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:

t	::=		**terms** :
	|	$I[T]$	I combinator
	|	$K[T, U]$	K combinator
	|	$S[T, U, W]$	S combinator
	|	$t\ t$	Application
v	::=		**values** :
	|	$I[T]$	I combinator
	|	$K[T, U]$	K combinator (1)
	|	$K[T, U]\ v$	K combinator (2)
	|	$S[T, U, W]$	S combinator (1)
	|	$S[T, U, W]\ v$	S combinator (2)
	|	$S[T, U, W]\ v\ v$	S combinator (3)
T, U, W	::=		**types** :
	|	$T \to T$	Function type

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 Checked Error Handling

In this exercise we use the Simply-Typed Lambda Calculus (STLC) extended with rules for error handling. In this language, terms may reduce to a normal form `error`, which is *not* a value. In addition, we add the new term form `try` t_1 `with` t_2, which allows handling errors that occur while evaluating t_1.

Here is a summary of the extensions to syntax and evaluation:

			terms :
t	::=		
	$\mid$	$\ldots$	
	$\mid$	`error`	run-time error
	$\mid$	`try` t `with` t	trap errors

New evaluation rules:

(E-AppErr1) `error` $t_2 \longrightarrow$ `error`

(E-AppErr2) v_1 `error` $\longrightarrow$ `error`

(E-TryValue) `try` v_1 `with` $t_2 \longrightarrow v_1$

(E-TryError) `try error with` $t_2 \longrightarrow t_2$

$$\text{(E-Try)}\ \frac{t_1 \longrightarrow t_1'}{\texttt{try}\ t_1\ \texttt{with}\ t_2 \longrightarrow \texttt{try}\ t_1'\ \texttt{with}\ t_2}$$

(Note that these extensions are exactly those summarized in Figures 14-1 and 14-2 on pages 172 and 174 of the TAPL book. However, also note that we will use *different* typing rules.)

The goal of this exercise is to define typing rules for STLC with the above extensions such that the following progress theorem holds:

If $\emptyset$; `false` $\vdash t : T$, then either t is a value or else $t \longrightarrow t'$.

The above theorem uses a typing judgment extended with a Boolean value E, written Γ ; $E \vdash t : T$ where $E \in \{\texttt{true}, \texttt{false}\}$. The theorem says that a well-typed term that is closed (that is, it does not have free variables, which is expressed using $\Gamma = \emptyset$) is either a value, or else it can be reduced *as long as* $E =$ `false`.

Your task is to find out how the value of E can be used to distinguish the terms that may reduce to `error` from those terms that may never reduce to `error`. Note that `error` is a normal form, but it is *not* a value.

1. Specify typing rules of the form Γ ; $E \vdash t : T$ for all term forms of STLC with the above extensions such that the above progress theorem holds.

2. Prove the above progress theorem using structural induction. (You can use the canonical forms lemma for STLC as seen in the lecture without proof.)

4 The call-by-value simply typed lambda calculus with returns

Consider a variant of the call-by-value simply typed lambda calculus specified in the appendix extended to support a new language construct: `return` t, which immediately returns a given term t from an **enclosing** lambda, disregarding any potential further computation typically needed for call-by-value evaluation rules.

The grammar of the extension is defined as follows. We distinguish top-level terms (tt) and nested terms (nt) to make sure that `return` t can only appear inside lambdas:

v	::=	$\lambda x :$ `T` . $nt \mid bv$	$(values)$
bv	::=	`true` | `false`	$(boolean\ values)$
nt	::=	$x \mid v \mid nt\ nt \mid$ `return` nt	$(nested\ terms)$
tt	::=	$x \mid v \mid tt\ tt$	$(top-level\ terms)$
t	::=	$nt \mid tt$	$(terms)$
p	::=	tt	$(programs)$
T	::=	`Bool` | `T` $\rightarrow$ `T`	$(types)$

In this exercise, you are to adjust the existing evaluation and typing rules, so that they correctly and comprehensively describe the semantics of the extension. More precisely, your task is two-fold:

1. Extend the evaluation rules (by adding new rules and/or changing existing ones) to express the early return semantics provided by return. Identify the evaluation strategy used by the specification and make sure that your extension adheres to it.

2. Extend the typing rules (by adding new rules and/or changing existing ones) to guarantee that types of values returned via return and via normal means are coherent. Make sure that progress and preservation conditions hold for your extension (you don't have to prove that formally, but your grade will be reduced if your extension ends up being unsafe).

   Hint: In addition to the immediate type of a term, you also need to keep track of the types of returned terms inside that term. For example, instead of the regular typing judgment $\Gamma \vdash t :$ `T`, you can use the $\Gamma \vdash t :$ `T` \| `R`, where `R` is a set of types of terms, i.e. $\{T_1, ...T_n\}$, that can be returned from `t`.

Before you begin, think carefully about the following simple term: $\lambda x :$ `Bool`. (`return true`) x. Intuitively, it makes sense. Once this lambda is applied, it is going to evaluate to `true`, regardless of the input. Now, which typing rules would be used to type this term, so that it is accepted by our language? In particular, what type or types need to be assigned to `return true`?

5 Appendix

5.1 For reference: predefined lambda terms

Predefined lambda terms that can be used as-is in "Hacking with the untyped call-by-value lambda calculus":

$$
\begin{array}{ll}
\texttt{tru} = & \lambda t.\ \lambda f.\ t \\
\texttt{fls} = & \lambda t.\ \lambda f.\ f \\
\texttt{iszro} = & \lambda m.\ m\ (\lambda x.\ fls)\ tru \\
\texttt{test} = & \lambda b.\ \lambda t.\ \lambda f.\ b\ t\ f\ unit \\
 & \\
\texttt{pair} = & \lambda f.\ \lambda s.\ \lambda b.\ b\ f\ s \\
\texttt{fst} = & \lambda p.\ p\ tru \\
\texttt{snd} = & \lambda p.\ p\ fls \\
 & \\
\texttt{c}_0 = & \lambda s.\ \lambda z.\ z \\
\texttt{c}_1 = & \lambda s.\ \lambda z.\ s\ z \\
\texttt{scc} = & \lambda n.\ \lambda s.\ \lambda z.\ s\ (n\ s\ z) \\
\texttt{plus} = & \lambda m.\ \lambda n.\ \lambda s.\ \lambda z.\ m\ s\ (n\ s\ z) \\
\texttt{times} = & \lambda m.\ \lambda n.\ m\ (plus\ n)\ c_0 \\
 & \\
\texttt{zz} = & pair\ c_0\ c_0 \\
\texttt{ss} = & \lambda p.\ pair\ (snd\ p)\ (scc\ (snd\ p)) \\
\texttt{prd} = & \lambda m.\ fst\ (m\ ss\ zz) \\
 & \\
\texttt{fix} = & \lambda f.\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))
\end{array}
$$

5.2 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:

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

Evaluation rules:

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \qquad \text{(E-App1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \qquad \text{(E-App2)}$$

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

Typing rules:

$$\frac{x\colon T \in \Gamma}{\Gamma \vdash x\colon T} \qquad \text{(T-Var)}$$

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

$$\frac{\Gamma \vdash t_1\ :\ T_1 \to T_2 \quad \Gamma \vdash t_2\ :\ T_1}{\Gamma \vdash t_1\ t_2\ :\ T_2} \qquad \text{(T-App)}$$

$$\frac{}{\Gamma \vdash \texttt{true}\ :\ \texttt{Bool}} \qquad \text{(T-False)}$$

$$\frac{}{\Gamma \vdash \texttt{false}\ :\ \texttt{Bool}} \qquad \text{(T-True)}$$