize ([1;1;1;2;2;2;3;3;3;4;4;4]);;
- : int = 4
# setSize ([1;2;3;2;1]);;
- : int = 3
```

(3) Define the following *record type* for rational numbers:

```
type rat = {num: int; den: int}
```

A record is a structure where the values in the structure can be accessed by *field name*. (This is similar to a `struct` in some languages—think of it as a limited form of dictionary where the keys are fixed and specified in advance.)

We can construct values of record type `rat`:

```
let half = {num = 1; den = 2}
let third = {num = 1; den = 3}
let fourth = {num = 1; den = 4}
```

We can also write code to access values inside records:

```
let floatR (r) =
  float(r.num) /. float(r.den)
```

Function `floatR` of type `rat -> float` takes a rational number `r` and computes its floating-point value by simply dividing the numerator and denominator as `float`s. You see that to access the field of a record `r`, we use dot notation (*`foo.bar`*).

(a) Code a function `simplify` of type `rat -> rat` taking a rational number and simplifying it so that the numerator and denominator are as small as possible. You may need auxiliary functions to help you. Watch out for negative numbers.

```
# simplify {num = 10; den = 20};;
- : rat = {num = 1; den = 2}
# simplify {num = 30; den = 35};;
- : rat = {num = 6; den = 7}
# simplify {num = 6; den = 4};;
- : rat = {num = 3; den = 2}
# simplify {num = 1; den = 2};;
- : rat = {num = 1; den = 2}
# simplify {num = -2; den = 4};;
- : rat = {num = -1; den = 2}
# simplify {num = -2; den = -4};;
- : rat = {num = 1; den = 2}
# simplify {num = 2; den = -4};;
- : rat = {num = -1; den = 2}
```

(b) Code two functions `addR` and `multR` of type `rat * rat -> rat` taking two rational numbers and adding them and multiplying them, respectively. The result should be a simplified rational number.

```
# addR(half, half);;
- : rat = {num = 1; den = 1}
# addR(half, third);;
- : rat = {num = 5; den = 6}
# addR(half, fourth);;
- : rat = {num = 3; den = 4}
# addR(fourth, addR(fourth, fourth));;
- : rat = {num = 3; den = 4}
# addR(fourth, addR(fourth, addR(fourth, fourth)));;
- : rat = {num = 1; den = 1}
# multR(half, half);;
- : rat = {num = 1; den = 4}
# multR(half, fourth);;
- : rat = {num = 1; den = 8}
# multR(third, {num = 3; den = 1});;
- : rat = {num = 1; den = 1}
# multR({num = 2; den = 3}, {num = 6; den = 4});;
- : rat = {num = 1; den = 1}
```

(c) Define the following type for numbers that can either be integers, rationals, or floating point:

```
type number = I of int
            | R of rat
            | F of float
```

Code a function `add` of type `number * number -> number` that takes two numbers in the above representation and returns their sum. You will need to bump up integers to rationals and rationals to floating point for some combinations of numbers you'll be trying to add. You will most likely want to use some of the functions you defined above to help you out.

```
# add(I 1, I 4);;
- : number = I 5
# add(I 1, R half);;
- : number = R {num = 3; den = 2}
# add(F 3.0, R third);;
- : number = F 3.3333333333333348
# add(I 1, F 3.5);;
- : number = F 4.5
```

Discretionary Exercises

Please complete *one* of questions (4), (5).

(4) Define the following union type for Boolean *constants*:

```
type bConst = True | False
```

Define the following type for Boolean *expressions*:

```
type bExpr = Constant of bConst
           | Variable of string
           | And of bExpr * bExpr
           | Or of bExpr * bExpr
           | Not of bExpr
```

Intuitively, a Boolean expression is a formula built up from conjunction, disjunction, negation, constants 1 and 0, and variables. Here are some Boolean expressions:

```
let sample1 = And(Not(Variable "a"), Not(Variable "b"))
let sample2 = Or(Not(Variable "a"), And(Variable "b", Constant(True)))
let sample3 = And(Variable "a", Not(Variable "a"))
```

Boolean expression `sample1` represents the formula $\neg a \wedge \neg b$, `sample2` represents the formula $\neg a \vee (b \wedge \mathit{true})$, and `sample3` represents the formula $a \wedge \neg a$.

(a) Code a function `vars` of type `bExpr -> string list` taking a Boolean expression and returning a list of all the variable names it contains.

```
# vars(Constant True);;
- : string list = []
# vars(And(Constant True, Constant False));;
- : string list = []
# vars(sample1);;
- : string list = ["a"; "b"]
# vars(sample2);;
- : string list = ["a"; "b"]
# vars(sample3);;
- : string list = ["a"]
```

Brownie points if your function eliminates duplicate names from the returned list, but it is not necessary. So returning `["a","a"]` as a result for `vars(sample3)` is perfectly fine.

(b) Code a function `subst` of type `bExpr * string * bExpr -> bExpr` that takes a source Boolean expression b, a variable name v, and a Boolean expression s, and replaces every occurrence of variable v in b by the Boolean expression s. Variables appearing in s are *not* replaced, even if they are named v.

```
# subst(sample1, "a", Constant False);;
- : bExpr = And (Not (Constant False), Not (Variable "b"))
# subst(sample1, "b", Variable "c");;
- : bExpr = And (Not (Variable "a"), Not (Variable "c"))
# subst(sample2, "a", Not(Variable "a"));;
- : bExpr = Or (Not (Not (Variable "a")), And (Variable "b", Constant
    True))
# subst(sample3, "a", Constant True);;
- : bExpr = And (Constant True, Not (Constant True))
```

(c) Code a function `eval` of type `bExpr -> bConst option` that *evaluates* a Boolean expression. Intuitively, constants evaluate to themselves, and the logical operations evaluate to what their truth tables say. Evaluation just means computing the value of the Boolean expression, and yields a value of either true or false. If a Boolean expression contains variables, it cannot be evaluated, and therefore `eval` should return `None` in that case. Otherwise, it returns `Some` c for c either `True` or `False`.

```
# eval(sample1);;
- : bConst option = None
# let sample1' = subst(sample1, "a", Constant True);;
val sample1' : bExpr = And (Not (Constant True), Not (Variable "b"))
# eval(sample1');;
- : bConst option = None
# let sample1'' = subst(sample1', "b", Constant False);;
val sample1'' : bExpr = And (Not (Constant True), Not (Constant False
   ))
# eval(sample1'');;
- : bConst option = Some False
# eval(sample3);;
- : bConst option = None
# eval(subst(sample3, "a", Constant False));;
- : bConst option = Some False
# eval(subst(sample3, "a", Constant True));;
- : bConst option = Some False
```

(5) This question does not require coding. It's old-fashioned mathematics.

let $\mathcal{U}$ be a universe of elements. Any set $A \subseteq \mathcal{U}$ can be represented by a *characteristic function* $\chi_A : \mathcal{U} \longrightarrow \{\top, \bot\}$ defined as:

$$\chi_A(x) = \begin{cases} \top & \text{if } x \in A \\ \bot & \text{if } x \notin A \end{cases}$$

Conversely, any function $f : \mathcal{U} \longrightarrow \{\top, \bot\}$ defines a set A_f as:

$$A_f = \{x \in \mathcal{U} \mid f(x) = \top\}$$

Thus, sets and functions $\mathcal{U} \longrightarrow \{\top, \bot\}$ are completely equivalent—whatever we do with sets we could do with functions $\mathcal{U} \longrightarrow \{\top, \bot\}$, and vice versa.

In particular, we can implement all the set operations using functions.

(a) Define an operation $+$ on functions $\mathcal{U} \longrightarrow \{\top, \bot\}$ with the property that $\chi_A + \chi_B = \chi_{A \cup B}$.
In other words, if apply $+$ to functions $\mathcal{U} \longrightarrow \{\top, \bot\}$ that happen to be the characteristic functions of A and B, you should get a function that happens to be the characteristic function of $A \cup B$.
Please define $+$ directly. That is, define $f + g$ without constructing sets for which f and g are characteristic functions, and without using the $\cup$ operation. Your definition should look like:
$$(f + g)(x) = ...$$

(b) Define an operation $\cdot$ on functions $\mathcal{U} \longrightarrow \{\top, \bot\}$ with the property that $\chi_A \cdot \chi_B = \chi_{A \cap B}$. Same caveats as in part (a).

(c) Define an operation $-$ on functions $\mathcal{U} \longrightarrow \{\top, \bot\}$ with the property that $\chi_A - \chi_B = \chi_{A - B}$. Same caveats as in part (a).

(d) Define an operation $\wp$ on functions $\mathcal{U} \longrightarrow \{\top, \bot\}$ with the property that $\wp(\chi_A) = \chi_{\wp(A)}$, where $\wp(A)$ is the powerset of A. Note that if f is a function $\mathcal{U} \longrightarrow \{\top, \bot\}$, then $\wp(f)$ is a function $\wp(\mathcal{U}) \longrightarrow \{\top, \bot\}$. Same caveats as in part (a).