CS 6371	**Advanced Programming Languages**	Assignment 1:
Spring 2020	Due: Jan 23, 2019 by 1:05pm	OCaml Intro

This programming assignment is to be completed in OCaml. Your solution should be submitted through eLearning as a single text file named `lastname.ml` where `lastname` is your last name. At the top of your file put an OCaml comment containing your name and email address. To grade your solutions I will add a `main()` function that calls your code with test inputs; therefore please do not name any of your types or functions "`main`". Your code should compile with: `ocamlc -o hwk1 lastname.ml`. You may use any OCaml features you wish, including any of the core or standard library functions not covered in class, but do not use any code or libraries that do not come with the standard OCaml distribution, and do not link with external code written in another language.

If you work with other students or draw any code or ideas from other sources, be sure to cite your sources using text in OCaml comments (* ... *).

(1) In mathematics, a *first-order logical sentence* (FOS) can be defined as follows:

1. The boolean constants *true* and *false* are FOSes that are true and false, respectively.
2. $s_1 \vee s_2$ is an FOS if s_1 and s_2 are both FOSes. It is true if and only if s_1 is true or s_2 is true.
3. $\neg s$ is an FOS if s is an FOS. It is true if and only if s is not true.
4. Variable names v (which we will represent as arbitrary OCaml strings) are also valid FOSes. They have boolean values (true or false).
5. $\forall v.s$ is an FOS if v is a variable name (any OCaml string) and s is an FOS. It is true if and only if s is always true irrespective of whether variable v has value *true* or *false*.
6. $\exists v.s$ is an FOS under the same conditions as the previous form. It is true if and only if s is true for at least one possible value (*true* or *false*) of variable v.

For example, $\forall x.\exists y.(x \vee y)$ is a true FOS (because for both possible values of x, there exists a value of y that makes sentence $x \vee y$ true), but $\forall x.x$ is a false FOS (because not every possible value of x makes sentence x true—in particular, value $x = \mathit{false}$ makes sentence x false).

The v appearing in item 4 is called a *variable instance*, whereas the v's appearing in items 5 and 6 are called *variable bindings*. By convention, each variable instance refers to the innermost enclosing binding of the variable name. For example, in the sentence $\forall x\,.\,((\forall x.\underline{x}) \vee \neg x)$, the underlined x (which is the first variable instance) refers to the second (innermost) of the two variable bindings. Thus, this FOS is false (because $\forall x.x$ is a false statement).

(a) (3 pts) Define an OCaml type `fos` that models the above language of FOSes. You may name your type constructors anything you wish (but follow the OCaml convention of capitalizing the first letter of constructor names). "Modeling" the language means that programmers should be able to express any valid FOS (including the false-valued FOSes) using your datatype. The syntax that the programmer uses need not be exactly the same as the syntax that mathematicians use (as defined above), as long as there is some way to express each valid FOS.

(b) (1 pts) Define a global variable named `mysentence` that models the FOS $\forall x\,.\,((\forall x\,.\,x) \vee \neg x)$. Your solution should have type `fos`.

(c) (6 pts) A *free variable* in an FOS is a variable instance that is not enclosed by any binding for that variable. For example, in sentence $(\forall x.x) \vee x$, the final x is free. (Note that only variable *instances* can be free, not variable bindings.) Implement a function `(freevars s)` that returns a list of all free variables in `s`. The returned list *may* contain duplicates if a variable appears free more than once. Your function should have type `freevars` : `fos` $\rightarrow$ `string list`.

(d) (7 pts) A *tautology* is an FOS that has no free variables and that is "true" according to the definition of FOS on page 1. Implement a function `(tautology s)` that returns `true` if and only if `s` is a tautology. Your function should have type `tautology : fos → bool`.

Hint: First implement a helper function `(istrue tfv s)` that takes as input a list `tfv` of "true free variables." When `istrue` encounters a free variable, it assumes the variable has value *true* if it's a member of list `tfv`, and assumes it has value *false* otherwise.

(e) (5 pts) Implement a function `(string_of_fos s)` that returns a string representation of sentence `s`, where *true* and *false* are denoted by the letters `"T"` and `"F"`, $\vee$ is denoted by the string `"\/"`, $\neg$ is written as a tilde `"~"`, $\forall$ is the letter `"A"`, and $\exists$ is the letter `"E"`. Use parentheses in your string to show the order of operations. For example, `(string_of_fos mysentence)` should return a string like `"Ax.((Ax.x)\/(~x))"`. Your function should have type `string_of_fos : fos → string`.

(2) (3 pts) Implement a function `(update f x y)` that accepts as input a function `f` and values `x` and `y`, and returns a new function g that is just like function `f` except that $g(x) = y$. Your solution should work no matter what types `x` and `y` have. For example, if I test it by writing:

```
let f s = s ^ "abc";;
let g = (update f "X" "Y");;
```

then `(g "A")` should yield `"Aabc"` but but `(g "X")` should yield `"Y"`. Your `update` function should have the polymorphic type: `('a→'b)→'a→'b→'a→'b`

Hint: A formal mathematical definition of g would look like this:

$$g(n) = \begin{cases} y & \text{if } n = x \\ f(n) & \text{if } n \neq x \end{cases}$$

(3) (4 pts) The notation $f^n(x)$ with $n \geq 0$ denotes $f(f(\cdots f(x)\cdots))$ with f nested n times. Implement a function `(compose_n f n)` that returns the function f^n. That is, `compose_n` expects a function `f` and an integer `n`, and returns a new function g such that $g(x) = f^n(x)$. When $n \leq 0$, `compose_n` ignores `f` and returns the identity function. For example, if I write:

```
let f n = n + 2;;
let g = (compose_n f 5);;
```

then `(g 0)` should yield 10 because $f^5(0) = f(f(f(f(f(0))))) = 10$.

Hint: In a correct solution, the `compose_n` function will have the following polymorphic type: `('a→'a)→int→'a→'a`.

(4) (6 pts) Using `List.fold_left`, implement a function `(selectall f l)` that takes as input a function `(f : int -> bool)` and a list `(l : 'a list)`, and returns a list of all the elements in `l` that are at indexes for which `f` returns true. That is, function `f` takes *integer list indexes* as input, not members of the list. List indexes start at 1. The contents of the returned list may be in any order. For example, the expression

```
selectall (fun n -> n mod 2 = 0) ["a"; "b"; "c"; "d"]
```

should return `["b"; "d"]` or `["d"; "b"]`. For full credit, your function should have a running time that is *linear* in the length of the input list.

Hint: Your accumulator for `fold_left` will need to have more information in it than just the returned list. Use a tuple as your accumulator (and base case) to store multiple values.