There are 5 problems on this exam. It is 9 pages long, so make sure you have the whole exam. You will have $2\frac{1}{2}$ hours in which to work on the problems. You will likely find some problems easier than others; read all problems before beginning to work, and use your time wisely. The exam is worth 100 points total, and each problem is worth 25 points. The point value of each problem is given with the problem, including the point breakdown for the subparts of each problem.

Do all written work in the blue book given to you. Use the blue book for both scratch paper and exam answers, and clearly label the answers to the questions. Ask the proctor for a new blue book if you need one. Make sure you write your identifying number on each blue book and number the blue books in sequence.

It is to your advantage to show your work—we will award partial credit for incorrect solutions that are headed in the right direction. If you feel rushed, try to write a brief statement that captures key ideas relevant to the solution of the problem.

This is a closed-book examination; you may *not* use outside materials.

Notify the proctor if you need to leave the room during the examination. If you finish up in the last ten minutes of the exam, please remain in your seat until the end of the exam as a courtesy to your fellow students.

1. Axiomatic semantics [25 pts]
(parts a–f)

(a) [2 pts] Briefly explain (1–2 sentences) the meaning of the assertion $\{P\}s\{Q\}$.

The following (C or Java) code implements binary search. On entry, the array `a` is assumed to be in ascending sorted order and the element `x` is assumed to be an element of `a`. The return value is the index of the array element containing `x`.

```
int binary_search(int a[], int x) {
    int l = 0;
    int r = a.length - 1;
    int m;

    while (l < r) {
        m = (l+r)/2;
        if (a[m] < x) {
            l = m+1;
        } else {
            r = m;
        }
    }
    return l;
}
```

(b) [3 pts] If the assignment `l = m+1` were changed to `l = m`, how would the correctness of the code be affected? (*Hint*: consider the array `a` = [0,1], `x` = 1.)

(c) [4 pts] Give the precondition and postcondition for this function as formal statements.

(d) [8 pts] Recalling that the Hoare rule for the correctness of `while` has the form

$$\frac{\{P \ \wedge \ b\} \ s \ \{P\}}{\{P\} \ \texttt{while} \ (b) \ s \ \{P \ \wedge \ \neg b\}}$$

give a formal statement of a loop invariant that would enable you to prove the partial correctness of this code.

(e) [4 pts] Argue briefly and informally that this loop invariant holds at the start of the loop.

(f) [4 pts] Argue briefly that at loop termination this loop invariant implies the postcondition.

2. Dataflow analysis [25 pts]
(parts a–g)

One of the standard dataflow analyses is *live variable analysis*, which identifies the set of live variables for each node of a control-flow graph. A variable is live if its value is needed for a later computation. Live variable analysis can be used, among other things, to perform *dead code elimination*: removing code whose only effect is to update a variable that is not live. However, removing code can affect the results of live variable analysis, revealing even more code as dead. In this problem you will develop a dataflow analysis that identifies all the dead code in one analysis phase, replacing repeated iterations of live variable analysis.

(a) [3 pts] What is another common use for live variable analysis? (one-sentence answer)

Suppose that we have a control flow graph composed of the following kinds of nodes, where a represents an integer constant; x, y, z represent variables, $\oplus$ represents some ALU operation, and f is the address of some function.

$x \leftarrow a$	$x \leftarrow y \oplus z$
$x \leftarrow \texttt{MEM}[y]$	$\texttt{MEM}[x] \leftarrow y$
$\texttt{if}\ x\ \texttt{goto}$	$x \leftarrow f(y, z, \ldots)$
`start`	$\texttt{return}\ x$

Each node n has a set of predecessor nodes $pred[n]$ and a set of successor nodes $succ[n]$; the set of successors has exactly one element except in the case of the `if` and `return` nodes, which have two and zero successors respectively. Only the unique `start` node has no predecessors.

Live variable analysis seeks to determine, for each node n, the set of variables $live[n]$ that are live on entry to the node. This can be found using the following dataflow equation:

$$live[n] = gen[n] \cup \left(\bigcup_{n' \in succ[n]} live[n'] - kill[n] \right)$$

(b) [2 pts] Is this a backward or a forward analysis?

(c) [2 pts] How should $live[n]$ be initialized for each node type n in order to solve the equations iteratively?

(d) [4 pts] Define the functions gen and $kill$ for each of the node types above so that the live variable analysis is correct.

3. [4 pts] Does the iterative algorithm always terminate for this analysis? Explain why or why not in 2–3 sentences.

Now, suppose we want the compiler to perform dead code elimination at the same time that it does live variable analysis. Dead code should not cause the variables it uses to remain live. For example, if a CFG contains the node $x \leftarrow x + y$ and the variable x is used in no other node, the node should always be marked dead. Statements of the form $x \leftarrow f(y, z)$, $\texttt{MEM}[x] \leftarrow y$, and $\texttt{return}\ x$ are considered conservatively never to be dead. For simplicity, we will never consider `if` statements dead either. Extend the analysis above to do dead code elimination. Your analysis should compute a boolean value $isdead[n]$ in addition to $live[n]$ for each node.

(f) [8 pts] Give the new dataflow equations.

(g) [2 pts] How should the dataflow values be initialized for iterative solving?

4. Type systems [25 pts]
(parts a–c)

Consider a simple imperative programming language, "core-Java":

$$
\begin{aligned}
x &\in \textbf{Var} \quad \text{(a countable set of legal variable names)}\\
n &\in \mathbb{Z}\\
e \in \textbf{Expr} &::= x \mid n \mid \texttt{true} \mid \texttt{false} \mid x := e \mid e_1 \texttt{==} e_2 \mid e(e_1, \ldots, e_n)\\
s \in \textbf{Stmt} &::= e \mid s_1;\ s_2 \mid \texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2 \mid \texttt{while}\ (e)\ s \mid \texttt{return}\ e; \mid \{d_1 \ldots d_n\ s\}\\
d \in \textbf{Decl} &::= \tau\ x;\\
\tau \in \textbf{Type} &::= \texttt{int} \mid \texttt{bool} \mid (\tau_1, ..., \tau_n) \to \tau\\
m \in \textbf{Method} &::= \tau\ x_M(\tau_1\ x_1, ..., \tau_n\ x_n)\{s\}
\end{aligned}
$$

The type system for the expression fragment of the language is specified with the judgment $\Gamma \vdash e : \tau$, which means that under the variable typing assumption Γ, the expression e has type τ. Here, Γ is a finite map from variables (x) to types (τ). The bodies of all methods are checked with respect to a Γ in which every method name x_M is bound to the appropriate function type. The judgment $\Gamma \vdash e : \tau$ is defined as follows:

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{}{\Gamma \vdash n : \texttt{int}}$$

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}} \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}}$$

$$\frac{\Gamma \vdash x : \tau \quad \Gamma \vdash e : \tau}{\Gamma \vdash x := e : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 \texttt{==} e_2 : \texttt{bool}}$$

$$\frac{\Gamma \vdash e : (\tau_1, ..., \tau_n) \to \tau \quad \Gamma \vdash e_i : \tau_i}{\Gamma \vdash e(e_1, ..., e_n) : \tau}$$

We wish to extend the type system so that we can check whether statements are well-formed.

One thing a type-checker for such a language must do is ensure that every method returns a value of the proper type. For instance, the following method should be rejected because it returns a boolean instead of an integer:

```
int foo(bool b) {
  return b;
}
```

In addition, this method should be rejected because it fails to return a value (falls off) when we pass a value other than 0:

```
int bar(int x) {
  if (x == 0)
    return 3;
```

```
  else
    4;
}
```

(a) [10 pts] Design a set of inference rules that allow us to conclude a judgment of the form:

$$\Gamma \vdash s : \tau \ \textbf{stmt}$$

with the intended meaning that if s returns by executing a `return` statement, then it returns a value of type τ.

(b) [10 pts] Design a set of inference rules that allow us to conclude a judgment of the form:

$$\Gamma \vdash s \ \textbf{no-fall-off}$$

with the intended meaning that s will not fall off. That is, s will either loop forever or else return.

(c) [5 pts] The type system as defined is still unsound. Explain what needs to be fixed in 1–2 sentences.

5. Semantics [25 pts]
(parts a–d)

The following is the syntax for a simple untyped stack-based language named FootNote[1]:

$$\begin{array}{rcl} c \in \textbf{Cmd} & ::= & \texttt{skip} \mid n \mid x \mid c_0\ c_1 \mid \oplus \mid \texttt{apply} \mid \texttt{def}\ x\ \texttt{in}\ c \mid \texttt{ifz} \\ v \in \textbf{Value} & ::= & n \mid \{c\} \\ S \in \textbf{Stack} & ::= & \bullet \mid v :: S \end{array}$$

Here, x denotes a legal variable, n denotes an integer, and $\oplus$ denotes an arithmetic operation. Any command c is a well-formed program. A sequence of two juxtaposed commands represents sequential composition (similar to `;` in many languages). An expression of the form $\{c\}$ is a *block*; it is a command packaged in a value that can be executed later using the `apply` command. The `def` command introduces a new variable x that is in scope in the command c. The `ifz` command checks whether top of the stack is the integer 0 and conditionally applies one of two blocks found below it on the stack.

In the small-step operational semantics for FootNote, a configuration is a pair c, S where S captures the current state of the stack. A final configuration has the form $\texttt{skip}, S$. The small-step SOS rules are the following, where $c\{v/x\}$ represents capture-avoiding substitution of v for x.

$$\frac{c_0, S \longmapsto c_0', S'}{c_0\ c_1, S \longmapsto c_0'\ c_1, S'} \qquad \frac{}{\texttt{skip}\ c, S \longmapsto c, S}$$

$$\frac{}{n, S \longmapsto \texttt{skip}, n :: S} \qquad \frac{n = n_2 \oplus n_1}{\oplus, n_1 :: n_2 :: S \longmapsto \texttt{skip}, n :: S}$$

$$\frac{}{\{c\}, S \longmapsto \texttt{skip}, \{c\} :: S} \qquad \frac{}{\texttt{apply}, \{c\} :: S \longmapsto c, S}$$

$$\frac{}{\texttt{def}\ x\ \texttt{in}\ c, v :: S \longmapsto c\{v/x\}, S}$$

$$\frac{}{\texttt{ifz}, 0 :: \{c_1\} :: \{c_2\} :: S \longmapsto c_2, S} \qquad \frac{n \neq 0}{\texttt{ifz}, n :: \{c_1\} :: \{c_2\} :: S \longmapsto c_1, S}$$

(a) [3 pts] What is the final state of the following program, executed with an initially empty stack?

```
{def f in def n in {0} {n 2 - f f apply n +} n ifz}
    def g in 4 g g apply
```

(b) [8 pts] Identify all the stuck configurations in the language.

Now, suppose we want to write a continuation-passing style semantics for FootNote. We start by defining some domain equations:

[1] It is based on PostScript

$$
\begin{array}{rcl}
\rho \in \mathit{Environment} & = & \mathbf{Var} \to (\mathit{Value} + \mathit{Error}) \\
\mathit{Error} & = & \mathit{Unit} \\
S \in \mathit{Stack} & = & \mathit{Unit} + \mathit{Value} \times \mathit{Stack} \\
\mathit{Value} & = & \mathbb{Z} + \mathit{Block} \\
k \in \mathit{Continuation} & = & \mathit{Stack} \to \mathit{Answer}
\end{array}
$$

(c) [5 pts] Give a suitable domain equation for the domain *Block*.

We can now give the CPS semantics with a well-founded definition of a semantic function $\mathcal{C}$ with the signature

$$\mathcal{C} \in \mathbf{Cmd} \to \mathit{Environment} \to \mathit{Stack} \to \mathit{Continuation} \to \mathit{Answer}$$

Assume we have a continuation k_{halt} whose invocation terminates the program. For convenience in writing the semantics you may use an *case* expression with deep pattern matching to deconstruct sums and products.

(d) [9 pts] Complete the CPS semantics for the remaining eight command forms. For brevity the semantic function has been applied on the left-hand side to arbitrary ρ, S, and k.

$\mathcal{C}[\![\texttt{skip}]\!]\rho S k = kS$

$\mathcal{C}[\![n]\!]\rho S k =$

$\mathcal{C}[\![x]\!]\rho S k =$

$\mathcal{C}[\![\oplus]\!]\rho S k =$

$\mathcal{C}[\![c_0\ c_1]\!]\rho S k =$

$\mathcal{C}[\![\texttt{apply}]\!]\rho S k =$

$\mathcal{C}[\![\{c\}]\!]\rho S k =$

$\mathcal{C}[\![\texttt{ifz}]\!]\rho S k =$

$\mathcal{C}[\![\texttt{def}\ x\ \texttt{in}\ c]\!]\rho S k =$