Recall the definition of MinML from the textbook and recall the notation we used in Homework Set 5 (which was due on November 4, 2008).

We augment this language with reference types and expression forms for creating and manipulating references—in the spirit of what was discussed in class. The additions are:

\[
\begin{align*}
\text{types} & : \quad \tau ::= \ldots | \tau \text{ ref} \\
\text{expressions} & : \quad e ::= \ldots \\
& \quad \text{ref } e \quad \text{create and init. ref.} \\
& \quad ! e \quad \text{dereference} \\
& \quad e ::= e \quad \text{update}
\end{align*}
\]

1. Notice that we did *not* add locations to the syntax of this language. In other words, the values that inhabit any type \(\tau \text{ ref}\) do not have a syntactic representation. This situation is similar to the one we encountered when we added continuations: there, too, we had semantic values—namely machine stacks, a.k.a. continuations—that did not have a syntactic representation.

   (a) Your task is to modify the setup the E machine (everything except the transition rules) in such a way that it supports the above language. Show all the relevant definitions (what are the states of the machine, what possible machine values exist, what do the stack frames look like, etc.).

   (b) Write down all the transition rules for the machine. (Include even those rules that do not directly deal with references.)

2. We now augment the above language further by, again, adding simple exceptions and exception handling using the fail and try forms that we have seen before.

   (a) Your first task is to modify the machine from question 1 in such a way that it also supports exceptions. The behavior should be such that modifications to the memory are not rolled back when an exception is raised. Show all the changes with respect to the machine from question 1.

   (b) Now modify this new machine in such a way that changes to the memory are rolled back when exceptions are raised.
(c) Explain how your solution to the previous question works. Based on your explanation, argue that such behavior might not be practical for a realistic language design.

3. **(extra credit):** Consider the language shown at the beginning of Section 4 (p. 25) of the Lecture Notes. Also, consider Section 4.3, i.e., the discussion of Continuation-Passing Style. Think about how to CPS-convert the `letcc`- and `throw` forms.

Hint: The target language (p. 33) does not have to be modified for this!

(a) What should the translation of a type $\tau$ \texttt{cont} be? Show the definition of CPS($\tau$ \texttt{cont}).

(b) Show the translation rule for `letcc`. For this, start with the \texttt{letcc} typing rule from page 26 and extend it with the translation part. The conclusion of your rule should have the form

$$\Gamma \vdash \texttt{letcc} \ k : \tau \ \texttt{cont} \ \texttt{in} \ e : \tau' \sim \hat{e} : \tau' \ \texttt{comp}.$$ 

You need to figure out what $\hat{e}$ looks like.

(c) Similarly, show the rule for translating a `throw` expression.