Please do not turn the page until everyone is ready.

Rules:

- The exam is closed-book, closed-note, except for one side of one 8.5x11in piece of paper.
- Please stop promptly at 11:50.
- You can rip apart the pages, but please write your name on each page.
- There are **100 points** total, distributed **unevenly** among **4** questions (which have multiple parts).

Advice:

- Read questions carefully. Understand a question before you start writing.
- Write down thoughts and intermediate steps so you can get partial credit.
- The questions are not necessarily in order of difficulty. **Skip around.** In particular, make sure you get to all the problems.
- If you have questions, ask.
- Relax. You are here to learn.
Name:______________________________

For your reference:

\[ s ::= \text{skip} \mid x := e \mid s \mid \text{if } e \ s \mid \text{while } e \ s \]
\[ e ::= c \mid x \mid e + e \mid e * e \]
\((c \in \{-\infty, -2, -1, 0, 1, 2, \ldots\})\)
\((x \in \{x_1, x_2, \ldots, y_1, y_2, \ldots, z_1, z_2, \ldots\})\)

\[
H ; e \Downarrow c
\]

<table>
<thead>
<tr>
<th>CONST</th>
<th>VAR</th>
<th>ADD</th>
<th>MULT</th>
</tr>
</thead>
<tbody>
<tr>
<td>(H ; c \Downarrow c)</td>
<td>(H ; x \Downarrow H(x))</td>
<td>(H ; e_1 \Downarrow c_1) (H ; e_2 \Downarrow c_2)</td>
<td>(H ; e_1 \Downarrow c_1) (H ; e_2 \Downarrow c_2)</td>
</tr>
</tbody>
</table>

\[
H_1 ; s_1 \rightarrow H_2 ; s_2
\]

\[
\text{IF 1}
\]
\[
H ; e \Downarrow c\quad c > 0\quad \text{IF 2}
\]  
\[
H ; e \Downarrow c\quad c \leq 0\quad \text{WHILE}
\]  
\[
H ; \text{if } s_1 s_2 \rightarrow H ; s_1\quad H ; \text{if } s_1 s_2 \rightarrow H ; s_2\quad \text{H ; while } e s \rightarrow H ; \text{if } (s ; \text{while } e s) \text{ skip}
\]

\[
e \ ::= \lambda x. e \mid x \mid e \mid e \mid c
\]
\[
v \ ::= \lambda x. e \mid c
\]
\[
\tau \ ::= \text{int} \mid \tau \rightarrow \tau
\]

\[
e \rightarrow e'
\]

\[
(\lambda x. e) v \rightarrow e[v/x]
\]
\[
e_1 \rightarrow e'_1\quad e_2 \rightarrow e'_2\quad e_1 \rightarrow e'_1\quad e_2 \rightarrow e'_2
\]

\[
e[e'/x] = e''
\]

\[
x[e/x] = e
\]
\[
e_1[e/x] = e'_1\quad y \neq x\quad y \notin \text{FV}(e)
\]
\[
(\lambda y. e_1)[e/x] = \lambda y. e'_1
\]
\[
y[e/x] = y\quad e_1[e/x] = e'_1\quad e_2[e/x] = e'_2
\]
\[
(e_1 e_2)[e/x] = e'_1 e'_2
\]

\[
\Gamma \vdash e : \tau
\]

\[
\Gamma \vdash c : \text{int}\quad \Gamma \vdash x : \Gamma(x)\quad \Gamma, x : \tau_1 \vdash e : \tau_2\quad \Gamma \vdash e_1 : \tau_1\quad \Gamma \vdash e_2 : \tau_1
\]

- If \(\vdash e : \tau\) and \(e \rightarrow e'\), then \(\vdash e' : \tau\).
- If \(\vdash e : \tau\), then \(e\) is a value or there exists an \(e'\) such that \(e \rightarrow e'\).
- If \(\Gamma, x : \tau' \vdash e : \tau\) and \(\vdash e' : \tau'\), then \(\vdash e[e'/x] : \tau\).
1. In this problem, we consider an expression language that is like expressions in IMP except we remove multiplication and we add a global counter. Our syntax is:

\[ e ::= c | x | e + e | \text{next} \]

Informally, the next expression evaluates to the current counter-value and has the side-effect of incrementing the counter value.

(a) (11 points) Give a large-step semantics for this expression language. The judgment should have the form \( H; c_1; e \Downarrow c_2; e \) where:

- \( H \), \( e \), and \( c \) are like in IMP.
- \( c_1 \) is the value of the global counter before evaluation.
- \( c_2 \) is the value of the global counter after evaluation.

(b) (16 points) Prove this theorem: If \( H; c_1; e \Downarrow c_2; e \) and \( c_1' > c_1 \), then there exist \( c_2' \) and \( c' \) such that \( H; c_1'; e \Downarrow c_2'; c' \) and \( c_2' > c_2 \).

(c) (7 points) Suppose we also extend IMP statement semantics to support the global counter (so the judgment has the form \( H; c_1; s \rightarrow^* H_2; c_2; \text{skip} \)). Argue that this theorem is false: If \( H_1; c_1; s \rightarrow^* H_2; c_2; \text{skip} \) and \( c_1' > c_1 \), then there exist \( H_2' \) and \( c_2' \) such that \( H_1; c_1; s \rightarrow^* H_2'; c_2'; \text{skip} \) and \( c_2' > c_2 \). You do not need to give the semantic rules for statements or show a full state sequence. Just give an example showing the theorem is false and explain why informally.

Solution:

(a)

\[
\begin{array}{ccc}
\text{CONST} & \text{VAR} & \text{ADD} \\
H; c_1; c_2 \Downarrow c_1; c_2 & H; c_1; x \Downarrow c_1; H(x) & H; c_1; e_1 \Downarrow c_1; c_1; c_2 \\
\text{next} & H; c_1; \text{next} \Downarrow c_1; c + 1; c \\
\end{array}
\]

(b) By induction on the derivation of \( H; c_1; e \Downarrow c_2; e \):

- If the derivation ends with \text{CONST}, then \( c_2 = c_1 \) and we can use \text{CONST} to derive \( H; c_1' \Downarrow c_1'; c \). Since \( c_1' > c_1 = c_2 \), letting \( c_2' = c_1' \) (and \( c' = c \)) suffices.
- If the derivation ends with \text{VAR}, then \( c_2 = c_1 \), and we can use \text{VAR} to derive \( H; c_1' \Downarrow c_1'; c \). Since \( c_1' > c_1 = c_2 \), letting \( c_2' = c_1' \) (and \( c' = c \)) suffices.
- If the derivation ends with \text{ADD}, then \( e = e_1 + e_2 \) and there exists some \( c_3, c_4, \) and \( c_5 \) such that \( H; c_1; e_1 \Downarrow c_3; c_4 \) and \( H; c_2; e_2 \Downarrow c_5 \). So by induction on the derivation for \( e_1 \) there exist \( c_3' > c_3 \) and \( c_4' \) such that \( H; c_1'; e_1 \Downarrow c_3'; c_4' \). Since \( c_3' > c_3 \), by induction on the derivation for \( e_2 \) there exist \( c_2' > c_2 \) and \( c_5' \) such that \( H; c_2'; e_2 \Downarrow c_2'; c_5' \). So using \text{ADD} with \( H; c_1'; e_1 \Downarrow c_3'; c_4' \) and \( H; c_2'; e_2 \Downarrow c_2'; c_5' \) we can derive \( H; c_1'; e_1 + e_2 \Downarrow c_3'; c_4'; c_5' \) where \( c_3' > c_2 \).
- If the derivation ends with \text{NEXT}, then \( c_2 = c_1 + 1 \) and we can use \text{NEXT} to derive \( H; c_1'; \text{next} \Downarrow c_1' + 1; c_1' \). Since \( c_1' > c_1 \), we know \( c_1' + 1 > c_1 + 1 = c_2 \).

(c) The essence of the problem is conditionals (or loops). For example, consider \( s = \text{if next skip next} \). If \( c_1 = 0 \) and \( c_1' = 1 \), then \( H; c_1; s \rightarrow^* H; 2; \text{skip} \) and \( H; c_1'; s \rightarrow^* H; 2; \text{skip} \), but \( 2 \neq 2 \).
Name: ________________________________
(This page intentionally blank)
2. (10 points) In this problem we extend IMP statements with the construct \texttt{repeat} \( c \) \texttt{s}. Informally, the idea is to execute \( s \) \( c \) times. Here are two separate ways one might add rules to the semantics:

- First way:

\[
\begin{array}{c}
c > 0 \\
H; \text{repeat } c \ s \rightarrow H; (s; \text{repeat } (c - 1) \ s) \\
\end{array}
\quad
\begin{array}{c}
c \leq 0 \\
H; \text{repeat } c \ s \rightarrow H; \text{skip} \\
\end{array}
\]

- Second way:

\[
H; \text{repeat } c \ s \rightarrow H; (s; \text{if } (c - 1) (\text{repeat } (c - 1) \ s) \text{ skip})
\]

One of these ways is wrong (in some situations) according to the informal description.

(a) Which way is wrong? Explain why it is wrong.
(b) Show how to change the wrong way to make it correct.

\textbf{Solution:}

(a) The second way is wrong; it always executes \( s \) at least once. If \( c \leq 0 \), it should not execute \( s \) any times.

(b) We can still use the idea of unrolling to an if-statement; we just cannot assume \( s \) executes at least once. This simpler approach works fine, just like for while-statements:

\[
H; \text{repeat } c \ s \rightarrow H; \text{if } c (s; \text{repeat } (c - 1) \ s) \text{ skip}
\]
3. (18 points) Note there is a part (a) and part (b) to this problem.

(a) For each Caml function below (q1, q2, and q3):

- Describe in 1–2 English sentences what the function computes.
- Give the type of the function. (Hint: For all three functions, the type has one type variable.)

```caml
let q1 x = 
  let rec g x y = 
    match x with 
    | [] -> y 
    | hd::tl -> g tl (hd::y) 
  in g x []

let rec q2 f lst = 
  match lst with 
  | [] -> [] 
  | hd::tl -> if f hd then hd::(q2 f tl) else q2 f tl

let q3 x g = g (g x)
```

(b) Consider this purposely complicated code that uses q3 as defined above.

```caml
let x = q3 2 
let y z = z+z 
let z = 9 
let x = x y
```

After evaluating this code, what is x bound to?

**Solution:**

(a) • q1 takes a list and returns its reverse. It has type `'a list -> 'a list`.

• q2 takes a function and a list and returns the list containing all the elements from the input list (in order) for which the function applied to the element returns true. (It's a filter.) It has type (`'a -> bool`) -> `'a list` -> `'a list`.

• q3 returns the result of applying its second argument to the result of applying its second argument to its first argument. It has type `'a -> ('a -> 'a) -> 'a`.

(b) 8
4. In this problem, we consider a call-by-value lambda-calculus with very basic support for profiling: In addition to computing a value, it computes how many times an expression of the form \texttt{count} \ e is evaluated. Here is the syntax and operational semantics:

\[ e ::= \lambda x. \ e \mid x \mid e \ e \mid c \mid \texttt{count} \ e \]

\[
\begin{array}{c}
c; e \rightarrow c'; e' \\
c; (\lambda x. \ e) \ v \rightarrow c; e[v/x] \\
c; e_1 \rightarrow c'; e'_1 \\
c; e_1 \ e_2 \rightarrow c'; e'_1 \ e_2 \\
c; e_2 \rightarrow c'; e' \ e_2 \\
c; \texttt{count} \ v \rightarrow c + 1; v \\
c; \texttt{count} \ e \rightarrow c'; \texttt{count} \ e'
\end{array}
\]

Given a source program \( e \), our initial state is 0; \( e \) (i.e., the count starts at 0). A program state \( c; e \) type-checks if \( e \) type-checks (i.e., the count can be anything).

(a) (6 points) Give a typing rule for \texttt{count} \ \ e that is sound and not unnecessarily restrictive.

(b) (13 points) State an appropriate Preservation Lemma for this language. Prove just the case(s) directly involving \texttt{count} \ \ e expressions.

(c) (13 points) State an appropriate Progress Lemma for this language. Prove just the case(s) directly involving \texttt{count} \ \ e expressions.

(d) (6 points) Give an example program that terminates in our language and would terminate if we changed function application to be call-by-name but under call-by-name it would produce a different resulting count. (Hint: This should not be difficult.)

Solution:

(a)

\[
\Gamma \vdash e : \tau \\
\hline
\Gamma \vdash \texttt{count} \ e : \tau
\]

(b) If \( \cdot \vdash e : \tau \) and \( c; e \rightarrow c'; e' \), then \( \cdot \vdash e' : \tau \). We can prove this by induction on the derivation of \( \cdot \vdash e : \tau \). In the case we’re asked to prove, the bottom of the derivation looks like:

\[
\cdot \vdash e_0 : \tau \\
\hline
\cdot \vdash \texttt{count} \ e_0 : \tau
\]

There are two possible ways \( c; \texttt{count} \ e_0 \) can step to some \( e' \). If \( e_0 \) is a value, then \( e' = e_0 \) and the assumed derivation’s hypothesis \( \cdot \vdash e_0 : \tau \) suffices. If \( e_0 \) is not a value, then \( e' = \texttt{count} \ e_0' \) where \( c; e_0 \rightarrow c'; e_0' \). So using \( \cdot \vdash e_0 : \tau \) and induction, \( \cdot \vdash e_0' : \tau \), so we can derive \( \cdot \vdash \texttt{count} \ e_0' : \tau \).

(c) If \( \cdot \vdash e : \tau \), then \( e \) is a value or there exists an \( e' \) and \( e' \) such that \( c; e \rightarrow c; e' \). In the case we’re asked to prove the bottom of the derivation looks like:

\[
\cdot \vdash e_0 : \tau \\
\hline
\cdot \vdash \texttt{count} \ e_0 : \tau
\]

So using \( \cdot \vdash e_0 : \tau \), by induction either \( e_0 \) is a value or \( c; e_0 \rightarrow c'; e_0' \) for some \( c' \) and \( e_0' \). If \( e_0 \) is a value, then \( c; \texttt{count} \ e_0 \rightarrow c + 1; e_0 \). If \( c; e_0 \rightarrow c'; e_0' \), then we can derive \( c; \texttt{count} \ e_0 \rightarrow c'; \texttt{count} \ e_0' \).

(d) One of an infinite number of examples is \((\lambda x. 0)(\texttt{count} 0)\).