**CS 476 Fall 2018 Final**

<table>
<thead>
<tr>
<th>Name:</th>
</tr>
</thead>
<tbody>
<tr>
<td>NetID:</td>
</tr>
</tbody>
</table>

- You have **2 hours** to complete this exam.

- This is a **closed-book** 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 be considered 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 13 pages to the exam, including one blank page for workspace. Once the exam begins (and not before!), please verify that you have all 13 pages.

- Please write your name and NetID in the spaces above, and also in the provided space at the top of every sheet.

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

- The pages at the end of the exam contain 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>7</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td><strong>Total:</strong></td>
<td><strong>100</strong></td>
<td></td>
</tr>
</tbody>
</table>

NetID: __________________________
Problem 1. (7 points)
Write an OCaml function \( \text{zip} : \text{int list} \to \text{bool list} \to (\text{int} \times \text{bool}) \text{ list} \) such that each element in the list returned by \( \text{zip} \ l1 \ l2 \) is a pair of an element from \( l1 \) and the corresponding element from \( l2 \). For instance, \( \text{zip} \ [1; 2; 3] \ [true; false; true] \) should return \( [(1, true); (2, false); (3, true)] \). You may assume that the two inputs are always the same length. For full credit, do not use built-in library functions like \text{map}. 

```
let rec zip (l1 : int list) (l2 : bool list) : (int * bool) list =
```
Problem 2. (12 points)
The typing rules for a simple object-oriented language are given in Appendix A. Suppose the following classes are declared in the class table $CT$:

```java
class Square extends Object {
    int side;
    int area() {
        return this.side * this.side;
    }
}

class ColorSquare extends Square {
    int color;
    int getColor() {
        return this.color;
    }
}
```

Write a proof tree for the judgment

$$
\Gamma \vdash c := \text{new ColorSquare}(2, 1); \ x := c.\text{area} : \text{ok}
$$

given that $\Gamma(c) = \text{ColorSquare}$ and $\Gamma(x) = \text{int}$.
Problem 3. (11 points)

(a) (5 points) For each of the following lambda-terms, draw a line from each variable occurrence to the place where it is bound.

- \( \lambda a. \lambda b. a \)
- \( \lambda y. \lambda x. \lambda y. y x y \)
- \( \lambda c. (\lambda d. c d) (\lambda d. d c) \)

(b) (6 points) Consider the lambda-term \( ((\lambda x. x) (\lambda y. y)) ((\lambda x. x) (\lambda y. y)) \).

- Evaluate the term according to call-by-name semantics, in which the argument to a function is not evaluated before the function is applied. Show each step.

- Evaluate the term according to call-by-value semantics, in which the argument to a function is fully evaluated before the function is applied. Show each step.
Problem 4. (10 points)
Consider the following OCaml program:

```ocaml
let y = 3;;
let z = true;;
let g x = if z then x else y;;
let z = false;;
g 1;;
```

(a) (4 points) What value does the program return?

(b) (6 points) What is the value of \( g \)?

Problem 5. (12 points)
In a simple functional language, the type inference rule for function application is:

\[
\frac{\Gamma \vdash e_1 : \tau_1 \mid C_1 \quad \Gamma \vdash e_2 : \tau_2 \mid C_2 \quad \tau \text{ fresh}}{\Gamma \vdash e_1 \ e_2 : \{\tau_1 = \tau_2 \rightarrow \tau\} \cup C_1 \cup C_2}
\]

Suppose we wanted to add an operator \( \cdot \) to the language such that \( f \cdot g \) composes the functions \( f \) and \( g \), that is, \((f \cdot g)\ x = f (g \ x)\). If \( f \) is of type \( b \rightarrow c \) and \( g \) is of type \( a \rightarrow b \), then \( f \cdot g \) is of type \( a \rightarrow c \). Write the type inference rule for the expression \( e_1 \cdot e_2 \). Remember that the types assigned to expressions in the premises must be unknown type variables (\( \tau_1, \tau_2, \) etc.) and not specific types (int, \( a \rightarrow b \), etc.).
Problem 6. (12 points)
The core algorithm of logic programming maintains a state with four components: a list $gs$ of goals to prove, a set $R$ of rules that have not yet been tried on the current goal, a substitution $\sigma$ that holds the solution to the query, and a backtracking stack $k$. The algorithm proceeds as follows:

1. Let the current goal be the first goal of $gs$, which we will call $g$.
2. Choose a rule $r$ from the set $R$ to apply to $g$.
3. Make a fresh copy of $r$ so that none of its variables clash with the variables of $gs$. Call the conclusion of this fresh copy $t$ and its premises $t_1, \ldots, t_n$.
4. Unify the conclusion $t$ with the goal $g$, obtaining a substitution $\sigma_1$.
5. Make the new state of the algorithm as follows:
   - Make the new list of goals by applying $\sigma_1$ to the list $t_1, \ldots, t_n$ followed by the rest of $gs$.
   - Change the set of rules not yet tried to $R_0$, the full set of rules from the original program.
   - Make the new substitution by composing $\sigma_1$ with the old substitution $\sigma$.
   - Make the new stack by adding the frame $(gs, R \setminus \{r\}, \sigma)$ to the top of the old stack $k$.

Suppose you were implementing an interpreter for a logic language by writing a function `logic_step` that takes arguments $rs0$, $gs$, $rs$, $s$, and $k$, corresponding to $R_0$, $gs$, $R$, $\sigma$, and $k$ respectively. Fill in the case of `logic_step` that executes the above algorithm. When choosing a rule to apply, you can simply choose the first element of $rs$. You do not need to provide code for the cases in which this process fails, for instance because there are no goals left or because unification fails. You may assume the existence of the following helper functions: `make_fresh`, which makes a fresh copy of a rule; `unify`, which performs unification; `apply_subst`, which applies a substitution to a list of goals; and `compose_subst`, which takes two substitutions and composes them.

```ml
let logic_step (rs0 : rule list) (gs : term list) (rs : rule list) (s : substitution) (k : stack)
  : term list * rule list * substitution * stack =
```
Problem 7. (12 points)

(a) (4 points) Give an informative precondition and postcondition for the following program:

\[
\text{if } x > y \text{ then } z := x \text{ else } z := y
\]

(b) (8 points) The proof rules for Floyd-Hoare Logic are given in Appendix B. Build a proof tree showing that the program satisfies the precondition and postcondition you gave in part (a). Make sure to check any necessary implications.
Problem 8. (12 points)
The operational semantics for a simple concurrent programming language with synchronous communication are given in Appendix C.

(a) (4 points) What is the next step that the configuration \(\text{send}(x, \{x = 2\}) \parallel (y := \text{recv()}, \{y = 1\})\) takes?

(b) (8 points) Construct a proof tree justifying your answer to (a).
Problem 9. (12 points)

The semantic rules for memory load and store operations in a simple assembly language are as follows:

\[
\begin{align*}
\text{\textbf{C}}(p) &= (x = \text{\textbf{load}} \ e) \quad (e, \rho) \Downarrow \ell \quad \sigma(\ell) = v \\
\text{\textbf{C}}, L \vdash (p, \rho, \sigma) &\rightarrow (p + 1, \rho[x \mapsto v], \sigma)
\end{align*}
\]

\[
\begin{align*}
\text{\textbf{C}}(p) &= (\text{\textbf{store}} \ e_1, e_2) \quad (e_1, \rho) \Downarrow v \quad (e_2, \rho) \Downarrow \ell \\
\text{\textbf{C}}, L \vdash (p, \rho, \sigma) &\rightarrow (p + 1, \rho, \sigma[\ell \mapsto v])
\end{align*}
\]

Suppose we wanted to add a compare-and-exchange command \texttt{cmpxchg} to the language, such that \(x = \texttt{cmpxchg} \ e_1, e_2, e_3\) does the following:

1. Evaluates \(e_3\) to a memory location \(\ell\).
2. Compares the value of \(e_1\) with the value in memory at \(\ell\).
3. If the two values are equal, stores the value of \(e_2\) at \(\ell\) and sets \(x\) to 1. If the two values are not equal, leaves the memory unchanged and sets \(x\) to 0.

Write one or more inference rules giving the semantics of \texttt{cmpxchg}. (Hint: it may be easiest to give two rules.)
A Typing Rules for a Simple Object-Oriented Language

\[
\begin{align*}
\frac{C <: C}{CT(C) = \text{class } C \text{ extends } D \{ \cdots \}} \quad & \quad \frac{C <: D}{C <: E} \\
\frac{(n \text{ is a number})}{\Gamma \vdash n : \text{int}} \quad & \quad \frac{(b \text{ is a boolean})}{\Gamma \vdash b : \text{bool}} \quad & \quad \frac{(\Gamma(x) = \tau)}{\Gamma \vdash x : \tau} \quad & \quad \frac{\Gamma \vdash e : \tau_1 \quad \tau_1 <: \tau_2}{\Gamma \vdash e : \tau_2} \\
\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{bool}}{\Gamma \vdash e_1 \oplus e_2 : \text{bool}} \quad & \quad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 = e_2 : \text{bool}} \quad & \quad \frac{\Gamma \vdash e_1 : \text{C}}{(\text{fields}(CT,C) = \ldots, \tau f_1, \ldots)} \quad & \quad \frac{\Gamma \vdash e_1 : \text{C}}{\Gamma \vdash e.f : \tau} \\
\frac{\Gamma \vdash e_1 : \text{C} \quad \Gamma \vdash e_2 : \text{C}}{\Gamma \vdash e_1 = e_2 : \text{bool}} \quad & \quad \frac{\Gamma \vdash e : C}{\text{methods}(CT,C) = \ldots, \tau m(x_1, \ldots, x_n) \{ \ldots \} \ldots} \quad & \quad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash x := e.m(x_1, \ldots, x_n) : \text{ok}} \\
\frac{\Gamma \vdash e : \text{C} \quad (\text{fields}(CT,C) = \ldots, \tau f_1, \ldots) \Gamma \vdash e_1 : \tau_1 \ldots \Gamma \vdash e_n : \tau_n}{\Gamma \vdash x := e.m(x_1, \ldots) : \text{ok}} \quad & \quad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 \oplus e_2 : \text{bool}} \\
\end{align*}
\]

B Floyd-Hoare Logic for a Simple Imperative Language

\[
\begin{align*}
\{P\} \text{skip} \{P\} \quad & \quad \{P\} \{Q\} \quad \{Q\} \{R\} \quad \{[x \mapsto e]\} \ x := e \{P\} \\
\{P\} \{Q\} \quad & \quad \{P \land (e = \text{true})\} \ c \{Q\} \quad \{P \land (e = \text{false})\} \ c \{Q\} \quad \{P \land (e = \text{false})\} \ c \{P\} \\
\{P\} \quad & \quad \{P\ \text{if} \ e \ \text{then} \ c_1 \ \text{else} \ c_2 \ \{Q\}\} \quad \{P\ \text{while} \ e \ \text{do} \ c \ \{P \land (e = \text{false})\}\} \\
\end{align*}
\]
C  Operational Semantics for a Simple Concurrent Language

\[
\begin{align*}
\text{(n is a number)} & \quad (n, \sigma) \Downarrow n \\
\text{(b is a boolean)} & \quad (b, \sigma) \Downarrow b \\
\text{(m = v)} & \quad (x, \sigma) \Downarrow v
\end{align*}
\]

\[
\begin{align*}
(e_1, \sigma) \Downarrow v_1 & \quad (e_2, \sigma) \Downarrow v_2 & \quad (v_1 \oplus v_2 = v) \\
(e_1 \oplus e_2, \sigma) \Downarrow v
\end{align*}
\]

where \( \oplus \) is an arithmetic, comparison, or boolean operator

\[
\begin{align*}
(e, \sigma) \Downarrow v \\
(x := e, \sigma) \rightarrow \text{(skip, } \sigma[x \mapsto v])
\end{align*}
\]

\[
\begin{align*}
(e_1, \sigma) \rightarrow (e'_1, \sigma'_1) & \quad (e_2, \sigma) \rightarrow (e'_2, \sigma'_2) \\
(e_1, \sigma_1) \parallel (e_2, \sigma_2) \rightarrow (e'_1, \sigma'_1) \parallel (e'_2, \sigma'_2)
\end{align*}
\]

\[
\begin{align*}
(e_1, \sigma_1) \rightarrow (e'_1, \sigma'_1) & \quad (e_2, \sigma_2) \rightarrow (e'_2, \sigma'_2) \\
(e_1, \sigma_1) \parallel (e_2, \sigma_2) \rightarrow (e'_1, \sigma'_1) \parallel (e'_2, \sigma'_2)
\end{align*}
\]

\[
\begin{align*}
(x := \text{recv}(\cdot), \sigma) \xrightarrow{\text{in}(v)} \text{(skip, } \sigma[x \mapsto v])
\end{align*}
\]

\[
\begin{align*}
(e_2, \sigma_2) \rightarrow (e'_2, \sigma'_2) \\
(e_1, \sigma_1) \parallel (e_2, \sigma_2) \rightarrow (e_1, \sigma_1) \parallel (e'_2, \sigma'_2)
\end{align*}
\]

\[
\begin{align*}
(e_1, \sigma_1) \xrightarrow{\text{in}(v)} (e'_1, \sigma'_1) & \quad (e_2, \sigma_2) \xrightarrow{\text{out}(v)} (e'_2, \sigma'_2) \\
(e_1, \sigma_1) \parallel (e_2, \sigma_2) \rightarrow (e'_1, \sigma'_1) \parallel (e'_2, \sigma'_2)
\end{align*}
\]

\[
\begin{align*}
(e_1, \sigma_1) \xrightarrow{\text{out}(v)} (e'_1, \sigma'_1) & \quad (e_2, \sigma_2) \xrightarrow{\text{out}(v)} (e'_2, \sigma'_2) \\
(e_1, \sigma_1) \parallel (e_2, \sigma_2) \rightarrow (e'_1, \sigma'_1) \parallel (e'_2, \sigma'_2)
\end{align*}
\]
D Scratch Space