Assignment 3
Introduction to Computational Logic, SS 2006

Prof. Dr. Gert Smolka, Dipl.-Inform. Mathias Möhl
http://www.ps.uni-sb.de/courses/cl-ss06/

We continue with the implementation of terms in Standard ML. As before, we omit types and constants and represent terms as follows:

```
type index = int  (* non-negative *)
datatype ter = I of index | A of ter * ter | L of ter
```

From last week, you know the procedures

$free : index \to ter \to bool$

$subst : (index \to ter) \to ter \to ter$

They will be useful in the following exercises.

Exercise 3.1 (Concrete Syntax) Write a procedure $lam : ter\ list \to ter\ list \to ter$ that implements a concrete syntax for terms such that, for instance, the term $\lambda xy.fx(fxy)$ can be obtained as follows:

```
val x = I 0
val y = I 1
val f = I 2
val t = lam[x,y][f,x,lam[][f,x,y]]
```

Exercise 3.2 (Reduction) Complete the following declaration such that it declares a procedure $red : ter \to ter$ that yields the $\beta\eta$-normal form of a term.

```
fun red (I x) = I x
  | red (A(s,t)) = beta (red s) (red t)
  | red (L t) = eta (red t)
and beta (L s) t = ··· apply β ···
  | beta s t = A(s,t)
and eta (A(t, I 0)) = ··· try η ···
  | eta t = L t
```

Test your code with the examples from exercise 2.4 and the procedure *lam* from Exercise 3.1.

Exercise 3.3 (Elimination) Complete the following declarations such that they declare a procedure *elim* : *int* → *ter* → *ter* such that *elim 0 t* yields a combinatory term equivalent to t. Use the following equivalences to eliminate abstractions:

$$\begin{aligned} \lambda x.fx &\equiv f && \eta \\ \lambda x.x &\equiv I && I \\ \lambda x.y &\equiv Ky && K \\ \lambda x.st &\equiv S(\lambda x.s)(\lambda x.t) && S \end{aligned}$$

Represent the constants I, K, and S with the variables 991, 992, and 993.

```
exception Error of string
fun id d = I(991+d)
fun kon d = I(992+d)
fun sch d = I(993+d)
fun lift s = subst (fn x => I(x-1)) s
fun elim1 d s = if free 0 s then elim2 d s else ··· apply K ···
and elim2 d (I 0) = ··· apply I ···
  | elim2 d (s as A(t, I 0)) = ··· try η ···
  | elim2 d s = elim3 d s
and elim3 d (A(s,t)) = ··· apply S ···
  | elim3 d _ = raise Error "elim3"
fun elim d (I x) = I x
  | elim d (A(s,t)) = A(elim d s, elim d t)
  | elim d (L s) = elim1 d (elim (d+1) s)
```

Make sure your code passes the following test:

```
elim 0 (lam[x][f,x,lam[][f,x,x]])
A(A(I 993, I 3), A(A(I 993, I 3), I 991)) : ter
```

Exercise 3.4 (Type Checking) We now implement terms with types:

```
datatype ty = C of string | F of ty * ty
datatype ter = I of index | A of ter * ter | L of ty * ter
```

Write a procedure *ty* : *ty list* → *ter* → *ty* such that *ty nil t* returns the type of t if t is a well-formed and closed term. If t is not closed or is not well-formed, an exception should be raised. The first argument of *ty* should act as a stack on which the argument types of the abstractions are pushed.

Exercise 3.5 (Challenge: Advanced Elimination) The elimination as implemented in Exercise 3.3 does no β reduction. Sometimes applying the β rule results in much smaller terms. The computed combinatory term for $(\lambda fx.fy)(\lambda x.x)$ (see Exercise 2.5 e)) is for example $S(KK)(SI(Ky))I$ instead of Ky which is obtained by applying β.

Modify the implementation of Exercise 3.3 such that β reduction is done where it seems to simplify the result.