CS 441 Programming Languages
Assignment #4

Due: Friday, 17 October 2008 (by class time)

DeBruijn indices are a way of representing bound variables conveniently and efficiently. Instead of using variable names (strings), one uses integer indices. DeBruijn indices uniquely identify the binding occurrence of a variable. In general, the index $[n]$ refers to the value bound by the nth binding expression counting outward from the occurrence. (We'll start the count at 1.). For example, consider the DeBruijn expression $(\lambda.\lambda.([2]\ [1]))$. In this expression, each λ introduces a variable (but doesn't it give it a name – the variable will be referred to using indices). The body of the inner function is the expression $([2]\ [1]))$. This expression is an application of the variable named [2] to the variable named [1]. The [2] refers to the variable bound 2 steps outside the current expression (ie: the outer λ) and the variable $[a]$ refers to the variable bound 1 step outside the current expression (ie: the inner λ). In our normal syntax, the expression $(\lambda.\lambda.([2]\ [1]))$ is equivalent to $(\lambda x.\lambda y.(x\ y))$. Here are a couple of other examples:

Reference Number	Representation Using Variables:	Representation Using DeBruijn:
1.	$\lambda x.x$	$\lambda.[1]$
2.	$\lambda x.\lambda x.x\ x$	$\lambda.\lambda.[1]\ [1]$
3.	$(\lambda x.x\ x)\ (\lambda x.x\ x)$	$(\lambda.[1]\ [1])\ (\lambda.[1]\ [1])$
4.	$(\lambda z.z\ z)\ (\lambda y.y\ y)$	$(\lambda.[1]\ [1])\ (\lambda.[1]\ [1])$
5.	$\lambda x.\lambda y.(x\ (\lambda x.x\ x))$	$\lambda.\lambda.([2]\ (\lambda.[1]\ [1]))$
6.	$\lambda x.x\ \lambda y.y\ x$	$\lambda.[1]\ \lambda.[1]\ [2]$

Notice that in example 6, the same variable x is referred to by different indices in the DeBruijn representation (once by [1] and once by [2]). Also in example 6, the same index [1] is used to refer to two different variables, once it corresponds to x and once it corresponds to y!

In general, we'll write DeBruijn expressions using the following notation:

$$\begin{array}{rrcl} \text{Index} & n & ::= & 1 \mid 2 \mid ... \\ \text{Expression} & e & ::= & [n] \mid \lambda.e \mid e_1\ e_2 \end{array}$$

As the examples above suggest, it is possible to convert back and forth between DeBruijn representations and normal variable representations. To do so, we'll need to use an *environment* that associates variable names with indices. We'll define the environment like this:

$$\text{Environment} \qquad E \ ::= \ \texttt{nil} \mid E, x = i$$

We'll also need a function to increment all the indices in an environment and to lookup variables in an environment. I'll define the judgements $\vdash \texttt{incr}\ E_1\ E_2$ and $\vdash \texttt{lookup}\ E_1\ x\ n$ for doing this. Note, instead

of working with constructors S and Z as we have before for natural numbers, I'll work with the numbers 1, 2, 3, etc that we are all familiar with and freely use operations like addition, knowing we could define them from first principles if we really wanted to.

$$\frac{}{\vdash \texttt{incr nil nil}} \qquad \frac{\vdash \texttt{incr } E_1\ E_2}{\vdash \texttt{incr } (E_1, x = n)\ (E_2, x = (n+1))}$$

$$\frac{}{\vdash \texttt{lookup } (E, x = n)\ x\ n} \qquad \frac{\vdash \texttt{lookup } E\ y\ n \qquad (x \neq y)}{\vdash \texttt{lookup } (E, x = n')\ y\ n}$$

Now we'll define the judgement for converting normal expressions into DeBruijn expressions. It has the form "$E \vdash e_{var} = e_{db}$". You can read this as saying "In environment E, expression with variables e_{var} can be converted into DeBruijn expression e_{db} and vice versa."

$$\frac{\vdash \texttt{lookup } E\ x\ n}{E \vdash x = [n]}$$

$$\frac{E \vdash e_1 = e_1' \qquad E \vdash e_2 = e_2'}{E \vdash e_1 e_2 = e_1' e_2'}$$

$$\frac{\vdash \texttt{incr } EE' \qquad E', x = 1 \vdash e = e'}{E \vdash \lambda x.e = \lambda.e'}$$

Q. 1[3 points] Defining Free Variables A *free variable* is a DeBruijn variable $[k]$ that is nested inside fewer than k lambdas. For example, we have marked with a $*$ the free variables in the following expression:

$$\lambda.\lambda.([2]\ [3]^*\ (\lambda.[3]\ [4]^*))$$

Notice also that if we wrap one more lambda around the entire expression, no variables are free. In other words, the following has no free variables:

$$\lambda.\lambda.\lambda.([2]\ [3]\ (\lambda.[3]\ [4]))$$

Consider again the expression $\lambda.\lambda.([2]\ [3]^*\ (\lambda.[3]\ [4]^*))$. In this expression, $[3]^*$ is under 2 lambdas and computing $3-2$ gives us 1. Similarly, $[4]^*$ is under 3 lambdas and computing $4-3$ is also 1. Consequently, $[3]^*$ and $[4]^*$ refer to the *same* free variable. If we wanted to refer to that variable outside of any λ expression, we'd call it the variable $[1]$. With this information, define a function FV that computes the set of free DeBruijn variables in an expression e. It should be the case that:

$$FV(\lambda.\lambda.([2]\ [3]^*\ (\lambda.[3]\ [4]^*))) = \{[1]\}$$

You may use a function *incr* that increments all variables in a set by 1 and a function *decr* that decrements all variables in a set by 1. For your reference, the definition of free variables for ordinary lambda terms can be found both in the lecture slides and in your textbook (definition 5.3.2).

Q. 2[3 points] Definition of Substitution for DeBruijn Give a definition of capture-avoiding substitution for DeBruijn expressions. In other words, define $e_1[e_2/i]$, the capture-avoiding substitution of e_2 for free variable $[i]$ in e_1. The definition should have a similar form to the definition given in the slides in class (in other words, define a function with 1 case for each different kind of DeBruijn expression). To facilitate the definition, you may define one or more auxiliary functions. Where appropriate, define your functions by induction over the structure of expressions (ie: one case per different sort of expression). Here are some examples:

Number	Substitution:	Result:
1.	$(\lambda.[1]\ [2])[\lambda.[1]/[1]]$	$(\lambda.[1]\ \lambda.[1])$
2.	$(\lambda.[1]\ [2])[[3]/[1]]$	$(\lambda.[1]\ [4])$
3.	$(\lambda.[1]\ [1])[[3]/[1]]$	$(\lambda.[1]\ [1])$

Q. 3[6 points] Implement and test the following functions in ML by downloading and modifying the file `http://www.cs.princeton.edu/courses/archive/fall08/cos441/assignments/a4code/db.sml`.

In the DB structure:

```
(* subst e1 i e2 = e1[e2/i] *)
fun subst : exp -> var -> exp -> exp
(* if e1 --> e2 then step e1 = e2 *)
fun step : exp -> exp
```

In the Lam structure:

```
(* Convert from Variables to DeBruijn Indices *)
fun var2db : exp -> DB.exp
(* Convert from DeBruijn Indices to Variables *)
fun db2var : DB.exp -> exp
```

Once you've implemented these functions, you can take a look at how they all work together using the function `step` that I've defined for you in the Lam module.

Q. 4[2 points] Recall that any single-step operational relation (including the single step call-by-value relation on DeBruijn terms) with the form $e_1 \longrightarrow e_2$ can be extended to a multi-step relation using the following two rules:

$$\frac{}{e \longrightarrow^* e} \text{ (reflexivity)} \qquad \frac{e_1 \longrightarrow e_2 \qquad e_2 \longrightarrow^* e_3}{e_1 \longrightarrow^* e_3} \text{ (transitivity)}$$

Prove that the multi-step relation itself is transitive. In other words prove that if $e_1 \longrightarrow^* e_2$ and $e_2 \longrightarrow^* e_3$ then $e_1 \longrightarrow^* e_3$.

Q. 5[1 points] Consider the following slightly different definition of multistep relations:

$$\frac{}{e \Longrightarrow^* e} \text{ (reflexivity2)} \qquad \frac{e_1 \Longrightarrow^* e_2 \qquad e_2 \longrightarrow e_3}{e_1 \Longrightarrow^* e_3} \text{ (transitivity2)}$$

Consider this theorem (identical to the one you proved in question 4): if $e_1 \Longrightarrow^* e_2$ and $e_2 \Longrightarrow^* e_3$ then $e_1 \Longrightarrow^* e_3$. In one sentence, what is the key difference between the proof you need to do for this theorem vs. the proof you did in question 4? In order to answer this question correctly, you probably need to do the proof. However, don't hand it in. Just explain what the key difference between the two proofs is.