Problem 1: Short Answer [20 points]

Evaluate the following expressions in the given models.

\[
(\text{let } ((a \ 1) \\
\quad (g (\lambda (x) \ 25))) \\
(\text{let } ((f (\lambda (y) (\text{if } (< \ a \ y) \\
\quad (g \ (/ \ 2 \ 0)) \\
\quad ((\lambda (x) \ 15) (g \ g))))) \\
(\text{let } ((a \ 4) \\
\quad (y \ 5) \\
\quad (g (\lambda (x) (x \ x)))) \\
(f \ 2))))
\]

a. [5 points] static scoping, call by value

**Solution:** error (divide by 0)

b. [5 points] dynamic scoping, call by value

**Solution:** \(\bot\) (infinite loop)

c. [5 points] static scoping, call by name

**Solution:** 25

d. [5 points] dynamic scoping, call by name

**Solution:** 15
Problem 2: Operational Semantics: Postfix + \{sdup\} [20 points]

Alyssa P. Hacker extended the PostFix language with a new command called sdup: smart dup. This allows us to compute \( \text{square}(x) = x^2 \) without hurting the termination property of PostFix programs. The informal semantics for sdup is as follows: duplicate the top of the stack if it is a number or a command sequence that doesn’t contain sdup; otherwise, report an error.

Formally, the operational semantics has been extended with the following two transition rules:

\[
\begin{align*}
\text{(sdup \cdot Q_{\text{rest}}, N \cdot S)} & \Rightarrow \langle Q_{\text{rest}}, N \cdot N \cdot S \rangle & \text{[sdup-numeral]} \\
\text{(sdup \cdot Q_{\text{rest}}, Q \cdot S)} & \Rightarrow \langle Q_{\text{rest}}, Q \cdot Q \cdot S \rangle & \text{[sdup-sequence]} \\
\end{align*}
\]

contains\_sdup : Command* → Bool is a helper function that takes a sequence of commands and checks whether it contains sdup or not (yes, contains\_sdup handles even nested sequences of commands).

As a new graduate student in Alyssa’s AHRG (Advanced Hacking Research Group), you were assigned to give a proof that all PostFix + \{sdup\} programs terminate. However, you are not alone! Alyssa already took care of most of the mathematical weaponry:

Consider the product domain \( P = \text{Nat} \times \text{Nat} \) (as usual, \text{Nat} is the set of natural numbers, starting with 0). On this domain, we define the relation \(<_P\) as follows:

**Definition 1 (lexicographic order)** \( \langle a_1, b_1 \rangle <_P \langle a_2, b_2 \rangle \) iff:

a. \( a_1 < a_2 \) or

b. \( a_1 = a_2 \) and \( b_1 < b_2 \).

E.g. \( \langle 3, 10000 \rangle < _P \langle 4, 0 \rangle, \langle 5, 2 \rangle < _P \langle 5, 3 \rangle \).

**Definition 2** A strictly decreasing chain in \( P \) is a finite or infinite sequence of elements \( p_1, p_2, \ldots \) such that \( p_i \in P, \forall i \) and \( p_{i+1} <_P p_i, \forall i \).

After a long struggle, Alyssa proved the following lemma for you:

**Lemma 1** There is no infinite strictly decreasing chain in \( P \).

Give a rigorous proof that each PostFix + \{sdup\} program terminates by using a cleverly defined energy function \( E_{\text{config}} \). **Hint:** Each transition of Postfix reduces the energy function \( E_{\text{config}} \) you saw in class. Try to see what is reduced by the two new rules, and how you can combine these two things into a single energy function.

*Note:* If you need to use some helper functions that are intuitively easy to describe but tedious to define (e.g. contains\_sdup), just give an informal description of them.

**Grading scheme:**

- [10 points] \( E_{\text{config}} \)
- [10 points] Termination proof.

**Solution:**

Consider the following energy function:

\[
E_{\text{config}} : C \rightarrow \text{Nat} \times \text{Nat} = \lambda (Q, S) . \langle \text{sdup\_count}[\langle Q, S \rangle], E_{\text{config}}[\langle Q, S \rangle] \rangle
\]

where \( \text{sdup\_count} \) is a helper function that computes the number of times sdup appears in a configuration and \( E_{\text{config}} \) is the energy function shown in class.

Let’s first prove that for any transition \( c_{\text{old}} \Rightarrow c_{\text{new}}, E_{\text{config}}[c_{\text{new}}] <_P E_{\text{config}}[c_{\text{old}}] \).
Old transitions: None of them introduces new \texttt{sdup} commands but they all strictly decrease $\mathcal{E}_{\text{config}}$. So, the first component of $\mathcal{E}_{\text{config}}$ doesn’t increase and the second one strictly decreases which implies $\mathcal{E}_{\text{config}}[c_{\text{new}}] < P \mathcal{E}_{\text{config}}[c_{\text{old}}]$.

New transitions: Each of the new \texttt{sdup} related rules “consumes” exactly one \texttt{sdup}: this is clearly true for \texttt{[dup-numeral]} and \texttt{[dup-sequence]} doesn’t duplicate sequences containing \texttt{sdup}. So the first component of $\mathcal{E}_{\text{config}}$ is strictly decreased by these transitions which implies that no matter what happens with the second component (note that \texttt{[dup-sequence]} might actually increase it), $\mathcal{E}_{\text{config}}[c_{\text{new}}] < P \mathcal{E}_{\text{config}}[c_{\text{old}}]$ for the new transitions too.

Suppose now for the sake of contradiction that there is some PostFix $+ \{ \texttt{sdup} \}$ program with an infinite execution $c_1 \Rightarrow c_2 \Rightarrow c_3 \Rightarrow \ldots$. This implies $\mathcal{E}_{\text{config}}[c_2] < P \mathcal{E}_{\text{config}}[c_1], \mathcal{E}_{\text{config}}[c_3] < P \mathcal{E}_{\text{config}}[c_2], \ldots$ and we’ve just constructed an infinite strictly decreasing chain in $P$! Contradiction with Lemma 1.
Problem 3: State: FLK! + {undo-once!} [30 points]

Ben Bitdiddle introduced a new undo-once! instruction to roll the store back one operation at a time. Informally speaking, undo-once! undoes the last store operation (cell or cell-set!). If there is no store operation to undo, undo-once! does nothing.

\[
E ::= \ldots \quad \text{[Classic FLK! expressions]}
| \quad (\text{undo-once!}) \quad \text{[Undo last store operation]}
\]

Initially, Ben thought of modifying the meaning function to use a stack of stores (as it did in the fall-98 midterm), but the implementors refused to work on such an idea and threatened to quite Ben’s company en masse. So, Ben had to turn to a more efficient idea: maintain the current store and a stack of undo functions. An undo function takes a store and reverses a specific store operation (one done with cell or cell-set!) to obtain the store before the operation.

Pursuing this idea, Ben modified the Cmdcont semantic domain and the top level function as follows:

\[
\text{Cmdcont} = \text{Store} \rightarrow \text{StoreTransformStack} \rightarrow \text{Expressible}
\]

\[
h \in \text{StoreTransformStack} = \text{StoreTransform}^*\]

\[
t \in \text{StoreTransform} = \text{Store} \rightarrow \text{Store}
\]

\[
T \mathcal{L}[E] = (E[E\emptyset] \text{ empty-env top-level-cont empty-store } [\text{StoreTransform}])
\]

As each store operation (cell or cell-set!) consists of assigning a Storable to a Location, it can be reversed by putting the old Assignment into that Location. Ben even wrote the following undo function producer for you:

\[
\text{make-undofun} : \text{Location} \rightarrow \text{Assignment} \rightarrow \text{StoreTransform}
\]

\[
= \lambda l \alpha . \lambda s . (\text{assign'} l \alpha s)
\]

assign’ is a function similar to assign which allows us to assign even unassigned:

\[
\text{assign'} : \text{Location} \rightarrow \text{Assignment} \rightarrow \text{Store} \rightarrow \text{Store}
\]

\[
= \lambda l_1 \alpha . \lambda l_2 . \text{if} (\text{same-location?} \ l_1 \ l_2) \text{ then } \alpha \text{ else } \text{fetch} \ l_2 \ s \ \text{fi}
\]

If a store operation modified location \( l \), the undo function for it can be obtained by calling make-undofun on \( l \) and the old assignment for \( l \). All the undo functions that you write in this problem must be obtained by calling make-undofun with the appropriate arguments.

Now, guess what?, Ben went away to deliver a better Internet and grab some more billions, and you were assigned to finish his job.

a. [10 points] Write the meaning function clause for \( \mathcal{E}[(\text{undo-once!})] \).

Solution:

\[
\mathcal{E}[(\text{undo-once!})] = \\
\lambda eksh . \text{matching} h
\]

\[
\triangleright t.h_{\text{rest}} \ (k (\text{Unit} \rightarrow \text{Value} \ unit) \ (t \ s) \ h_{\text{rest}})
\]

\[
\triangleright [\text{StoreTransform}] \ (k (\text{Unit} \rightarrow \text{Value} \ unit) \ s \ h)
\]

\text{endmatching}

We specially treat the case of an empty stack of undo functions: when there is nothing to undo, undo-once! does nothing.

b. [10 points] Write a revised version for \( \mathcal{E}[(\text{primop cell-set! E}_1 E_2)] \).

Solution:
\[ \mathcal{E}[\text{primop cell-set! } E_1 E_2] = \lambda ek. (\mathcal{E}[E_1] e (\text{test-location } (\lambda l. (\mathcal{E}[E_2] e (\lambda vsh. (k (\text{Unit} \mapsto \text{Value unit}) (\text{assign } l v s) (\text{make-undofun } l (\text{fetch } l s).h))))))) \]

The store that is passed to \( k \) is, as previously, the store obtained by assigning \( v \) to location \( l \); we add to the head of the stack of store transformers an undo function that restores the old assignment for \( l \).

c. [10 points] Write a revised version for \( \mathcal{E}[(\text{cell } E)] \). Note: we want to be able to undo even cell creation operations. That is, the following program must end with an error:

\begin{verbatim}
(let ((c (cell 0)))
  (begin
   (undo-once!)
   (primop cell-ref c)))
\end{verbatim}

Solution:
\[ \mathcal{E}[(\text{cell } E)] = \lambda ek. (\mathcal{E}[E] e (\lambda vsh. ((\lambda l. (k (\text{Location} \mapsto \text{Value } l) (\text{assign } l v s) (\text{make-undofun } l (\text{Unassigned} \mapsto \text{Assignment } \text{unassigned}).h))) (\text{fresh-loc } s)))) \]

Undoing a cell allocation is done by assigning back \textit{unassigned} to the cell location \( l \). Now, that cell is free to be allocated again! Calling \((\lambda l. \ldots)\) on \((\text{fresh-loc } s)\) is just a trick to avoid us writing \((\text{fresh-loc } s)\) three times (it’s like the desugaring for \texttt{let} in FL).
Problem 4: Denotational Semantics: Control [30 points]

Sam Antics of eFLK.com wants to cash in on the election year media bonanza by introducing a new feature into standard FLK!:

\[(\text{elect } E_{\text{pres}} E_{\text{vp}}) ; \text{ evaluates to } E_{\text{pres}} \text{ unless } \text{impeach} \]
\[; \text{ is evaluated within } E_{\text{pres}}, \text{ in which} \]
\[; \text{ case evaluates to } E_{\text{vp}}. \text{ If } \text{impeach} \text{ is} \]
\[; \text{ evaluated within } E_{\text{vp}}, \text{ signals an error.} \]

\[(\text{reelect}) ; \text{ if evaluated within } E_{\text{pres}} \text{ of } (\text{elect } E_{\text{pres}} E_{\text{vp}}), \]
\[; \text{ goes back to the beginning of elect.} \]
\[; \text{ otherwise, signals an error.} \]

\[(\text{impeach}) ; \text{ if evaluated within } E_{\text{pres}} \text{ of } (\text{elect } E_{\text{pres}} E_{\text{vp}}), \]
\[; \text{ causes the expression to evaluate to } E_{\text{vp}}. \]
\[; \text{ otherwise, signals an error.} \]

For example:

(let ((scandals (primop cell 0)))
  (elect (if (< (primop cell-ref scandals) 5)
   (begin (primop cell-set! (* (primop cell-ref scandals) 1))
   (reelect))
   (impeach))
  (* (primop cell-ref scandals) 2))))
⇒ 10

You are hired by eFLK.com to modify the standard denotational semantics of FLK! to produce FLK! 2000 Presidential Edition (TM). To get you started, Sam tells you that he has added the following domains:

\[ r \in \text{Prescont} = \text{Cmdcont} \]
\[ i \in \text{Vpcont} = \text{Cmdcont} \]

He also changed the signature of the meaning function:

\[ \mathcal{E} : \text{Exp} \to \text{Environment} \to \text{Prescont} \to \text{Vpcont} \to \text{Expcont} \to \text{Cmdcont} \]

a. [9 points] give the meaning function for \((\text{elect } E_{\text{pres}} E_{\text{vp}})\).

Solution:

\[ \mathcal{E}[(\text{elect } E_{\text{pres}} E_{\text{vp}})] = \]
\[ \lambda erik . (\text{fix}_{\text{Cmdcont}} \lambda r_1 . \mathcal{E}[E_{\text{pres}}] e r_1 (\lambda s . \mathcal{E}[E_{\text{vp}}] e \text{error-cont cannot-reelect-vp}) \text{error-cont cannot-impeach-vp} k k)) \]

b. [7 points] give the meaning function for \((\text{reelect})\).

Solution:

\[ \mathcal{E}[(\text{reelect})] = \lambda erik . r \]
c. [7 points] give the meaning function for \((\text{impeach})\).

Solution:
\[
\mathcal{E}[\text{(impeach)}] = \lambda \text{erik} . \text{i}
\]

d. [7 points] using the meaning functions you defined, show that \((\text{elect (reelect)} 1)\) is equivalent to \(\bot\).

Solution:
\[
\begin{align*}
\mathcal{E}[\text{(elect (reelect)} 1)] &= \lambda \text{erik} . (\text{fix}_{\text{Cmdcont}} (\lambda r_1 . \mathcal{E}[\text{(reelect)}] e r_1 (\lambda s . \mathcal{E}[1] e \\
&\quad \text{(error-cont cannot-reelect-vp)} \\
&\quad \text{(error-cont cannot-impeach-vp)} k) k)) \\
\Rightarrow \mathcal{E}[\text{(elect (reelect)} 1)] &= \lambda \text{erik} . (\text{fix}_{\text{Cmdcont}} (\lambda r_1 . (\lambda \text{erik} . r e r_1 (\lambda s . \mathcal{E}[1] e \\
&\quad \text{(error-cont cannot-reelect-vp)} \\
&\quad \text{(error-cont cannot-impeach-vp)} k) k)) \\
\Rightarrow \mathcal{E}[\text{(elect (reelect)} 1)] &= \lambda \text{erik} . (\text{fix}_{\text{Cmdcont}} (\lambda r_1 . r_1)) \\
\Rightarrow \mathcal{E}[\text{(elect (reelect)} 1)] &= \bot
\end{align*}
\]