1 Lazy Krivine Machine $K$ [100 points]

Welcome to the last exam in the course! In this problem, we design an abstract machine, called Lazy Krivine Machine $K$, based on the call-by-need reduction strategy. The call-by-need reduction strategy is a variant of the call-by-name strategy: it reduces a function application without evaluating the argument, but is also designed so that it never evaluates the same argument more than once. To this end, it delays the evaluation of the argument until the result is needed. Once the argument is fully evaluated, it stores the result in the heap so that when it needs the argument again, it does not need to repeat the same evaluation.

The abstract machine $K$ is similar to the abstract machine $N$ from Assignment 6 in that both describe the implementation of a lazy functional programming language. Unlike the abstract machine $N$, however, the abstract machine $K$ makes no distinction between analysis states and return states. Moreover, while the abstract machine $N$ stores in its heap two different kinds of objects, namely delayed expressions and computed values, the abstract machine $K$ stores all objects in a uniform way. The transition rules of the machine $K$ are designed to properly handle an expression retrieved from the heap whether it has been fully evaluated or not. You will see that the abstract machine $K$ is much simpler in its definitions and transition rules even though both machines go through similar sequences of transition.

The abstract machine $K$ uses the following definitions:

- $\text{expression} \quad e ::= x \mid \lambda x. e \mid e\ e$
- $\text{closure} \quad c ::= [e, \eta]$
- $\text{environment} \quad \eta ::= \cdot \mid \eta, x \mapsto l$
- $\text{location} \quad l$
- $\text{heap} \quad h ::= \cdot \mid h, l \mapsto c$

- The definition of expressions is the same as in the untyped $\lambda$-calculus.
- A closure $[e, \eta]$ contains an expression $e$ and its evaluation environment $\eta$. Environment $\eta$ maps free variables of expression $e$ to locations pointing to closures (representing their actual values). The machine may start an evaluation of expression $e$ at any time since environment $\eta$ stores locations for all its free variables.
- The machine looks up an environment to retrieve the location $l$ for a free variable $x$.
- A location points to a closure in the heap whereas the heap maps locations to closures. Locations serve as indirections from variables and closures so that multiple environments may share the same closure stored in the heap. For example, we may have two different environments $\eta_1 = x_1 \mapsto l$ and $\eta_2 = x_2 \mapsto l$ where $l \mapsto c$ is in the heap. When the machine updates the heap with a new closure $c'$ for $l$, the effect becomes immediately visible to both variables $x_1$ and $x_2$.

Similarly to the abstract machine $N$, the abstract machine $K$ uses frames and stacks in its states:

- $\text{frame} \quad \phi ::= \text{arg}(c) \mid \text{mark}(l)$
- $\text{stack} \quad \sigma ::= \cdot \mid \sigma; \phi$
- $\text{state} \quad s ::= (e, \eta, \sigma, h)$

- A frame $\text{arg}(c)$ means that the machine has started the evaluation of a function expression but delayed the computation of its argument whose closure is $c$. A frame $\text{mark}(l)$ means that the machine is currently evaluating an argument whose closure is stored at location $l$ in the heap. After fully evaluating the argument, the machine updates location $l$ of the heap with the new closure.

As an example, consider an application $e_1\ e_2$. The machine is designed so that it directly proceeds to evaluating function expression $e_1$ after delaying the evaluation of argument $e_2$. Since the machine
needs to remember that it has delayed the evaluation of argument $e_2$, its pushes a new frame $\text{arg}([e_2, \eta])$ onto the stack where $\eta$ is the current environment. When the machine later produces $\lambda x. e'_1$ from $e_1$, it retrieves $\text{arg}([e_2, \eta])$ from the stack and updates the heap and the environment appropriately before starting to evaluate function body $e'_1$.

Now suppose that the machine is evaluating function body $e'_1$. During the evaluation of function body $e'_1$, the machine may have to evaluate variable $x$ (unless $e'_1$ does not use variable $x$ at all). When it encounters variable $x$, the machine retrieves the location $l$ for variable $x$ from the environment and pushes $\text{mark}(l)$ onto the stack in order to indicate that it is evaluating argument $e_2$ whose closure is stored at location $l$ in the heap. After evaluating argument $e_2$, the machine updates the heap with the new computed value for $e_2$.

- The stack is a sequence of frames.
- The state of the machine is specified by an expression $e$ being evaluated, its evaluation environment $\eta$, the stack $\sigma$, and the heap $h$.

The reduction judgment for the abstract machine $K$ is as follows:

$$s \xrightarrow{K} s' \iff \text{the machine makes a transition from state } s \text{ to another state } s'$$

To define the transition rules, we use a couple of auxiliary functions. First we define the function $\text{dom}(h)$ which returns the set of all locations in a given heap $h$; we also use $[l \mapsto e]h$ for updating the contents of location $l$ in heap $h$ with closure $c$:

$$\text{dom}(\cdot) = \emptyset$$
$$\text{dom}(h, l \mapsto e) = \text{dom}(h) \cup \{l\}$$

$$[l \mapsto c](h, l \mapsto e) = h, l \mapsto c$$

The abstract machine $K$ has four transition rules. Below we informally explain how each transition rule changes the state of the machine. The goal in this problem is to formally translate this idea into four transition rules.

- Rule $\text{App}$:
  When the machine attempts to evaluate a function application $e_1 \ e_2$, it first pushes a closure for the argument $e_2$ in the stack and proceeds to reducing the function $e_1$. The machine pushes $\text{arg}(c)$ to the stack where $c = [e_2, \eta]$. The frame $\text{arg}(c)$ will be used by $\text{Call}$ when the machine completes the evaluation of the function $e_1$ and tries to apply the argument $e_2$.

- Rule $\text{Call}$:
  The machine gets the computed value $\lambda x. e'_1$ after it completes the evaluation of the function $e_1$. The machine takes the argument $e_2$ from the frame $\text{arg}(c)$ which was pushed by $\text{App}$. Then, it stores the closure $c$ in the heap and inserts the location $l$ for $c$ into the environment so that later the machine retrieves $c$ by searching the variable $x$ in the environment. Now that the machine stored the argument in the heap and has the location $l$ for the variable $x$ in the environment, the machine is ready to evaluate the function body $e'_1$.

- Rule $\text{Var}$:
  During the evaluation of the function body $e'_1$, the machine may encounter the variable $x$. When the machine evaluates the variable $x$, it retrieves the argument stored in the heap. The argument $e'_2$ retrieved from the heap is either the computed value of $e_2$ or a delayed expression $e_2$ itself. In either case, the machine pushes a $\text{mark}(l)$ to the stack and starts to evaluate the argument $e'_2$ under its environment $\eta$. Here, $l$ is the location for $c = [e'_2, \eta]$ in the heap.

- Rule $\text{Update}$:
  Once the machine completes the evaluation of $e'_2$ and has the computed value of $e'_2$ in the state, the machine examines the top of the stack. If the stack has $\text{mark}(l)$ on its top, the machine updates the heap with the computed value of $e'_2$. 


Question 1. [80 points] Complete the definition of each transition rule by filling in the premise part, the previous state, and the next state.

(Transition rules)

\[
\frac{(e', e, \eta, \sigma, h) \mapsto_K (e, \eta, \sigma; \text{arg}(e'), \text{mark}(h), \text{mark}(h'))}{\text{App}}
\]

\[
\frac{\lambda x. e, \eta, \sigma \not\in \text{dom}(h) \quad h' = h, l \mapsto e \quad \eta' = \eta, x \mapsto l}{\text{Call}}
\]

\[
\frac{x \mapsto l \in \eta \quad l \mapsto [e, \eta'] \in h}{\text{Var}}
\]

\[
\frac{(\lambda x. e, \eta, \sigma; \text{mark}(h), \text{mark}(h'))}{\text{Update}}
\]

Question 2. [20 points] Show the reduction sequence of the following initial state. There are two blanks for each step: one for the the name of transition rule and the other for the next state.

\[
(\lambda x. x, x) ((\lambda y. y) (\lambda z. z))
\]

\[
\frac{}{\text{App}}(\lambda x. x, x) (\lambda y. y) (\lambda z. z)
\]

\[
\frac{}{\text{Call}}(\lambda x. x, x) ((\lambda y. y) (\lambda z. z))
\]

\[
\frac{}{\text{Var}}(\lambda x. x, x) ((\lambda y. y) (\lambda z. z))
\]

\[
\frac{}{\text{App}}((\lambda y. y) (\lambda z. z))
\]

\[
\frac{}{\text{Call}}((\lambda y. y) (\lambda z. z))
\]

\[
\frac{}{\text{Var}}((\lambda y. y) (\lambda z. z))
\]

\[
\frac{}{\text{Update}}((\lambda y. y) (\lambda z. z))
\]

\[
\frac{}{\text{Call}}((\lambda y. y) (\lambda z. z))
\]

\[
\frac{}{\text{Var}}((\lambda y. y) (\lambda z. z))
\]

\[
\frac{}{\text{Update}}((\lambda y. y) (\lambda z. z))
\]

\[
\frac{}{\text{Call}}((\lambda y. y) (\lambda z. z))
\]

\[
\frac{}{\text{Var}}((\lambda y. y) (\lambda z. z))
\]

\[
\frac{}{\text{Update}}((\lambda y. y) (\lambda z. z))
\]