Computer Science 131, Fall 2000
Assignment 8: Implementing Theory

Out: Tuesday, November 28

Due: Tuesday, December 5, 2:45pm (Note the time!)

To complete this assignment, retrieve the files `assign8.sml` from the Assignments web page. No special version of the compiler is required.

Your task is to complete the functions in the `assign8.sml` file and turn this in. *As always, your file should contain no syntax or type errors.* The submission process is the same as usual; when you are ready, run the command

```
cs131submit assign8.sml
```

1 Combinatory Logic (60%)

1. **Translation from Lambda Terms**

   The `assign8.sml` file contains the following code for representing λ-terms and terms in combinatory logic:

```
type varname = string              (* variables are strings again *)
datatype lam = Var of varname
             | Lam of varname * lam    (* arg. variable and body *)
             | App of lam * lam


datatype cl  = CLVar of varname
             | S
             | K
             | I                    (* an extra constant *)
             | B                    (* an extra constant *)
             | C                    (* an extra constant *)
             | CLApp of cl * cl
```

   The use of these representations should be fairly obvious. For example, the λ-term $\lambda x.\lambda y.(xy)$ would be represented as `Lam("x",Lam("y",App(Var "x",Var "y")))` and the CL-term $\mathsf{S}x\mathsf{K}$ would be represented as `CLApp(CLApp(S,CLVar "x"), K)`. The language of combinatory logic has also been extended with the three constants I, B, and C, which can be ignored until Part 2.

In class, you saw the following definition of bracket abstraction and the translation of λ terms into CL-terms:

$$
\begin{array}{lcll}
[x]\,\mathsf{K} & = & \mathsf{KK} & \\
[x]\,\mathsf{S} & = & \mathsf{KS} & \\
[x]\,x & = & \mathsf{SKK} & \\
[x]\,y & = & \mathsf{K}y & (\text{if } x \neq y) \\
[x]\,ab & = & \mathsf{S}([x]\,a)([x]\,b) & \\
\\
\mathrm{CL}(x) & = & x & \\
\mathrm{CL}(MN) & = & \mathrm{CL}(M)\,\mathrm{CL}(N) & \\
\mathrm{CL}(\lambda x.M) & = & [x]\,\mathrm{CL}(M) &
\end{array}
$$

Write the functions

```
bracket0 : varname * cl -> cl
toCL0    : lam -> cl
```

where `bracket0 ("x",`a`)` returns the combinatory logic term representing of $[x]\,a$ and `toCL0` M returns the combinatory logic term corresponding to the lambda term M.

2. **A Better Translation** Although it is possible to represent every lambda term using only S and K, but the above translation can create extremely large combinators. We can do much better using new constants I, B, and C with the following behavior:

$$
\begin{array}{lll}
\mathsf{I}\,u & \rightarrow_{\mathrm{CL}} & u \\
\mathsf{B}\,u_1\,u_2\,u_3 & \rightarrow_{\mathrm{CL}} & u_1\,(u_2\,u_3) \\
\mathsf{C}\,u_1\,u_2\,u_3 & \rightarrow_{\mathrm{CL}} & u_1\,u_3\,u_2
\end{array}
$$

Define a function `optimize` such that

$$
\begin{array}{lcl}
\mathtt{optimize}(\mathsf{S}\,(\mathsf{K}\,p)\,(\mathsf{K}\,q)) & = & \mathsf{K}\,(p\,q) \\
\mathtt{optimize}(\mathsf{S}\,(\mathsf{K}\,p)\,\mathsf{I}) & = & p \\
\mathtt{optimize}(\mathsf{S}\,(\mathsf{K}\,p)\,q) & = & \mathsf{B}\,p\,q \\
\mathtt{optimize}(\mathsf{S}\,p\,(\mathsf{K}\,q)) & = & \mathsf{C}\,p\,q \\
\mathtt{optimize}(p) & = & p
\end{array}
$$

where, as in SML, the intent is that you take the first case that matches. Convince yourself that optimize preserves the meaning of combinatory terms in the following sense: in all of the cases when you apply the both sides to the same argument, both applications reduce to the same result.

Then define $[x]\,u_1\,u_2$ to be $\mathtt{optimize}(\mathsf{S}\,([x]\,u_1)\,([x]\,u_2))$, and $[x]\,x$ to be I.

Write the functions

```
bracket : varname * cl -> cl
toCL    : lam -> cl
```

That implement both of these optimizations.

Compare the `fromLambda0` and `fromLambda` translations for a few lambda terms — the difference can be impressive. The functions

```
pplam : lam -> string
ppcl  : cl -> string
```

provided convert lambda terms and combinatory logic terms into a string which can be passed to `print`. The `assign8.sml` file also contains several pre-defined lambda terms for you to play with.

3. The call-by-name operational semantics for this combinatory logic can be given as follows:

$$
\begin{array}{ll}
\mathsf{K}\, a_1\, a_2 & \rightarrow_{\mathrm{CL}} \; a_1 \\
\mathsf{S}\, a_1\, a_2\, a_3 & \rightarrow_{\mathrm{CL}} \; (a_1\, a_3)\, (a_2\, a_3) \\
\mathsf{I}\, u & \rightarrow_{\mathrm{CL}} \; u \\
\mathsf{B}\, a_1\, a_2\, a_3 & \rightarrow_{\mathrm{CL}} \; a_1\, (a_2\, a_3) \\
\mathsf{C}\, a_1\, a_2\, a_3 & \rightarrow_{\mathrm{CL}} \; a_1\, a_3\, a_2 \\
\multicolumn{2}{l}{\dfrac{a_1 \rightarrow_{\mathrm{CL}} a_1'}{a_1\, a_2 \rightarrow_{\mathrm{CL}} a_1'\, a_2}}
\end{array}
$$

Values would be anything which can't be reduced *at the top-level*. (For example, $\mathsf{K}\,\mathsf{S}$ or $\mathsf{C}\,(\mathsf{I}\,\mathsf{B})\,(\mathsf{I}\,B)$).

Here is an alternative presentation of this same call-by-name semantics:

$$
\begin{array}{lll}
\mathsf{K}\, a_1\, a_2\, b_1\, \cdots\, b_n & \rightarrow_{\mathrm{CL}} & a_1\, b_1\, \cdots\, b_n \quad (n \geq 0) \\
\mathsf{S}\, a_1\, a_2\, a_3\, b_1\, \cdots\, b_n & \rightarrow_{\mathrm{CL}} & (a_1\, a_3)\, (a_2\, a_3)\, b_1\, \cdots\, b_n \quad (n \geq 0) \\
\mathsf{I}\, u\, b_1\, \cdots\, b_n & \rightarrow_{\mathrm{CL}} & u\, b_1\, \cdots\, b_n \quad (n \geq 0) \\
\mathsf{B}\, a_1\, a_2\, a_3\, b_1\, \cdots\, b_n & \rightarrow_{\mathrm{CL}} & a_1\, (a_2\, a_3)\, b_1\, \cdots\, b_n \quad (n \geq 0) \\
\mathsf{C}\, a_1\, a_2\, a_3\, b_1\, \cdots\, b_n & \rightarrow_{\mathrm{CL}} & a_1\, a_3\, a_2\, b_1\, \cdots\, b_n \quad (n \geq 0)
\end{array}
$$

Convince yourself that this gives the same relation as the previous definition.

You have been given part of an implementation for

```
cbn : cl -> cl
```

that evaluates closed combinatory logic terms based on this second presentation. Finish this implementation.

2 The *Unityped* λ-Calculus (20%)

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.

Making the correspondence exact would require a type `D` satisfying `D` = `(D -> D)`. However, it suffices to find a type `D` such that `D` and `D -> D` are *isomorphic*. That is, it suffices to find two functions

$$\begin{array}{l} \Phi \; : \; \texttt{D -> (D -> D)} \\ \Psi \; : \; \texttt{(D -> D) -> D} \end{array}$$

such that $\Psi \circ \Phi$ 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.

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

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`. (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 $\overline{1} = \lambda b.\lambda f.f(b)$

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.

3 Curry-Howard (20%)

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.

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. **[20% extra credit]** $\neg\neg(\neg\neg p \Rightarrow p)$