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 5 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:

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

\[ H ; e \Downarrow c \]

\begin{align*}
\text{CONST} & & \text{VAR} & & \text{ADD} & & \text{MULT} \\
H ; c & \Downarrow c & H ; x & \Downarrow H(x) & H ; e_1 & \Downarrow c_1 & H ; e_2 & \Downarrow c_2 & H ; e_1 \ast e_2 & \Downarrow c_1 \ast c_2 \\
H_1 ; s_1 & \rightarrow H_2 ; s_2 & H ; e ; c & \rightarrow H ; x \rightarrow c ; \text{skip} & H ; \text{skip}; s & \rightarrow H ; s & H ; s_1 & \rightarrow H' ; s_1' & H ; s_1; s_2 & \rightarrow H' ; s_1'; s_2 \\
\text{IF1} & & H ; e ; c & \rightarrow c > 0 & H ; e ; c & \rightarrow c \leq 0 & H ; \text{while } e ; s & \rightarrow H ; s & \text{IF2} & & H ; \text{if } e ; s_1 s_2 & \rightarrow H ; s_1 & H ; \text{if } e ; s_1 s_2 & \rightarrow H ; s_2 & H ; \text{while } e ; s & \rightarrow H ; \text{if } e \text{ (s; while } e \text{ s) skip}
\end{align*}

\[ e \rightarrow e' \]

\[
(\lambda x. e) v \rightarrow e[v/x]
\]

\[
e_1 \rightarrow e'_1 \quad e_1 e_2 \rightarrow e'_1 e_2 \quad e_2 \rightarrow e'_2 \quad v e_2 \rightarrow v e'_2
\]

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

\[
x[e/x] = e \\
y \neq x \\
y[e/x] = y \\
c[e/x] = c
\]

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

\[
e_1[e/x] = e'_1 \\
e_2[e/x] = e'_2
\]

\[
(\lambda y. e_1)(e_2)[e/x] = e'_1 e'_2
\]

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

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

- Preservation: If \( \vdash e : \tau \) and \( e \rightarrow e' \), then \( \vdash e' : \tau \).
- Progress: If \( \vdash e : \tau \), then \( e \) is a value or there exists an \( e' \) such that \( e \rightarrow e' \).
- Substitution: If \( \Gamma, x : \tau' \vdash e : \tau \) and \( \Gamma \vdash e' : \tau' \), then \( \Gamma \vdash e[e'/x] : \tau \).
1. In this problem, we change IMP by adding one more constant, □:

\[ c \in \{\ldots, -2, -1, 0, 1, 2, \ldots\} \cup \{\square\} \]

Because □ is a constant, it can also be an expression, the result of evaluating an expression, or the contents of a heap variable. However, □ is not a legal argument to any “math” operators like “(blue) plus” except “=” and “≠”.

Informally, if an expression has any subexpression that evaluates to □, then the expression evaluates to □.

(a) (7 points) Add four inference rules to the \( H ; e \Downarrow c \) judgment to account for □.

(b) (17 points) Considering all the inference rules now in the language, prove that if \( e \) contains a □ and \( H ; e \Downarrow c \), then \( c \) is □. Hint: Use induction. The new rules from part (a) are not the difficult cases.

(c) (6 points) Our IMP statement semantics can now get stuck. In English, explain exactly how this could occur. Propose a small change to the statement semantics to avoid this. Give any new inference rule(s) and explain in English how you changed the meaning of the language.

Solution:

(a)

sq1

\[ H ; e_1 \Downarrow \square \quad H ; e_2 \Downarrow \square \quad H ; e_1 \Downarrow \square \quad H ; e_2 \Downarrow \square \]

\[ H ; e_1 + e_2 \Downarrow \square \quad H ; e_1 + e_2 \Downarrow \square \quad H ; e_1 \cdot e_2 \Downarrow \square \quad H ; e_1 \cdot e_2 \Downarrow \square \]

(b) Proof by induction on the derivation of \( H ; e \Downarrow c \) proceeding by the rules instantiated at the bottom of the derivation:

CONST Then \( e = c \), so if \( e \) contains □, then \( e \) and \( c \) are □.

VAR Holds vacuously because \( e \) is some \( x \) and therefore does not contain □.

ADD Because the result is \( c_1 + c_2 \), neither \( c_1 \) nor \( c_2 \) is □. Therefore, since \( H ; e_1 \Downarrow c_1 \) and \( H ; e_2 \Downarrow c_2 \), by induction neither \( e_1 \) nor \( e_2 \) contains □. So \( e_1 + e_2 \) does not contain □ and the theorem holds vacuously.

MULT Because the result is \( c_1 \cdot c_2 \), neither \( c_1 \) nor \( c_2 \) is □. Therefore, since \( H ; e_1 \Downarrow c_1 \) and \( H ; e_2 \Downarrow c_2 \), by induction neither \( e_1 \) nor \( e_2 \) contains □. So \( e_1 \cdot e_2 \) does not contain □ and the theorem holds vacuously.

sq1 Holds trivially because \( c \) is □.

sq2 Holds trivially because \( c \) is □.

sq3 Holds trivially because \( c \) is □.

sq4 Holds trivially because \( c \) is □.

Note: The proof can also be done by structural induction on \( e \), but then you need to argue for each form of \( e \) what rules could apply.

(c) A statement can get stuck if the expression in an if-statement evaluates to □ because it is neither the case that □ > 0 nor □ ≤ 0. There are of course many ways to fix this. We could add the rule below, which has the effect of treating □ as false, just like a non-positive number.

\[ \text{if3} \quad H ; e \Downarrow \square \]

\[ H \quad \text{if} e \quad s_1 \quad s_2 \quad \rightarrow \quad H \quad s_2 \]
Name:______________________________

(This page intentionally blank)
2. In this problem, we use the following large-step semantics for IMP statements: The judgment is $H; s \Downarrow H'$ meaning $s$ under heap $H$ produces heap $H'$. The inference rules are:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Premise</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SKIP</strong></td>
<td>$H; \text{skip} \Downarrow H$</td>
<td>$H; \text{assign} c \Downarrow c$</td>
</tr>
<tr>
<td><strong>ASSIGN</strong></td>
<td>$H; e \Downarrow c$</td>
<td>$H; x := e \Downarrow H, x \mapsto c$</td>
</tr>
<tr>
<td><strong>SEQ</strong></td>
<td>$H; s_1 \Downarrow H_1$</td>
<td>$H; s_2 \Downarrow H_2$</td>
</tr>
<tr>
<td><strong>IF1</strong></td>
<td>$H; e \Downarrow c$</td>
<td>$H; s_1 \Downarrow H_1$</td>
</tr>
<tr>
<td><strong>IF2</strong></td>
<td>$H; e \Downarrow c$</td>
<td>$H; s_1 \Downarrow H_1$</td>
</tr>
<tr>
<td><strong>WHILE</strong></td>
<td>$H; e \Downarrow H'$</td>
<td>$H; \text{while } e \Downarrow H'$</td>
</tr>
</tbody>
</table>

The sequence operator is associative. That is, $s_1; (s_2; s_3)$ and $(s_1; s_2); s_3$ are equivalent.

(a) (5 points) State this associativity fact formally as a theorem in terms of the large-step semantics for statements.

(b) (16 points) Prove the theorem you stated in part (a). Hint: Do not use induction.

**Solution:**

(a) (For all $H$, $H'$, $s_1$, $s_2$, and $s_3$), $H; (s_1; (s_2; s_3)) \Downarrow H'$ if and only if $H; ((s_1; s_2); s_3) \Downarrow H'$.

(b) We prove the two directions of the if and only if separately.

First assume $H; (s_1; (s_2; s_3)) \Downarrow H'$. Inverting the derivation ensures we have a derivation that looks like this for some $H_1$ and $H_2$:

\[
\vdots \quad H; (s_1; (s_2; s_3)) \Downarrow H' \quad \vdots
\]

\[
H; s_1 \Downarrow H_1 \quad H_1; s_2 \Downarrow H_2 \quad H_2; s_3 \Downarrow H'
\]

So we know $H; s_1 \Downarrow H_1$, $H_1; s_2 \Downarrow H_2$, and $H_2; s_3 \Downarrow H'$, from which we can derive:

\[
H; s_1 \Downarrow H_1 \quad H_1; s_2 \Downarrow H_2 \quad H_2; s_3 \Downarrow H' \quad H_1; ((s_1; s_2); s_3) \Downarrow H'
\]

Now assume $H; ((s_1; s_2); s_3) \Downarrow H'$. Inverting the derivation ensures we have a derivation that looks like this for some $H_1$ and $H_2$:

\[
\vdots \quad H; ((s_1; s_2); s_3) \Downarrow H' \quad \vdots
\]

\[
H; s_1 \Downarrow H_1 \quad H_1; s_2 \Downarrow H_2 \quad H_2; s_3 \Downarrow H'
\]

So we know $H; s_1 \Downarrow H_1$, $H_1; s_2 \Downarrow H_2$, and $H_2; s_3 \Downarrow H'$, from which we can derive:

\[
H; s_1 \Downarrow H_1 \quad H_1; s_2 \Downarrow H_2 \quad H_2; s_3 \Downarrow H' \quad H_1; (s_1; (s_2; s_3)) \Downarrow H'
\]
Name:______________________________

(This page intentionally blank)
3. (14 points) Describe what, if anything, each of the following Caml programs would print:

(a) let f x y = x y in
   let z = f print_string "hi " in
   f print_string "hi"

(b) let f x = (fun y -> print_string x) in
    let g = f "elves " in
    let x = "trees " in
    g "cookies "

(c) let rec f n x =
    if n>=0
    then (let _ = print_string x in f (n-1) x)
    else ()
    in
    f 3 "hi "

(d) let rec f n x =
    if n>=0
    then (let _ = print_string x in f (n-1) x)
    else ()
    in
    f 3

(e) let rec f x = f x in
    print_string (f "hi ")

Solution:

(a) hi hi
(b) elves
(c) hi hi hi hi
(d) prints nothing (evaluates to a function that prints when called)
(e) prints nothing (goes into an infinite loop)
4. In this problem, we use the untyped lambda calculus with small-step call-by-value left-to-right evaluation. Recall this encoding of pairs:

“mkpair” \( \lambda x. \lambda y. \lambda z. z \ x \ y \)

“fst” \( \lambda p. p(\lambda x. \lambda y. x) \)

“snd” \( \lambda p. p(\lambda x. \lambda y. y) \)

(a) (9 points) For any values \( v_1 \) and \( v_2 \), “fst” (“mkpair” \( v_1 \ v_2 \)) produces a value in 6 steps. Writing only lambda terms (i.e., no abbreviations), show these steps. Show just the result of each step, not the derivation that produces it.

\[
(\lambda p. (\lambda x. \lambda y. x)) ((\lambda x. \lambda y. \lambda z. z \ x \ y) \ v_1 \ v_2)
\]

\[
→
\]

\[
→
\]

\[
→
\]

\[
→
\]

\[
→
\]

\[
→
\]

(b) (6 points) Again using no abbreviations, extend the encoding to include a “swap” function. Given an encoding of the pair \( (v_1, v_2) \), “swap” should return an encoding of the pair \( (v_2, v_1) \).

Solution:

(a) \( (\lambda p. (\lambda x. \lambda y. x)) ((\lambda x. \lambda y. \lambda z. z \ x \ y) \ v_1 \ v_2) \)

\[
→ (\lambda p. (\lambda x. \lambda y. x)) ((\lambda y. \lambda z. z \ v_1 \ y) \ v_2)
\]

\[
→ (\lambda p. (\lambda x. \lambda y. x)) (\lambda z. z \ v_1 \ v_2)
\]

\[
→ (\lambda z. z \ v_1 \ v_2) (\lambda x. \lambda y. x)
\]

\[
→ (\lambda x. \lambda y. x) \ v_1 \ v_2
\]

\[
→ (\lambda y. \ v_1) \ v_2
\]

\[
→ v_1
\]

(b) There are an infinite number of correct solutions. Here are four:

- \( \lambda p. (\lambda x. \lambda y. \lambda z. z \ x \ y)(p \lambda x. \lambda y. y)(p \lambda x. \lambda y. x) \)
- \( \lambda p. \lambda z. z (p \lambda x. \lambda y. y)(p \lambda x. \lambda y. x) \)
- \( \lambda p. (\lambda x. \lambda y. \lambda z. z \ y \ x)(p \lambda x. \lambda y. y)(p \lambda x. \lambda y. x) \)
- \( \lambda p. p \lambda x. \lambda y. \lambda z. z \ y \ x \)
5. In this problem, we consider the simply-typed lambda-calculus (using small-step call-by-value left-to-right evaluation). We suppose the integer constants $c$ (of type $\text{int}$) include only positive integers (1, 2, 3, ...), i.e., we remove negative numbers. We add a subtraction operator ($e ::= \ldots | e - e$) and these rules:

\[
\begin{align*}
&c_3 \text{ is math's subtraction of } c_2 \text{ from } c_1 \\
&\frac{c_1 - c_2 \rightarrow c_3}{\Gamma \vdash e_1 : \text{int}} \quad \frac{\Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 - e_2 : \text{int}}
\end{align*}
\]

(a) (4 points) Our operational semantics needs two additional rules. Give them.

(b) (6 points) Our language is not type-safe. Demonstrate this.

(c) (10 points) Consider the Preservation Lemma, the Progress Lemma, and the Substitution Lemma. Which of these lemmas are true in our language? Explain your answers briefly, but proofs are not required.

Solution:

(a)

\[
\begin{align*}
&\frac{e_1 \rightarrow e_1'}{e_1 - e_2 \rightarrow e_1' - e_2} \\
&\frac{e_2 \rightarrow e_2'}{v - e_2 \rightarrow v - e_2'}
\end{align*}
\]

(b) Consider an expression like $3 - 4$. It type-checks under the empty context (\cdot) with type $\text{int}$, but it cannot take a step because the result of the mathematical subtraction is -1, which is not in our language, so no rule applies.

(c) The Progress Lemma does not hold; see part (b).

The Preservation Lemma does hold: all three new operational rules produce an expression of type $\text{int}$ assuming the expression before the step type-checks with type $\text{int}$.

The Substitution Lemma does hold (assuming $(e_1 - e_2)[e/x] = (e_1[e/x]) - (e_2[e/x])$); the case of the proof for subtractions expressions follows from induction just like the case for application.