You have 2 hours to complete this exam.

This is a closed-notes exam.

Do not share anything with other students. Do not talk to other students. Do not look other students’ exams. Do not expose your exam to easy viewing by other students. Violation of any of these rules will count as cheating.

If you believe there is an error or an ambiguous question, you may seek clarification from the instructor. Please speak quietly or write your question out.

Including this cover sheet and rules at the end, there are 11 pages to the exam. Once the exam begins, please verify that you have all 11 pages.

Please write your name and NetID in the spaces above.

Show your work. Partial credit will be given for incomplete answers.

The pages at the end of the exam contain definitions and inference rules for various systems. You may detach these pages. If you do, please turn them in with the rest of your exam.

If you finish with time remaining, check your work!
<table>
<thead>
<tr>
<th>Question</th>
<th>Points</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td><strong>Total:</strong></td>
<td><strong>100</strong></td>
<td></td>
</tr>
</tbody>
</table>
Problem 1. (16 points)
Suppose the type of arithmetic expressions has been defined as follows:

\[
\text{Inductive aexp :=}
\]
| ANum (n : nat) |
| AVar (x : string) |
| AScc (a : aexp) |
| APlus (a1 a2 : aexp). |

(a) (5 points) Write a function \texttt{used_vars} of type \texttt{aexp -> list string} that takes an \texttt{aexp} and returns the list of variables that are used in it.

Solution:
\[
\text{Fixpoint used_vars (a : aexp) : list string :=}
\]
match a with
| ANum n => [] |
| AVar x => [x] |
| AScc a = used_vars a |
| APlus a1 a2 => used_vars a1 ++ used_vars a2 |
end.

(b) (4 points) Suppose you were given a function \texttt{replace} of type \texttt{string -> nat -> aexp -> aexp} that takes a variable name, a number, and an \texttt{aexp} and replaces all occurrences of the variable with the number provided. For example, \texttt{replace "x" 5 (APlus (AVar "x") (ANum 3))} would return \texttt{APlus (ANum 5) (ANum 3)}. Using the \texttt{used_vars} function you defined, formally state a theorem that says that for all expressions \(a\), if a variable \(x\) is not used in \(a\), then calling \texttt{replace} with \(x\) on \(a\) returns the same \(a\). You may use the predicate \texttt{In} to assert that a string appears in a list of strings.

Solution:
\[
\text{forall a x, \neg \text{In} x (\text{used_vars} a) -> replace x n a = a}
\]

(c) (7 points) Suppose you decided to prove the theorem above by induction on the expression \(a\). For each case of the proof by induction, write out the goal for that case and any inductive hypotheses, either informally or formally. You only have to state the cases, not prove them.

Solution:
\[
\text{ANum case: no IHs. Goal: \neg \text{In} x [] -> replace x n (ANum m) = ANum m}
\]
\[
\text{AVar case: no IHs. Goal: \neg \text{In} x [y] -> replace x n (AVar y) = AVar y}
\]
\[
\text{AScc case: IH: \neg \text{In} x (\text{used_vars} a) -> replace x n a = a}
\]
Goal: \neg \text{In} x (\text{used_vars} a) -> replace x n (AScc a) = AScc a
\[
\text{APlus case: IHs: \neg \text{In} x (\text{used_vars} a1) -> replace x n a1 = a1 and}
\neg \text{In} x (\text{used_vars} a2) -> replace x n a2 = a2
\]
Goal: \neg \text{In} x (\text{used_vars} a1 ++ \text{used_vars} a2) -> replace x n (APlus a1 a2) = APlus a1 a2
Problem 2. (15 points)
Consider the following proof state.

\[
\begin{align*}
&\text{b : bool} \\
&\text{n : nat} \\
&\text{H1 : n = 4 \lor n = 6} \\
&\text{H2 : False} \\
&\text{H3 : exists m, n = 2 \cdot m} \\
&\text{-----------------------------} \\
&(\text{if b then 4 else 6}) = n
\end{align*}
\]

Suppose you were to call the \texttt{destruct} with each of the following arguments (separately, not one after the other). For each argument, write the number of subgoals that will be left after \texttt{destructing} it, and any changes to the hypotheses and/or conclusion in each of those subgoals.

- **b**

  \textbf{Solution}: b is replaced with \texttt{true} in one subgoal and \texttt{false} in the other.

- **n**

  \textbf{Solution}: n is replaced with \texttt{0} in one subgoal and \texttt{S n'} in the other.

- **H1**

  \textbf{Solution}: H1 becomes \texttt{n = 4} in one subgoal and \texttt{n = 6} in the other.

- **H2**

  \textbf{Solution}: No more subgoals.

- **H3**

  \textbf{Solution}: One subgoal, we introduce a new \texttt{nat m} and H3 becomes \texttt{n = 2 \cdot m}.
Problem 3. (18 points)
Suppose we are defining a programming language with the following abstract syntax:

\[
\text{Inductive exp :=}
\begin{align*}
| & \text{Literal (s : string)} \\
| & \text{Concat (e1 e2 : exp)} \\
| & \text{Longer (e1 e2 : exp)} \\
| & \text{First (e : exp)}.
\end{align*}
\]

Every expression is intended to evaluate to a string value. The intended behavior of the language is as follows:

- **Literal** \(s\) evaluates to \(s\).
- **Concat** \(e_1 e_2\) evaluates to the string made by appending the value of \(e_2\) onto the end of the value of \(e_1\).
- **Longer** \(e_1 e_2\) evaluates to the value of \(e_1\) if it is longer than the value of \(e_2\), and the value of \(e_2\) otherwise.
- **First** \(e\) evaluates to the one-character string that is the first character of the value of \(e\), or the empty string "" if the value of \(e\) is empty.

(a) (8 points) Write an inductive relation \(\text{eval}\) of type \(\text{exp} \to \text{string} \to \text{Prop}\) such that \(\text{eval} e s\) holds exactly when the value of \(e\) is \(s\). You may assume the existence of functions \(\text{append}\) for appending two strings, \(\text{length}\) for getting the length of a string (the length of the empty string is 0), and \(\text{hd}\) for getting the first element of a string. If you use any other helper functions, make sure to explain what they do.

**Solution:**

\[
\text{Inductive eval : exp -> string -> Prop :=}
\begin{align*}
| & \text{eval_lit s : eval (Literal s) s} \\
| & \text{eval_concat e1 e2 s1 s2 : eval e1 s1 -> eval e2 s2 ->}
  \quad \text{eval (Concat e1 e2) (append s1 s2)} \\
| & \text{eval_longer1 e1 e2 s1 s2 : eval e1 s1 -> eval e2 s2 -> length s1 > length s2 ->}
  \quad \text{eval (Longer e1 e2) s1} \\
| & \text{eval_longer2 e1 e2 s1 s2 : eval e1 s1 -> eval e2 s2 -> length s1 <= length s2 ->}
  \quad \text{eval (Longer e1 e2) s2} \\
| & \text{eval_first0 e : eval e "" -> eval (First e) ""} \\
| & \text{eval_first e s : eval e s -> s <> "" -> eval (First e) (hd s)}.
\end{align*}
\]

(b) (10 points) Two expressions in this language are said to be *equivalent* if they evaluate to the same value. Which of the following statements are true? For full credit, explain your reasoning.

- **First (Literal "")** is equivalent to **Literal ""**.

  **Solution:** True. First of "" is defined to be "".

- For all expressions \(e\), **Longer e e** is equivalent to **e**.

  **Solution:** True. The two arguments have the same length, so **Longer** will return the second one, which is \(e\).

- For all expressions \(e_1\) and \(e_2\), **Longer e1 e2** is equivalent to **Longer e2 e1**.

  **Solution:** False. In particular, when the values of \(e_1\) and \(e_2\) have the same length but are not the same string, the first will return \(e_2\) while the second will return \(e_1\).

- For all expressions \(e_1\) and \(e_2\), **Longer e1 (Concat e1 e2)** is equivalent to **Concat e1 e2**.

  **Solution:** True. **Concat e1 e2** will always be at least as long as **e1**, so **Longer** will return its second argument.
- For all expressions $e_1$ and $e_2$, $\text{First} \ (\text{Concat} \ e_1 \ e_2)$ is equivalent to $\text{First} \ e_1$.

   **Solution:** False. If $e_1$ is empty, this will compute $\text{First} \ e_2$ instead.
Problem 4. (20 points)
The semantics and type system of simply-typed lambda calculus are given in Appendices A and B respectively.

(a) (4 points) Does the term $((\lambda x. \lambda y. \text{test } x y \text{ fls}) \text{ fls}) \text{ tru}$ evaluate to a value? If so, what is that value?

**Solution:** Yes, fls.

(b) (4 points) State the progress and preservation theorems for simply-typed lambda calculus.

**Solution:**

Progress: $\forall e T, \emptyset \vdash e : T \Rightarrow value e \lor \exists e', e \rightarrow e'$

Preservation: $\forall e T e', \emptyset \vdash e : T \Rightarrow e \rightarrow e' \Rightarrow \emptyset \vdash e' : T$

(c) (6 points) Would adding the step rule $\text{test } e e_1 e_2 \rightarrow e$ invalidate progress and/or preservation? Explain why, including a specific counterexample for each invalidated property.

**Solution:**

It would invalidate preservation. For instance, $\text{test } \text{ tru } (\lambda x : \text{ Bool } x) (\lambda x : \text{ Bool } x)$ is of type $\text{ Bool } \rightarrow \text{ Bool }$, but it would step to $\text{ tru }$, which is of type $\text{ Bool }$.

(d) (6 points) Would adding the typing rule $\Gamma \vdash e_1 : \text{ Bool } \Gamma \vdash e_2 : \text{ Bool } \Gamma \vdash (e_1, e_2) : \text{ Bool }$ invalidate progress and/or preservation? Explain why, including a specific counterexample for each invalidated property.

**Solution:**

It would invalidate progress. For instance, $\text{ tru } \text{ fls }$ would have type $\text{ Bool }$, but it is not a value and cannot take a step.
Problem 5. (16 points)
The semantics of Imp with memory are given in Appendix C.
(a) (5 points) What is the next state that the following configuration steps to?

\[
\ast x ::= y + 1 ;; z ::= \ast y / (x \mapsto 10, y \mapsto 11, z \mapsto 12), (10 \mapsto 1, 11 \mapsto 2, 12 \mapsto 3)
\]

Solution:

\[
\text{SKIP ;; } z ::= \ast y / (x \mapsto 10, y \mapsto 11, z \mapsto 12), (10 \mapsto 12, 11 \mapsto 2, 12 \mapsto 3)
\]

(b) (5 points) What is the final state and memory reached once the program above steps to SKIP?

Solution:

\[
(x \mapsto 10, y \mapsto 11, z \mapsto 2), (10 \mapsto 12, 11 \mapsto 2, 12 \mapsto 3)
\]

(c) (6 points) Suppose we wanted to add a new operation \texttt{trystore} such that \texttt{trystore }e_1 e_2 does nothing (i.e., steps to SKIP with no other effects) if \(e_1\) points to location 0, and otherwise has the same behavior as \(\ast e_1 ::= e_2\). Define the behavior of \texttt{trystore} by giving one or more small-step semantic rules for it.

Solution:

\[
\begin{align*}
\text{aeval st } e_1 = 0 & \quad \text{trystore } e_1 e_2 / st, m \rightarrow \text{SKIP} / st, m \\
\text{aeval st } e_1 = \ell & \quad \ell \neq 0 \\
\text{trystore } e_1 e_2 / st, m \rightarrow \text{SKIP} / st, m(\ell \mapsto v)
\end{align*}
\]
Problem 6. (15 points)
Consider the following program:

\[
\begin{align*}
*W & ::= 3;; \\
*X & ::= 1;; \\
\text{WHILE } Y > 0 \text{ DO} & \\
& \quad T ::= *X;; \\
& \quad *X ::= T * 2;; \\
& \quad Y ::= Y - 1 \\
\text{END}
\end{align*}
\]

(a) (5 points) What does this program do? Write an informative pre- and postcondition for the program.

**Solution:** It sets the value at \(W\) to 3 and the value at \(X\) to \(2^Y\).

Precondition: \(\{W \mapsto a \* X \mapsto b \* Y = c\}\)
Postcondition: \(\{W \mapsto 3 \* X \mapsto 2^c\}\)

(b) (10 points) The rules of separation logic for Imp with memory are given in Appendix D. Using your pre- and postcondition from the previous part, decorate the program with separation logic assertions to prove its correctness. You will need to come up with an invariant for the loop. Make sure to write all necessary implications and check that they hold.

**Solution:**

\[
\begin{align*}
\{W \mapsto a \* X \mapsto b \* Y = c\} & \\
*W & ::= 3;; \\
\{W \mapsto 3 \* X \mapsto b \* Y = c\} & \\
*X & ::= 1;; \\
\{W \mapsto 3 \* X \mapsto 1 \* Y = c\} & \rightarrow \{W \mapsto 3 \* X \mapsto 2^{c-Y}\} \\
\text{WHILE } Y > 0 \text{ DO} & \\
& \quad T ::= *X;; \\
& \quad \{W \mapsto 3 \* X \mapsto 2^{c-Y} \land Y > 0\} \\
& \quad \{W \mapsto 3 \* X \mapsto 2^{c-Y} \land Y > 0 \land T = 2^{c-Y}\} \\
& \quad \{W \mapsto 3 \* X \mapsto T \* 2 \land 2 \land Y > 0 \land T = 2^{c-Y}\} & \rightarrow \{W \mapsto 3 \* X \mapsto 2^{c-(Y-1)}\} \\
& \quad Y ::= Y - 1 \\
& \quad \{W \mapsto 3 \* X \mapsto 2^{c-Y}\} \\
\text{END} & \\
& \quad \{W \mapsto 3 \* X \mapsto 2^{c-Y} \land Y = 0\} & \rightarrow \{W \mapsto 3 \* X \mapsto 2^c\}
\end{align*}
\]
A  Small-Step Semantics of Simply-Typed Lambda Calculus

\[ e \rightarrow e' \] means that the \( \lambda \)-term \( e \) steps to the term \( e' \) in one step. Values are either abstractions \( \lambda x : T, e \) or boolean values \( \text{tru} \) or \( \text{fls} \). In the following rules, any term named \( v \) must be a value, while other terms may not be values.

\[
\frac{(\lambda x : T, e) v \rightarrow [x := v]e}{e \rightarrow e'}
\]

\[
\frac{e_1 \rightarrow e'_1}{e_1 e_2 \rightarrow e'_1 e_2}
\]

\[
\frac{e_2 \rightarrow e'_2}{v_1 e_2 \rightarrow v_1 e'_2}
\]

\[
\frac{e \rightarrow e'}{test \ e_1 e_2 \rightarrow test \ e' \ e_1 e_2}
\]

B  Type System of Simply-Typed Lambda Calculus

\( \Gamma \vdash e : T \) means that in the type context \( \Gamma \), the expression \( e \) has type \( T \).

\[
\frac{\Gamma(x) = T}{\Gamma \vdash x : T}
\]

\[
\frac{\Gamma(x \mapsto T_1) \vdash e : T}{\Gamma \vdash (\lambda x : T_1, e) : T}
\]

\[
\frac{\Gamma \vdash e_1 : T_1 \Rightarrow T_2 \quad \Gamma \vdash e_2 : T_2}{\Gamma \vdash e_1 \ e_2 : T}
\]

\[
\frac{\Gamma \vdash \text{tru} : \text{Bool}}{\Gamma \vdash e_1 e_2 \rightarrow \text{test} \ e_1 e_2 : T}
\]

\[
\frac{\Gamma \vdash \text{fis} : \text{Bool}}{\Gamma \vdash e_1 e_2 \rightarrow \text{test} \ e'_1 e_1 e_2 : T}
\]

C  Small-Step Semantics of Imp with Memory

\( c / st, m \rightarrow c' / st', m' \) means that the command \( c \), when executed starting in the state \( st \) and memory \( m \), takes a single step to the new command \( c' \) in the state \( st' \) and memory \( m' \).

\[
\frac{\text{aeval} \ st \ a = v}{x ::= a / st, m \rightarrow \text{SKIP} / st(x \mapsto v), m}
\]

\[
\frac{c_1 / st, m \rightarrow c'_1 / st', m'}{c_1 ; c_2 / st, m \rightarrow c'_1 ; c_2 / st', m'}
\]

\[
\frac{\text{beval} \ st \ b = \text{true}}{\text{TEST} \ b \ \text{THEN} \ c_1 \ \text{ELSE} \ c_2 \ \text{FI} / st, m \rightarrow c_1 / st, m}
\]

\[
\frac{\text{beval} \ st \ b = \text{false}}{\text{TEST} \ b \ \text{THEN} \ c_1 \ \text{ELSE} \ c_2 \ \text{FI} / st, m \rightarrow c_2 / st, m}
\]

\[
\frac{\text{WHILE} \ b \ \text{DO} \ c \ \text{END} / st, m \rightarrow \text{TEST} \ b \ \text{THEN} \ c ; \ \text{WHILE} \ b \ \text{DO} \ c \ \text{END} \ \text{ELSE} \ \text{SKIP} \ \text{FI} / st, m}{\text{aeval} \ st \ a = \ell \quad m(\ell) = v}
\]

\[
\frac{\text{aeval} \ st \ a = \ell}{x ::= *a / st, m \rightarrow \text{SKIP} / st(x \mapsto v), m}
\]

\[
\frac{\text{aeval} \ st \ a_2 = v}{*a_1 ::= a_2 / st, m \rightarrow \text{SKIP} / st, m(\ell \mapsto v)}
\]
D  Separation Logic

\{P\} c \{Q\} means that if \( P \) holds before executing \( c \), then \( Q \) will hold after executing \( c \).

\[
\begin{align*}
\{P\} \text{ SKIP } \{P\} & \quad \{Q[x \mapsto a]\} \ x ::= a \ \{Q\} \\
\quad P_1 \Rightarrow P_2 & \quad \{P_2\} \ c \ \{Q_2\} \quad Q_2 \Rightarrow Q_1 \\
\quad \{P_1\} \ c \ \{Q_1\} & \quad \{P \land b\} \ c_1 \ \{Q\} \quad \{P \land \neg b\} \ c_2 \ \{Q\} \\
\quad \{P\} \text{ TEST } b \text{ THEN } c_1 \text{ ELSE } c_2 \text{ FI } \{Q\} & \\
\quad \{P \land (e = \text{true})\} \ c \ \{P\} & \quad \{P \land (e = \text{false})\} \\
\quad \{P\} \text{ WHILE } e \text{ DO } c \text{ END } \{P \land (e = \text{false})\} & \\
\quad x \neq y & \quad \{x \mapsto v\} \ *x ::= e \ \{x \mapsto e\} \\
\quad \{y \mapsto v\} \ x ::= *y \ \{y \mapsto v \land x = v\} & \\
\quad \{x \mapsto v\} \ *x ::= e \ \{x \mapsto e\} & \\
\end{align*}
\]

E  Scratch Space