You have 50 minutes to complete this exam.

This is a closed-book, 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 8 pages to the exam, including one blank page for workspace. Please verify that you have all 8 pages.

The page at the end of the exam contains inference rules for various systems. You may detach this page. If you do, please turn it in with the rest of your exam.

Please write your name and NetID in the spaces above.

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

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>15</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td><strong>Total:</strong></td>
<td><strong>100</strong></td>
<td></td>
</tr>
</tbody>
</table>
Problem 1. (15 points)

Suppose we have a type `intlist` defined as:

```ocaml
type intlist = Nil | Cons of int * intlist
```

Write a function `append : intlist -> intlist -> intlist` that takes two intlists `l1` and `l2` and returns a single list that contains all the elements of `l1` followed by all the elements of `l2`. For example, `append (Cons (1, Cons (2, Nil))) (Cons (3, Nil))` should return `Cons (1, Cons (2, Cons (3, Nil)))`.

Solution:

```ocaml
let rec append (l1 : intlist) (l2 : intlist) : intlist =
  match l1 with
  | Nil -> l2
  | Cons (i, rest) -> Cons (i, append rest l2)
```

Problem 2. (20 points)

Consider the following BNF grammar:

```
S ::= ⟨ident⟩$ = ⟨num⟩ | print ⟨ident⟩ | REM ⟨ident⟩
P ::= ⟨num⟩: S | P P
```

(a) (10 points) Write OCaml datatypes `ast_S` and `ast_P` that encode the abstract syntax trees of `S` and `P` respectively. You may represent ⟨ident⟩ with the `string` type and ⟨num⟩ with the `int` type.

Solution:

```ocaml
type ast_S = Assign of string * int | Print of string | Rem of string

type ast_P = Line of int * ast_S | Seq of ast_P * ast_P
```
(b) (10 points) Write the instance of the `ast_P` type corresponding to the AST for the term

```
100: a$ = 5  200: print a
```

If you prefer, you can instead draw the AST for the term.

**Solution:**
```
Seq (Line (100, Assign ("a", 5)), Line (200, Print "a"))
```

**Problem 3.** (25 points)

Suppose you were writing a type-checking function `well_typed : context -> cmd -> bool` that takes a type context `gamma` and an AST for a command `c` and returns `true` if `c` is type-correct given `gamma`. Fill in the skeleton below by translating the typing rule for the if-then-else command from Appendix A into OCaml code, where `IfC (e, c1, c2)` represents the command `if e then c1 else c2`. You may assume the existence of a function `well_typed_exp : context -> exp -> typ -> bool` that takes a type context `gamma`, an expression `e`, and a type `t` and returns `true` if `e` has type `t`, where `t` is either `Tint` or `Tbool`.

```ocaml
let rec well_typed (gamma : context) (c : cmd) : bool =
  match c with
  | IfC (e, c1, c2) ->
```

**Solution:**
```
let rec well_typed (gamma : context) (c : cmd) : bool =
  match c with
  | IfC (e, c1, c2) -> well_typed_exp gamma e Tbool &&
    well_typed gamma c1 && well_typed gamma c2
```
Problem 4. (20 points)
The operational semantics rules for a simple imperative language are shown in Appendix B. Write a proof tree for the next step that

\[(i := \text{if } d \text{ then } 5 \text{ else } i + 1, \{d = \text{false}, i = 0\})\]

takes.

Solution:
Let \(\sigma\) be \(\{d = \text{false}, i = 0\}\).

\[
\begin{array}{c}
(d, \sigma) \Downarrow \text{false} \quad (i, \sigma) \Downarrow 0 \quad (1, \sigma) \Downarrow 1 \\
\hline
\hline
\text{if } d \text{ then } 5 \text{ else } i + 1, \sigma \Downarrow 1 \\
\hline
(i := \text{if } d \text{ then } 5 \text{ else } i + 1, \sigma) \rightarrow (\text{skip}, \{d = \text{false}, i = 1\})
\end{array}
\]
Problem 5. (20 points)
Suppose you wanted to extend the simple imperative language with a command \( x :=? \ e \) that sets \( x \) to the value of \( e \) if the value of \( e \) is non-zero, and does nothing otherwise. For instance, 
\[
(a :=? b - 1; \ c :=? a + 1, \{a = 13, b = 1, c = 14\})
\]
should eventually step to \((\text{skip}, \{a = 13, b = 1, c = 14\})\).
Write hybrid-style semantic rules (big-step for expressions, small-step for commands) describing the behavior of \( x :=? \ e \).

Solution:

\[
\begin{array}{c}
(e, \sigma) \Downarrow 0 \\
\hline
(x :=? e, \sigma) \rightarrow (\text{skip}, \sigma)
\end{array}
\quad
\begin{array}{c}
(e, \sigma) \Downarrow v \quad (v \neq 0) \\
\hline
(x :=? e, \sigma) \rightarrow (\text{skip}, \sigma[x \mapsto v])
\end{array}
\]

or

\[
(x :=? e, \sigma) \rightarrow (\text{if } e = 0 \text{ then skip else } x := e, \sigma)
\]
A Typing Rules for a Simple Imperative Language

\[ \Gamma \vdash n : \text{int} \]
\[ \Gamma \vdash b : \text{bool} \]
\[ (\Gamma(x) = \tau) \]

\[ \Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \]
\[ \Gamma \vdash e_1 \oplus e_2 : \text{int} \]
where \( \oplus \) is an arithmetic operator

\[ \Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{bool} \]
\[ \Gamma \vdash e_1 \otimes e_2 : \text{bool} \]
where \( \otimes \) is a boolean operator

\[ \Gamma \vdash e_1 = e_2 : \text{bool} \]
\[ \Gamma \vdash e_1 \oplus e_2 = v \]
where \( \oplus \) is an arithmetic or boolean operator

\[ \Gamma \vdash e : \text{bool} \]
\[ \Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \]
\[ \Gamma \vdash e_1 = e_2 : \text{int} \]
\[ \Gamma \vdash e_1 \oplus e_2 : \text{int} \]
\[ \Gamma \vdash x := e : \text{ok} \]
\[ \Gamma \vdash e_1 = e_2 : \text{int} \]
\[ \Gamma \vdash e_1 \oplus e_2 = v \]

B Operational Semantics for a Simple Imperative Language

\[ (n, \sigma) \Downarrow n \]
\[ (b, \sigma) \Downarrow b \]
\[ (\sigma(x) = v) \]
\[ (x, \sigma) \Downarrow v \]

\[ (e_1, \sigma) \Downarrow v_1 \]
\[ (e_2, \sigma) \Downarrow v_2 \]
\[ (v_1 \oplus v_2 = v) \]
\[ (e_1 \oplus e_2, \sigma) \Downarrow v \]
where \( \oplus \) is an arithmetic or boolean operator

\[ (e, \sigma) \Downarrow \text{false} \]
\[ (e_1, \sigma) \Downarrow v \]
\[ (\text{if } e \text{ then } e_1 \text{ else } e_2, \sigma) \Downarrow v \]
\[ (e, \sigma) \Downarrow \text{false} \]
\[ (e_1, \sigma) \Downarrow v \]
\[ (\text{if } e \text{ then } e_1 \text{ else } e_2, \sigma) \Downarrow v \]
\[ (e, \sigma) \Downarrow \text{true} \]
\[ (e_1, \sigma) \Downarrow v \]
\[ (\text{if } e \text{ then } e_1 \text{ else } e_2, \sigma) \Downarrow v \]
\[ (e, \sigma) \Downarrow \text{true} \]
\[ (e_1, \sigma) \Downarrow v \]
\[ (\text{if } e \text{ then } e_1 \text{ else } e_2, \sigma) \Downarrow v \]

\[ (c_1; c_2, \sigma) \rightarrow (c_1; c_2, \sigma) \]
\[ (c_1; c_2, \sigma) \rightarrow (c_1; c_2, \sigma) \]
\[ (c_1; c_2, \sigma) \rightarrow (c_1; c_2, \sigma) \]
\[ (c_1; c_2, \sigma) \rightarrow (c_1; c_2, \sigma) \]
\[ (c_1; c_2, \sigma) \rightarrow (c_1; c_2, \sigma) \]
\[ (c_1; c_2, \sigma) \rightarrow (c_1; c_2, \sigma) \]
\[ (c_1; c_2, \sigma) \rightarrow (c_1; c_2, \sigma) \]
\[ (c_1; c_2, \sigma) \rightarrow (c_1; c_2, \sigma) \]
\[ (c_1; c_2, \sigma) \rightarrow (c_1; c_2, \sigma) \]
\[ (c_1; c_2, \sigma) \rightarrow (c_1; c_2, \sigma) \]
C  Scratch Space