1 Hacking with the untyped call-by-value lambda calculus

In this exercise, you have to implement some operations for Church encoding of lists. There are several ways to Church encode a list, among which Church encoding based on its right fold function is more popular. As an example, an empty list (`nil`) and the `cons` construct are represented as follows in this encoding:

```
nil = λc.  λn.  n
cons = λh.  λt.  λc.  λn.  c h (t c n)
```

As another example, a list of 3 elements x, y, z is encoded as:

```
λc.  λn.  c x (c y (c z n))
```

The complete list of predefined operations can be found in the appendix, and only these operations can be used in the exercise. Define the following operations on a list:

1. (2 points) The *map* function which applies the given function to each element of the given list.

2. (2 points) The *length* function which returns the size of the given list. The result should be in Church encoding.

3. (2 points) The *sum* function which returns the sum of all elements of the given list. Assume all elements and the result are Church encoded numbers.

4. (2 points) The *concat* function which concatenates two input lists.

5. (2 points) The *exists* function which checks if there is any element satisfying the given predicate. The given predicate and the result should be both in Church encoding.

2 Simply typed SKI combinators

In this exercise we're going to explore an alternative calculus called SKI that's based on three combinators: S, K and I instead of lambda abstraction. Those combinators can be translated into STLC as derived forms:

$$I[T] = \lambda x : T.\ x \tag{D-I}$$

$$K[T, U] = \lambda x : T.\ \lambda y : U.\ x \tag{D-K}$$

$$S[T, U, W] = \lambda x : T \to U \to W.\ \lambda y : T \to U.\ \lambda z : T.\ xz(yz) \tag{D-S}$$

An interesting aspect of SKI is that those combinators are sufficient to exclude lambda abstraction from the language without loss of expressiveness. More concretely the system has the following syntax:

$$
\begin{array}{llll}
t & ::= & & \textbf{terms}: \\
 & | & I[T] & I \text{ combinator} \\
 & | & K[T, U] & K \text{ combinator} \\
 & | & S[T, U, W] & S \text{ combinator} \\
 & | & t\ t & \text{Application} \\
u, v, w & ::= & & \textbf{values}: \\
 & | & I[T] & I \text{ combinator} \\
 & | & K[T, U] & K \text{ combinator (1)} \\
 & | & K[T, U]\ v & K \text{ combinator (2)} \\
 & | & S[T, U, W] & S \text{ combinator (1)} \\
 & | & S[T, U, W]\ v & S \text{ combinator (2)} \\
 & | & S[T, U, W]\ v\ v & S \text{ combinator (3)} \\
T, U, V, W & ::= & & \textbf{types}: \\
 & | & T \to T & \text{Function type}
\end{array}
$$

Values in this language are the aforementioned combinators as well as their partially applied versions.

Questions:

- Provide small-step reduction rules assuming call-by-value evaluation semantics (4 points).
- Provide typing rules $\Gamma \vdash t : T$ and prove the *preservation* property (6 points).

Note: There is no lambda abstraction in the language any longer. You may not use it as a means to express typing or evaluation rules.

3 Appendix

3.1 The call-by-value simply typed lambda calculus

The complete reference of the variant of simply typed lambda calculus (with *Bool* ground type representing the type of values *true* and *false*) used in "The call-by-value simply typed lambda calculus with returns" is as follows:

$$
\begin{array}{ll}
v ::= \lambda x\colon T.\ t \mid bv & (\textit{values}) \\
bv ::= \texttt{true} \mid \texttt{false} & (\textit{boolean values}) \\
t ::= x \mid v \mid t\ t & (\textit{terms}) \\
p ::= t & (\textit{programs}) \\
T ::= \texttt{Bool} \mid T \to T & (\textit{types})
\end{array}
$$

Evaluation rules:

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \tag{E-App1}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \tag{E-App2}$$

$$(\lambda x\colon T_1.\ t_1)\ v_2 \longrightarrow [x \mapsto v_2]t_1 \tag{E-AppAbs}$$

Typing rules:

$$\frac{x\colon T \in \Gamma}{\Gamma \vdash x\colon T} \tag{T-Var}$$

$$\frac{\Gamma,\ x\colon T_1 \vdash t_2\ :\ T_2}{\Gamma \vdash (\lambda x\colon T_1.\ t_2)\ :\ T_1 \to T_2} \tag{T-Abs}$$

$$\frac{\Gamma \vdash t_1\ :\ T_1 \to T_2 \quad \Gamma \vdash t_2\ :\ T_1}{\Gamma \vdash t_1\ t_2\ :\ T_2} \tag{T-App}$$

$$\frac{}{\Gamma \vdash \texttt{true}\ :\ \texttt{Bool}} \tag{T-False}$$

$$\frac{}{\Gamma \vdash \texttt{false}\ :\ \texttt{Bool}} \tag{T-True}$$