1. Short Answers [25 pts]

(a) [2 pts] A bottom-up parser might yield a reduce-reduce conflict while processing an input program with incorrect syntax. True or false?

(b) [2 pts] In a language with a sound type system, evaluation of expressions never gets stuck. True or false?

(c) [2 pts] Structural operational semantics defines the meaning of a program by induction on the structure of the program’s type. True or false?

(d) [3 pts] Which of the following analyses have distributive transfer functions?
   - Variable liveness
   - Available expressions
   - Constant folding
   - Reaching definitions

(e) [3 pts] If types \( A, B, \) and \( C \) are three types such that \( A \) is a subtype of \( B \), and \( B \) is a subtype of \( C \), then which of the following function types are subtypes of \( B \rightarrow B \)?
   1) \( A \rightarrow B \)
   2) \( B \rightarrow C \)
   3) \( C \rightarrow C \)
   4) \( C \rightarrow A \)
   5) \( A \rightarrow A \)

(f) [2 pts] Consider the following SML program which declares a function \( f \) with a nested function \( g \). Is the function \( f \) tail-recursive?

```sml
fun f(n:int, g2: int->int) : int = 
  let fun g(y: int) = y + n + 10 
      in 
      if n = 1 then n + g(n) + g2(n) 
      else f(n-1, g) 
    end 
```

(g) [2 pts] Now suppose we also define a function \( g0 \):

```sml
fun g0(y: int) : int = 100 
```

What is the value of the expression \( f(1, g0) \), using the definition of \( f \) from part 1(f)? Recall that SML has lexical variable scoping.

(h) [2 pts] What is the value of the expression \( f(2, g0) \)?

(i) [3 pts] What does the following Prolog predicate \( f \) do? (Recall that \( [X|Y] \) is the Prolog notation for list \( \text{cons} \), similar to ML’s \( X :: Y \).)

```prolog
f([], X). 
\( f([X|Y], [X|Z]) :- f(Y,Z). \)
```

(j) [2 pts] Reference counting can fail to identify some garbage. True or false?

(k) [2 pts] If the Hoare triple \( \{P\} c\{Q\} \) holds, then so does \( \{P \lor P'\} c\{Q\} \) for any formula \( P' \). True or false?
2. **Object-Oriented Features** [25 pts]

Consider a object-oriented language similar to Java with the following syntax:

- **Classes** $C ::= \text{class } c \text{ extends } c' \{ \overline{f}; \overline{d} \}$
- **Methods** $d ::= c \text{ m}(\overline{x}) \{ \text{return } e; \}$
- **Expressions** $e ::= x | \text{new } c(\overline{x}) | \text{null} | \text{this} | e.f | e.m(\overline{x})$

where $c$ ranges over class names, $f$ over field names, $m$ over method names, and $x$ over variables. We use overlines as a shorthand for sequences. For instance, $\overline{f}$ represents a sequence of field declarations $c_1 f_1; \ldots; c_n f_n$. All of the methods in the program are virtual. Program variables $x$ include the special variable `this` that represents the receiver object. Class names include a special class `Object` with no fields or methods.

The expression `new c(\overline{x})` creates a new object of class $c$ and initializes its fields $\overline{f}$ with the values of $\overline{x}$. Each class name can be used throughout the program, even before the class is defined. A program consists of a set of class declarations and an expression $e$ to evaluate.

The above language supports method overloading and overriding. This makes it possible to declare multiple methods with the same name in the same class hierarchy, as illustrated in the example below:

```java
class A extends Object {
    B f;
    A m(A a) { return a; } /* method 1 */
    A m(B b) { return b; } /* method 2 */
}
class B extends A {
    A m(A a) { return a.f; } /* method 3 */
    A m(B b) { return b.f; } /* method 4 */
}
```

(a) [5 pts] Define the concepts of method overriding and method overloading, and indicate where they occur in the example above.

(b) [3 pts] Briefly explain (in no more than 2 sentences) when and how are method overloading and overriding being resolved?

(c) [7 pts] Now we want to define a model that precisely describes the semantics of our language. We describe the evaluation of expressions using a small-step evaluation relation $e \rightarrow e'$. The evaluation eventually yields a value $v$, which is either an object or null:

$$Values \ v ::= \text{null} | \text{new } c(\overline{x})$$
The following rules describe the execution of all expressions except method calls:

\[
\begin{align*}
\text{new } c(\overline{e}) & \longrightarrow \text{new } c(\overline{e}') \\
\overline{e} & \longrightarrow \overline{e}' \\
f_i \in \text{fields}(c) \quad v_i & \in \overline{v} \\
\text{new } c(\overline{v}).f_i & \longrightarrow v_i
\end{align*}
\]

Here, \(\text{fields}(c)\) is the sequence of fields of class \(c\), including those defined in the superclasses. The notation \(v_i \in \overline{v}\) indicates that \(v_i\) is the \(i\)-th element in the sequence \(\overline{v}\) (and similarly for \(f_i\)). The evaluation of sequences \(\overline{e} \longrightarrow \overline{e}'\) is done by evaluating each expression from left to right:

\[
\begin{align*}
\overline{e} & \longrightarrow \overline{e}' \\
(v_1, \ldots, v_{i-1}, e_i, v_{i+1}, \ldots, v_n) & \longrightarrow (v_1, \ldots, v_{i-1}, e'_i, v_{i+1}, \ldots, v_n)
\end{align*}
\]

For method calls, our semantic model first records the static types of its arguments, before evaluating the call:

\[
\forall e_i \in \overline{e}, c_i \in \overline{c} : \quad c_i = \text{type}(e_i) \\
\overline{e}.m(\overline{c}) & \longrightarrow \overline{e}.[\overline{c}]\overline{e}
\]

Write the remaining rules to complete the evaluation of method calls in the presence of overloading and overriding. You can assume that \(\text{decl}(c, m, \overline{c})\) returns the declaration of the method with name \(m\) and signature \(\overline{c}\) in class \(c\).

(d) [6 pts] Next, we are concerned with the implementation of our language in a standard virtual machine. Consider a program that evaluates the expression:

\[
(\text{new } A(\text{null})).m(\text{new } A(\text{new } B(\text{null})))
\]

where classes \(A\) and \(B\) are the ones defined at the beginning of the problem. Draw a reasonable memory layout of the virtual machine when the program execution is about to evaluate the body of method \(m\). Clearly indicate the stack, the heap, and the static area in your figure.

(e) [4 pts] In C++, the usual object-oriented method behavior is obtained by defining methods with the keyword \texttt{virtual}. By default, C++ methods (i.e., without the \texttt{virtual} or \texttt{static} qualifiers) are not virtual, but are still able to access the fields of the enclosing object. Briefly compare the way virtual methods and C++ default methods are implemented, indicating how they differ and in what respects they are similar. (2–3 sentences)
3. Dataflow analysis [25 pts]

Consider the control-flow graph $G = (V, E)$ for a procedure, which has a distinguished node $START$ that has the property that there is a path from $START$ to every node in $V$. A node $d$ is said to dominate a node $v$ if every path from $START$ to $v$ contains $d$.

(a) [5 pts] Draw a CFG for the following IMP code, with each node annotated by the corresponding line number:

1: if x > 0 then
2:   while n < 10 do
3:     n := n + x
4: else
5:   n := 0

(b) [5 pts] The dominance relation is the set of pairs $(d, v)$ such that $d$ dominates $v$. Give the dominance relation for the previous example.

(c) [5 pts] Write down a set of dataflow equations for computing the dominance relation. Your answer must state clearly the domain for equations and whether your dataflow scheme is forward-flow or backward-flow.

(d) [5 pts] A program is said to be in static single assignment (SSA) form if every variable is defined exactly once in the program and every use of a variable is reached by exactly one definition of that variable in the program. By convention, $START$ is considered to be a (second) definition for all variables in the program. Assuming every variable is known to be defined once, briefly explain how you would use a standard dataflow analysis to determine whether or not a program is in SSA form.

(e) [5 pts] Suppose a program is in SSA form. If a definition $d$ of a variable $x$ is the only definition that reaches a use of $x$, is it necessarily true that $d$ dominates that use of $x$?
4. Types and Exceptions [25 pts]

Exceptions are a commonly supported language feature, but they are supported differently in different languages. In SML, exceptions are not statically checked, so statically the type checker does not know what exceptions a given function might throw. In Java, (most) exceptions must be statically declared and the type checker ensures that all exceptions are caught unless they are allowed to escape.

(a) [5 pts] Why are exceptions a useful language feature? Discuss briefly in two or three sentences.

Let us explore static typing of exceptions in the context of the simply typed lambda calculus. The typed lambda calculus with integers has the following syntax.

\[
x \in \text{Var} \\
n \in \mathbb{Z} \\
e ::= n \mid x \mid \lambda x:\tau.e \mid e_0 e_1
\]

A program is a well-typed term \( e \), which evaluates according to a standard small-step operational semantics \( e \rightarrow e' \) to arrive at a value \( v \), which is either an integer \( n \) or a lambda term \( \lambda x:\tau.e \).

The following new terms generate and handle exceptions respectively:

\[
e ::= \ldots \mid \text{throw } X \mid \text{try } e_1 \text{ catch } X \Rightarrow e_2
\]

In the \texttt{throw} expression, \( X \) is the name of the exception. For simplicity, there is no associated value as there would be in SML or Java. Informally, the \texttt{try} expression evaluates \( e_1 \) to obtain the value of the whole expression, but if \( e_1 \) throws the exception \( x \), the expression \( e_2 \) is evaluated instead. Throwing an exception \( X \) immediately transfers control to the closest dynamically enclosing clause \texttt{catch}(X) with the same name \( X \).

To statically keep track of possible exceptions, the typing judgement will take the form

\[
\Gamma \vdash e : \tau, S
\]

where \( S \) is a set \( \{X_1, \ldots, X_n\} \) including all exceptions that might be thrown by \( e \). Further, the type of a function must, as in Java, include its possible exceptions, so it will have the form \( \tau \rightarrow \tau'/S \) meaning that it either has a result of type \( \tau' \) or throws one of the exceptions in the set \( S \).
(b) [12 pts] Complete the typing rules below for this language.

\[
\begin{align*}
\tau & ::= \text{int} \mid \tau \to \tau'/S \\
\Gamma & ::= \Gamma, x: \tau \mid \emptyset
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash n : \text{int}, \emptyset \\
x \in \text{dom}(\Gamma) & \quad \Gamma \vdash x : \Gamma(x), \emptyset
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash e_0 : \tau \to \tau'/S_2, S_0 & \quad \Gamma \vdash e_1 : \tau, S_1 \\
\Gamma \vdash e_0 \ e_1 : [ ] & \quad \Gamma, x: \tau \vdash e : [ ] & \quad \Gamma \vdash (\lambda x: \tau. e) : [ ]
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash \text{throw} \ X \ : \ \tau, [ ]
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash e_1 : \tau, S_1 & \quad \Gamma \vdash e_2 : [ ], S_2 \\
\Gamma \vdash \text{try} \ e_1 \ \text{catch} \ X \Rightarrow e_2 : [ ]
\end{align*}
\]

(c) [3 pts] What guarantees that a type checker for this language, implemented according to these rules, will terminate? Explain in one or two sentences.

(d) [5 pts] Suppose we wanted to translate this language into a target language that is the simply typed lambda calculus with sum types \( \tau_1 + \tau_2 \). If a term has type \( \tau, \{X_1, \ldots, X_n\} \), what target language type might be used as a suitable representation? Sketch how this representation would work in 2–3 sentences.