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 17:20.

• 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.
For your reference:

IMP syntax as presented in class:

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

IMP large-step semantics as presented in class:

\[ H ; e \Downarrow c \]
\[ \text{CONST} \quad \text{VAR} \quad \text{ADD} \quad \text{MULT} \]
\[ H ; c \Downarrow c \quad H ; x \Downarrow H(x) \quad \frac{H ; e_1 \Downarrow c_1 \quad H ; e_2 \Downarrow c_2}{H ; e_1 + e_2 \Downarrow c_1 + c_2} \quad \frac{H ; e_1 \Downarrow c_1 \quad H ; e_2 \Downarrow c_2}{H ; e_1 \ast e_2 \Downarrow c_1 \ast c_2} \]

IMP small-step semantics as presented in class:

\[ H_1 ; s_1 \rightarrow H_2 ; s_2 \]
\[ \text{ASSIGN} \quad \text{SEQ1} \quad \text{SEQ2} \quad \text{WHILE} \]
\[ H ; e \Downarrow c \quad H ; \text{skip}; s \rightarrow H ; s \quad H ; s_1 \rightarrow H' ; s_1' \quad H ; \text{while} \ e \ s \rightarrow H ; \text{if} (e ; \text{while} \ e) \text{skip} \]

Call-by-value, left-to-right evaluation language presented in class:

\[ e ::= \lambda x. e \mid x \mid e \mid e \mid c \]
\[ v ::= \lambda x. e \mid c \]

\[ e \rightarrow e' \]
\[ (\lambda x. e) \triangleright e[v/x] \quad \frac{e_1 \rightarrow e_1' \quad e_2 \rightarrow e_2'}{e_1 e_2 \rightarrow e_1' e_2'} \quad \frac{v \rightarrow v'}{v e_2 \rightarrow v' e_2'} \]
\[ e[e'/x] = e'' \]
\[ x[e/x] = e \quad e_1[e/x] = e_1' \quad y \neq x \quad y \notin \text{FV}(e) \]
\[ e_1[e/x] = e_1' \quad (\lambda y. e_1)[e/x] = \lambda y. e_1' \]
\[ y \neq x \quad e_1[e/x] = e_1' \quad e_2[e/x] = e_2' \]
\[ (e_1 e_2)[e/x] = e_1' e_2' \]
1. (IMP with booleans)

In this problem we extend the IMP expression language with booleans: true, false, negation, and inclusive-or. (Variables hold integers or booleans, but that is not directly relevant to the questions below.) The new syntax forms are:

$$ e ::= \ldots \mid \text{true} \mid \text{false} \mid \neg e \mid e \lor e $$

The result of evaluating an expression can be an integer (not relevant below), true, or false. That is, we have $H; e \Downarrow v$ where $v ::= c \mid \text{true} \mid \text{false}$.

Negation and inclusive-or can be “stuck” if a subexpression does not evaluate to a boolean.

(a) (10 points) Add rules to our large-step operational semantics to support the new syntax forms. For $e_1 \lor e_2$, use short-circuiting left-to-right evaluation (like $||$ in many languages, i.e., if the first operand evaluates to true, the result is true). If your rules all contain explicit uses of false and true, then you should expect to need a total of 7 rules. Three of these rules are given below, write the remaining 4 rules.

$$
\begin{align*}
H; \text{true} \Downarrow \text{true} & & H; \text{false} \Downarrow \text{false} & & H; e \Downarrow \text{true} \\
H; e_1 \Downarrow \text{true} & & H; e_1 \Downarrow \text{false} & & H; e_2 \Downarrow \text{true} & & H; e_1 \Downarrow \text{false} & & H; e_2 \Downarrow \text{false}
\end{align*}
$$

(b) (10 points) Theorem: If $e$ always evaluates to a boolean, then $e$ and $\neg \neg e$ are equivalent.

- Restate this theorem formally.
- Prove this theorem formally (hint: you do not need induction; use derivation based on the large-step rules).

(c) (10 points) Add implication ($e$ implies $e$) to the language. Recall “a implies b if a is false or b is true.” For example, the following IMP expressions all evaluate to true: $e_1 = \text{true implies true}$, $e_2 = \text{false implies true}$, and $e_3 = \text{true} \lor \text{false} \Rightarrow \text{true}$. On the other hand, the following IMP expressions evaluate to false: $e_4 = \text{true implies false}$ and $e_5 = \text{true} \lor \text{false} \Rightarrow \text{false}$.

- Give large-step operational semantics rules that support this extension “directly,” using short-circuiting left-to-right evaluation. If your rules all contain explicit uses of false and true, then you should expect to write 3 rules.
- Give 1 rule that works just as well as your 3 rules by treating implication as a derived form (i.e., in the rule’s hypothesis, define implies by using other IMP operations, e.g., $\neg$ and $\lor$ on some expressions $e_1$ and $e_2$). Remember this should be a large-step rule. Use $v$ in this rule as the result of the evaluation.

Solution:

(a)

$$
\begin{align*}
H; \text{true} \Downarrow \text{true} & & H; \text{false} \Downarrow \text{false} & & H; e \Downarrow \text{true} & & H; e \Downarrow \text{false} \\
H; e_1 \Downarrow \text{true} & & H; e_1 \Downarrow \text{false} & & H; e_2 \Downarrow \text{true} & & H; e_1 \Downarrow \text{false} & & H; e_2 \Downarrow \text{false} & & H; e_1 \lor e_2 \Downarrow \text{true} & & H; e_1 \lor e_2 \Downarrow \text{false}
\end{align*}
$$
(b) Assume for all $H$ that $H; e \Downarrow \text{true}$ or $H; e \Downarrow \text{false}$. Then for all $H, e$, and $v$, $H; e \Downarrow v$ if and only if $H; \neg e \Downarrow v$. We prove the two directions separately. First assume $H; e \Downarrow v$. Because $v$ is true or false, one of these derivations suffices to show $H; \neg \neg e \Downarrow v$:

\[
\begin{align*}
H; e \Downarrow \text{true} & \quad \quad H; e \Downarrow \text{false} \\
H; \neg e \Downarrow \text{false} & \quad \quad H; \neg e \Downarrow \text{true} \\
H; \neg \neg e \Downarrow \text{true} & \quad \quad H; \neg \neg e \Downarrow \text{false}
\end{align*}
\]

Now assume $H; \neg e \Downarrow v$. By inspection of the semantic rules, $v$ is true or false. In the former case, we can derive this only if $H; \neg e \Downarrow \text{false}$, which in turn we can derive only if $H; e \Downarrow \text{true}$, as desired. Similarly, in the latter case, we can derive this only if $H; \neg e \Downarrow \text{true}$, which in turn we can derive only if $H; e \Downarrow \text{false}$, as desired.

(c)

\[
\begin{align*}
H; e_1 \Downarrow \text{false} & \quad \quad H; e_1 \Downarrow \text{true} \quad H; e_2 \Downarrow \text{true} \\
H; e_1 \Rightarrow e_2 \Downarrow \text{true} & \quad \quad H; e_1 \Rightarrow e_2 \Downarrow \text{true} \\
H; (\neg e_1) \lor e_2 \Downarrow v & \quad \quad H; e_1 \Rightarrow e_2 \Downarrow \text{false} \\
H; e_1 \Rightarrow e_2 \Downarrow \text{false}
\end{align*}
\]
2. (12 points) (IMP with large-step semantics)

We can give IMP statements a large-step semantics with a judgment of the form \( H; s \Downarrow H' \). The rules below do so, but there are errors. (The rules match neither our informal understanding nor our small-step semantics.) Find three errors (two of which are the same conceptual error), explain the problem, why it is a problem, and how to change the rules to solve the problem.

Solution:

In the sequence rule, we evaluate the \( s_2 \) using the original heap. This “forgets” any assignments that \( s_1 \) may have done. The fix is to evaluate \( s_2 \) using \( H_1 \), as in this rule:

\[
\text{SEQ} \\
H; s_1 \Downarrow H_1 \quad H_1; s_2 \Downarrow H_2 \\
\hdashline
H; (s_1; s_2) \Downarrow H_2
\]

In the if rules, we evaluate both statements. This is incorrect if the “branch not taken” has an infinite loop (or could otherwise get stuck, but in IMP there are no stuck states). For example, we cannot derive \( H; \text{if } \text{skip} (\text{while } \text{skip}) \Downarrow H \) but we would like to. This is how to fix the rules:

\[
\text{IF1} \\
H; e \Downarrow c \\
H; \text{if } e \Downarrow H_1 \\
\hdashline
H; \text{if } \text{if } e \Downarrow H_1
\]

\[
\text{IF2} \\
H; e \Downarrow c \\
H; s_1 \Downarrow H_1 \\
H; s_2 \Downarrow H_2 \\
\hdashline
H; \text{if } e \Downarrow H_1
\]

In the while rule, we evaluate the expression \( e \) twice. This is incorrect if \( e \) does not converge, but in IMP there are no non-convergent expressions. For example, we cannot derive \( H; \text{while } \text{e } \text{e } \Downarrow H' \) but we would like to. This is how to fix the rules:

\[
\text{WHILE} \\
H; (e; \text{while } e) \Downarrow H' \\
\hdashline
H; \text{while } e \Downarrow H'
\]
3. (IMP with small-step semantics) Consider the original definition of IMP without any expression extensions. In this problem we extend IMP statements with the construct `repeat c s`. Informally, the idea is to execute `s` `c` times.

(a) (10 points) Here are two separate ways one might add rules to the semantics:

- First way:

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

- 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.

i. Which way is wrong? Explain why it is wrong.

ii. Show how to change the wrong way to make it correct.

Solution:

i. The second way is wrong; it always executes `s` at least one time. If `c \leq 0`, it should not execute `s` any times.

ii. 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; (s; \text{if } (c - 1) \ (\text{repeat } (c - 1) \ s) \ \text{skip})
```
(b) (10 points) For the following input, show the small-step derivations until termination (i.e., \( H; \text{skip} \) is produced). What is the final value of \( y \)?

\[ \text{x := 2; y := 0; repeat 2 y := y + x} \]

**Solution:**

\[
\begin{align*}
\text{x := 2; y := 0; repeat 2 y := y + x} & \quad \text{[ASSIGN]} \\
\rightarrow \quad ., \text{x} \mapsto 2; \text{skip; y} \mapsto 0; \text{repeat 2 y := y + x} & \quad \text{[SEQ1]} \\
\rightarrow \quad ., \text{x} \mapsto 2; \text{y} \mapsto 0; \text{repeat 2} \quad y := y + x & \quad \text{[REPEAT]} \\
\rightarrow^2 \quad ., \text{x} \mapsto 2; \text{y} \mapsto 0; \text{repeat 2} \quad y := y + x & \quad \text{[ASSIGN,SEQ1]} \\
\rightarrow \quad ., \text{x} \mapsto 2; \text{y} \mapsto 0; \text{y} := y + x; \text{repeat 1} \quad y := y + x & \quad \text{[REPEAT]} \\
\rightarrow^2 \quad ., \text{x} \mapsto 2; \text{y} \mapsto 0; \text{y} := y + x; \text{repeat 1} \quad y := y + x & \quad \text{[ASSIGN,SEQ1]} \\
\rightarrow \quad ., \text{x} \mapsto 2; \text{y} \mapsto 0; \text{y} := y + x; \text{repeat 0} \quad y := y + x & \quad \text{[REPEAT]} \\
\rightarrow^2 \quad ., \text{x} \mapsto 2; \text{y} \mapsto 0; \text{y} := y + x; \text{repeat 0} \quad y := y + x & \quad \text{[ASSIGN,SEQ1]} \\
\rightarrow \quad ., \text{x} \mapsto 2; \text{y} \mapsto 0; \text{y} := y + x; \text{skip} & \quad \text{[REPEAT]} \\
\end{align*}
\]

The final value of \( y \) is 4.

(c) (8 points) Consider IMP with the repeat \( c \) \( s \) extension. Prove that there is at least one case when the repeat loop will not terminate (hint: you do not need induction).

**Solution:**

**Proof:** By example, consider the derivation for \( s = \text{while 1 skip} \). Then \( H; \text{repeat c while 1 skip} \rightarrow H; \text{while 1 skip; repeat c - 1 while 1 skip} \). It suffices to show that \( s \) does not terminate.

\[
\begin{align*}
\text{while 1 skip} & \\
\rightarrow H; \text{if 1 (while 1 skip) skip} & \\
\rightarrow H; \text{while 1 skip} & \\
\ldots &
\end{align*}
\]
4. (15 points) (Caml and functional programming)
Consider this Caml code, which type-checks and runs correctly.

```caml
type dumbTree = Empty | Node of dumbTree * dumbTree

let rec s f t =
  match t with
  Empty -> f t
| Node(x,y) -> f t + s f x + s f y

let c1 t = s (fun x -> 1) t
let c2 t = s (fun x -> match x with Node(l,Empty) -> 1 | _ -> 0) t
```

(a) What are the types of \( s \), \( c1 \), and \( c2 \)?

(b) What do \( c1 \) and \( c2 \) compute? (Hint: The answers are straightforward.)

(c) Rewrite the last two lines of the code so they are shorter and equivalent.

Solution:

(a) \( \text{val } s : (\text{dumbTree \to int}) \to \text{dumbTree \to int} \)
\( \text{val } c1 : \text{dumbTree \to int} \)
\( \text{val } c2 : \text{dumbTree \to int} \)

(b) \( c1 \) takes a tree and returns the total number of internal nodes plus leaves it has. \( c2 \) takes a tree and returns how many of its internal nodes have right-children that are leaves (i.e., \text{Empty}).

(c) \( \text{let } c1 = s (\text{fun } x \to 1) \)
\( \text{let } c2 = s (\text{fun } x \to \text{match } x \text{ with } \text{Node}(l,\text{Empty}) \to 1 \mid _ \to 0) \)
5. 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.

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

\[
\rightarrow
\]

\[
\rightarrow
\]

\[
\rightarrow
\]

\[
\rightarrow
\]

\[
\rightarrow
\]

\[
\rightarrow
\]

\[
\rightarrow
\]

(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)

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

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

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

\[
\rightarrow (\lambda y. v_1) v_2
\]

\[
\rightarrow v_1
\]

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

- \( \lambda p. (\lambda x. \lambda y. \lambda z. z x y)(p \lambda x. \lambda y. y)(p \lambda x. \lambda y. x) \)
- \( \lambda p. \lambda z. z (p \lambda x. \lambda y. y)(p \lambda x. \lambda y. x) \)
- \( \lambda p. (\lambda x. \lambda y. \lambda z. z y x)(p \lambda x. \lambda y. x)(p \lambda x. \lambda y. y) \)
- \( \lambda p. \lambda x. \lambda y. \lambda z. z y x \)
Name:________________________________________

(This page intentionally blank.)