CS101 Homework 1

Due: Friday, Oct 14, 1PM (sharp)

Collaboration: You are allowed and encouraged to work together and collaborate with others. However, your submission must be your own; you must write up your homework without referring to material developed with other groups.

You may use the WWW for reference material. You may use the material you found to develop your understanding, but your submission must be your own.

Summary: you may use any and all resources at your disposal, but your submission must be your own work.

Part I: λ-Calculus

1. Which of the following expressions are well-typed? For each expression that you believe to be well-typed, rewrite it by turning each untyped "$\lambda x.e$" into an appropriately typed "$\lambda x : t.e$" and provide a typing derivation (please annotate each step with the name of the rule you used, as we did in class). If there is more than one way to pick the types, you may pick any types that work.

a. $\lambda x.(x\ x)$

b. $(\lambda x.1)\ (\lambda x.(x\ x))$

c. $\lambda x.\lambda y.(x\ (y\ x))$

d. $\lambda y.((\lambda x.y)\ (\lambda x.y))$

e. $\lambda y.((\lambda x.x\ y)\ (\lambda x.x\ y))$

2. Consider the following OCaml code:

```
let f x y = x (y x);;
let g x = x + 1;;
let h x = x 1;;
```

a. Write the λ-Calculus expressions that represents the values of variables `f`, `g` and `h` (we'll denote those λ-Calculus expressions as f', g' and h').

b. Write the proof tree demonstrating what the expression $f'\ g'\ h'$ evaluates to. Note: since this proof tree is quite big, feel free to stop once you build it to a point where at least twelwe "$\longrightarrow$" judgments have their right-hand-side filled in.

How to submit: Bring it to class on Friday. If you are unable to come to class, put it into Aleksey Nogin's mailbox on the second floor of Jorgensen.

Part II: OCaml

3. Suppose you have a representation of simple arithmetical expressions defined as follows:

```
type binop =
   Plus
 | Minus
 | Times

type expr =
   Var of string
 | Num of int
 | Binop of expr * binop * expr
```

Write a function `evaluate` of type `(string -> int) -> expr -> int` that computes the value of an expression. The first argument is a function that maps variable names to values of those variables.

4. Define a polymorphic type of binary trees. Write a polymorphic function that folds over a list (similar in style to the polymorphic "sum" function done in class).

How to submit: Write your solutions to problems 3 and 4 into a single text file, make sure it compiles without any warnings (**non-compiling code or code that compiles with warnings will not receive any credit**) and email it to `nogin@cs.caltech.edu`.