HW 4 **Prof. Caldwell**
COSC 4780

Discussion:

We say two lambda terms are α-*equivalent* (read "alpha equivalent") if they have the same shape modulo the names of bound variables. For example:

$$\begin{aligned} \lambda x.x &=_\alpha \lambda y.y \\ \lambda x.\lambda y.xy &=_\alpha \lambda y.\lambda x.yx \\ \lambda x.\lambda y.x(\lambda x.x) &=_\alpha \lambda y.\lambda x.y(\lambda z.z) \end{aligned}$$

The following definition specifies when a pair of terms are α-equivalent:

$$\begin{array}{lcl} x =_\alpha y & \stackrel{\text{def}}{=} & \begin{cases} \mathit{true} & \text{if } x \text{ and } y \text{ are identically the same variable.} \\ \mathit{false} & \text{otherwise} \end{cases} \\ MN =_\alpha M'N' & \stackrel{\text{def}}{=} & M =_\alpha M' \wedge N =_\alpha N' \\ (\lambda x.M) =_\alpha (\lambda y.N) & \stackrel{\text{def}}{=} & M[x := z] =_\alpha N[y := z] \\ & & \quad \text{where } z \text{ is fresh with respect to } fv(M) \text{ and } fv(N). \\ M =_\alpha N & \stackrel{\text{def}}{=} & \mathit{false} \text{ otherwise} \end{array}$$

The first line says that variables are α-equivalent iff they are the same variable. The second line says simply that an application $M\,N$ is α-equivalent iff their respective parts are. The third line says that abstractions are α-equivalent if the the substitution instances of their bodies are, after the free occurrences of the bound variable have been renamed by a *fresh* variable. A variable is fresh if it is not x, is not y, and is not free in either M or N. The renaming is performed using capture-avoiding substitution. The final condition covers all the cases where M and N are lambda terms having different constructors. For example, no variable is α-equivalent to an abstraction or application.

In OCaml, the following codes checks whether two terms are α-equivalent.

```
type lambda = Var of string | Ap of lambda * lambda | Abs of string * lambda ;;
let rec fv m =
      match m with
        Var x -> [x]
      | Ap (m,n) -> fv m @ fv n
      | Abs (x,m) -> List.filter (fun y -> y <> x) (fv m)
                    (* alphaeq : term -> term -> bool *)
let rec alphaeq m n =
  match (m,n) with
      (Var x, Var y) -> x = y
    | (Ap(m,n),Ap(m',n')) -> m = m' && n = n'
    | (Abs(x,m),Abs(y,n)) ->
         let z = fresh ``z'' ([x;y] @ (fv m) @ (fv n)) in
             alphaeq (subst (x,Var z) m) (subst (y,Var z) n)
    | _ -> false    ;;
```

0.1 DeBruijn Indicies

An alternative to checking alpha-equivalence as above is to eliminate bound variables using DeBruijn indices[1] In this representation, α-equivalent terms are syntactically identical.

Example 0.1. Here's a list of lambda terms and their corresponding representations using DeBruijn indices. Note that α-equivalent terms have identical representations ad DeBruijn terms.

lambda term	DeBruijn term
$\lambda x.x$	$\lambda 1$
$\lambda y.y$	$\lambda 1$
$\lambda x.\lambda y.y\,x$	$\lambda\lambda 1\,2$
$\lambda x.\lambda z.z\,x$	$\lambda\lambda 1\,2$
$\lambda x.\lambda y.w\,x\,y$	$\lambda\lambda w\,2\,1$
$\lambda x.\lambda y.w\,x\,x$	$\lambda\lambda w\,2\,2$
$\lambda z.\lambda y.w\,z\,z$	$\lambda\lambda w\,2\,2$
$\lambda x.(\lambda x.x)(\lambda y, x\,y)$	$(\lambda((\lambda 1)(\lambda(21))))$

[1] In the Wikipedia article [https://en.wikipedia.org/wiki/De_Bruijn_index] they start at 1.

To code up an algorithm that translates lambda terms into DeBruijn terms, we'll need to represent DeBruijn terms as an OCaml type. Here's one way

```
type deBruijn =  DBIndex of int
              | DBVar of string
              | DBAp of debruijn * debruijn
              | DBAbs of debruijn
;;
```

We have two kinds of variables now - in a `lambda` term, *free variables* of the form `(Var ''x'')` are represented as `deBruijn` terms as `(dbVar''x'')`. Within a `lambda` term, a *bound variable* of the form `Var ''x''` will become a `deBruijn` term of the form `DBIndex k` where `k` is the number of lambdas up the syntax tree that binds that variable. Note that a `lambda` term of the form `Abs ''x''` M will be translated into a `deBruijn` term of the form `DBAbs` $\hat{M}$ where $\hat{M}$ is the translation of the `lambda` term M into a `deBruijn` term.

Mathematically, we can write the transformation as follows. We use a function f to map variables to their indices. Recall the point-wise update of a function.

```
update f (x, v) = fun y -> if x = y then v else f y;;
```

The idea is to update the function `f` on input `x` to have value `v`.

Example 0.2. If `f x = x + 1` and `f' = update f (2,2)` then `f 2 = 3` and `f' 2 = 2`. In every other case, `f'` behaves like `f`. We have updated the function `f` at the input (point) `2`.

We'll use a function to keep track of DeBruijn indices. When a bound variable is encountered it will be mapped to a deBruijn (term) with the

We use a similar idea for keeping track of, and updating DeBruijn indices. The update function will map functions from strings to deBruijn (terms) to new functions of the same type. If the string is we need to add a new index and update the indexes of all the others to add one. The value returned by the updated function is determined by the values of the function being updated!

```
dbUpdate :: (string → deBruijn) → string → (string → deBruijn)
let dbUpdate f x =
```

```
            fun y -> if x = y then
                         dBIndex 1
                     else
                         match (f y) with
                           (dBIndex k) -> (dBIndex (k+1)
                          | _ -> dBVar y
;;
```

The `dbUpdate` function really will turn out to do all the work of keeping track of the indices.

Here's a definition of the transformation ($\mapsto_f$), that transforms `lambda` terms into `deBruijn` terms. Note that f is a parameter to the transformation and intially $fx =$ `DBVar` x - i.e. it maps all strings to a `DBVar`.

$$
\begin{array}{lll}
Var\,x & \mapsto_f & f\,x \\
Ap(m,n) & \mapsto_f & dbAp\,(m',n') \quad \text{where } m \mapsto_f m' \wedge n \mapsto_f n' \\
Abs(x,m) & \mapsto_f & dBAbs\,m' \quad\;\; \text{where } m \;\mapsto_{f'}\; m' \text{ and } f' = \text{dbUpdate}\;fx
\end{array}
$$

The transformation uses the function f to figure out how variables get mapped – to DeBruijn indices or just to variables. Initially f is the function that maps a string x to the DeBruijn `DBVar x`. When it encounters an abstraction of the form $\lambda x.m$ (`ABs(x,m)`), it has to update f so that the free occurrences of x in M get mapped to the DeBruijn index `DBIndex 1`. If another abstraction is encountered, f will be updated again to map x to index 1, and all the other indices will be incremented to indicate another depth of abstraction has been encountered. This worked, even if f is updated for x a second (or third, or fourth) time.

Problem 0.1. Implement a function `debruijnize` which maps lambda terms to their equivalent DeBruijn representation.

Problem 0.2. Write a program that uses the DeBruijn representation to check for α-equivalence and provide some test cases to convince the grader your code works.