Computer Science 131, Spring 2002
Assignment 12: More λ-Calculus

Out: Wednesday, April 7

Due: Monday, April 22, at the beginning of class

Your code should be placed in a file named `assign12.sml` and handed in using `cs131submit`.

1 The *Unityped* λ-Calculus (40%)

Although historically the untyped λ-calulus was studied before the typed λ-calculus, it turns out that the untyped case can be thought of as a very special case of the typed case in which there is exactly one type, where everything has this type.

Making the correspondence exact would require a type `D` satisfying `D = (D -> D)`. However, for mimicking the untyped lambda calculus it suffices to find a type `D` such that `D` and `D -> D` are *isomorphic*, so that all values of type `D` are also functions of type `D->D` and all such functions are also values of type `D`. We can arrange this implicitly by having a language in which `D->D` $\preceq$ `D` and `D` $\preceq$ `D->D`, or more explicitly by specifying functions which can convert any value of type `D` into a value of type `D->D`, and vice versa. Historically such functions are called Φ and Ψ, where

$$\Phi \texttt{ : D -> (D -> D)}$$
$$\Psi \texttt{ : (D -> D) -> D}$$

and $\Psi \circ \Phi$ is the identity on `D` and $\Phi \circ \Psi$ is the identity on `D -> D`.

Given such a type, we can then translate every untyped λ-calculus term into a term of type `D`. We write $|M|$ to represent the translation of the term M.

$$\begin{aligned} |x| &= x \\ |\lambda x.M| &= \Psi(\lambda x{:}\texttt{D}.|M|) \\ |M\,N| &= (\Phi|M|)|N| \end{aligned}$$

We can define such a type and the required functions in SML using the following code:

```
datatype D = Psi of D -> D
val Phi : D->(D->D)  =  (fn (Psi f) => f)
```

Here `Psi : (D->D)->D` puts a tag onto a `D->D` function, and `Phi : D->(D->D)` strips the tag off to get the underlying function.

1. Convince yourself that any closed λ-term can be translated into an SML expression of type `D`. For example, according to the translation above the identity function $\lambda x.x$ is be represented as `Psi(fn x:D => x)`. (Nothing need be turned in for this part.) Note that evaluating an ML expression of type `D` that represents a λ-term corresponds to call-by-value evalution of that lambda term, stopping if it reduces to a function.

2. Define a term `one : D` which is the translation of the Church numeral one, $\lambda f.\lambda b.(fb)$

3. Complete the definition of the following function

```
val loop = (fn () => ...)
```

   where the call `loop()` does not terminate. The catch is that you may not use any of the following:

   - side-effects such as assignment or exceptions or continuations
   - defining recursive functions using `fun` or `val rec`
   - functions from the built-in basis library.
   - `while` or other iterative constructs.

2 Curry-Howard (60%)

For each of the following propositions, give as a comment the corresponding type in SML according to the Curry-Howard isomorphism. Give an SML value of this type (showing that the proposition is provable). You may use the definitions

```
datatype void = VOID of void
fun anything (VOID v) = anything v
```

to define a type containing no values (which corresponds to the false proposition), and a function whose type `void -> 'a` corresponds to the proposition that anything follows from falsehood. (You may not need to use this function.)

Your definitions should not otherwise use recursion or looping in any form (or side-effects such as exceptions) as in general these additions to the language correspond to inconsistent logics. The term for part 1 (if any) should be called `m1`, the term for part 2 (if any) should be called `m2`, and so on. You should make your terms polymorphic, with type variables (such as `'a` or `'b`) corresponding to the propositional variables (such as `p` or `q`). So, for example, if the proposition were $p \Rightarrow \neg\neg p$, you would supply an SML value of type `'a -> (('a -> void) -> void)` such as `fn x:'a => (fn f:'a->void => f x)`.

1. $(p \Rightarrow q \Rightarrow r) \Rightarrow q \Rightarrow p \Rightarrow r$. (Recall that implication associates to the right.)

2. $\neg(p \wedge \neg p)$

3. $(p \wedge q) \Rightarrow \neg(p \Rightarrow \neg q)$

4. $(p \Rightarrow q) \Rightarrow (\neg q \Rightarrow \neg p)$

5. **[10% extra credit; incredibly hard]** $\neg\neg(\neg\neg p \Rightarrow p)$
   Hint: $\neg(\neg\neg p \Rightarrow p) \Rightarrow \neg p$ and $\neg p \Rightarrow (\neg\neg p \Rightarrow p)$.