CS 410/510 Compiling Advanced Languages Homework 1 – due Friday, April 7, 2000

Pencil and Paper

1. Do Field & Harrison Exercises 6.1, 6.2(ii), 6.5(a,c), 6.7a, 6.8. Draw trees where appropriate!

2. Define an addition function for the Church numerals, as described in class.

Programming

These problems refer to the file `lambda.ml` in the course directory. To load and compile the declarations in this file, start `ocaml` and execute "`#use "lambda.ml";;`". Make sure you can execute these functions before you continue!

3. (a) Write a function `cbv` that attempts to evaluate an `exp` to normal form using call-by-value.

(b) Write a similar function `cbn` that attempts to evaluate an `exp` to normal form using call-by-name.

Hints:

- Examine the function `reduce` to get the general idea of doing case analysis on the various forms of `exp`.
- There are several ways to go about writing `cbv`. The most straightforward way is probably to use an auxiliary recursive function

  ```
  left_inner : exp -> exp option
  ```

  (where

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

  is a standard type defined in the CAML pervasive library). The idea is that (`left_inner` e) attempts to perform a single leftmost innermost transformation on e. If it succeeds with result e', it returns `Some` e'; if it fails, it returns `None`. The `cbv` function itself then calls `left_inner` repeatedly until no further transformation can be made.
- Similar ideas apply to `cbn`.
- Another approach, trickier but less messy and tedious, is to write a directly recursive version of `cbv` (or `cbn`) in which each recursive call returns only when it has performed all possible cbv (or cbn) reductions on its argument.
- If you're having trouble distinguishing the behavior of your functions, consider the expression `App(zero,dd)`. Applying `cbn` to this expression should result in the normal form `Abs("x",Var "x")`, but applying `cbv` to it should result in an infinite loop.

5. Extend `lambda.ml` to support built-in integer constants and simple arithmetic operations on them. Do this by adding new `exp` constructors `Const of int` and `Plus`, and extending all the functions to operate correctly on these constructors.

6. The following datatype is suitable for describing pure λ-expressions using de Bruijn notation:

```
type dexp = Dabs of dexp | Dapp of dexp * dexp | Dvar of int
```

Write a function `convert: exp -> dexp` that converts *closed* λ-expressions from `exp` format to `dexp` format by computing the correct de Bruijn indices. For example, the output of `(convert one)` should equal `Dabs(Dabs(Dapp(Dvar 1,Dvar 0)))`. Remember to handle rebinding of names in inner scopes correctly. Do something sensible if your function is handed a non-closed argument. Hint: Your function can use a list to maintain an environment of bound variable names as it traverses the `exp`; the position of a variable in this list is closely connected with its de Bruijn index.

Extra Credit (for fun)

Here is a function $\overline{\texttt{pred}}$ that calculates the predecessor of a Church numeral (where $\overline{\texttt{succ}}$, $\overline{\texttt{true}}$ and $\overline{\texttt{false}}$ are as defined in class).

$\lambda\texttt{k.(k(}\lambda\texttt{p.}\lambda\texttt{u.u(}\overline{\texttt{succ}}\texttt{(p }\overline{\texttt{true}}\texttt{))(p }\overline{\texttt{true}}\texttt{)) (}\lambda\texttt{u.u }\overline{\texttt{0}}\texttt{ }\overline{\texttt{0}}\texttt{))}\overline{\texttt{false}}\texttt{.}$

(a) Draw the λ-tree (parse tree) of this function.

(b) Explain, in English, how it works. (Hint: Function names can be misleading.)