1. Implementation: Dispatching [20 pts]

Many programming languages have a case or switch statement that allows a multi-way branch on the value of some expression. For example, the Modula-3 CASE statement (slightly simplified) has the form

CASE e OF
  $L_1 => S_1$
| ...
| $L_n => S_n$
END

Here, each label L_i is either a constant expression of the same type as e, or else a range $e_1..e_2$.
One question is how this statement can be implemented to quickly dispatch to the correct statement S_i. Obviously, a CASE statement can be translated to a linear sequence of comparisons and branches, but this implementation can result in slow dispatch to statements late in the ordering. Another possible implementation uses a jump table with one entry for each index i.

(a) [7 pts] Describe a third implementation that, like the first, uses only direct branches, but dispatches to all statements S_i in comparable time.

(b) [8 pts] Suppose a compiler chooses the implementation of each CASE statement depending on the actual values L_i. Considering modern architectures, in what situations would each of these three implementations give the best performance? Explain.

(c) [5 pts] In object-oriented programming, statements like the CASE statement and relatives are typically used less frequently than in non-object-oriented languages. Explain why.

2. Static Analysis: Definite Assignment [25 pts]

Consider a simple, imperative core language with the following expression and statement forms:

$$\begin{array}{llll} \text{(expressions)} & e & ::= & x \mid i \mid e_1 + e_2 \\ \text{(statements)} & s & ::= & \textbf{skip} \mid x := e \mid s_1; s_2 \mid \textbf{if}0\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \mid \textbf{while}0\ e\ \textbf{do}\ s \end{array}$$

The operational semantics for the language can be described using two sets of inference rules: one for expressions and one for statements.

The inference rules for expressions allow us to deduce judgments of the form $\langle e, \sigma \rangle \Downarrow i$ where σ is a *finite* partial function mapping variables to integers and i is an integer. Note that when a variable x does not occur in the domain of σ, we cannot derive $\langle x, \sigma \rangle \Downarrow i$.

$$\frac{}{\langle i, \sigma \rangle \Downarrow i} \qquad \frac{}{\langle x, \sigma \rangle \Downarrow \sigma(x)} \ (\text{when } x \text{ in } \textit{Dom}(\sigma)) \qquad \frac{\langle e_1, \sigma \rangle \Downarrow i_1 \quad \langle e_2, \sigma \rangle \Downarrow i_2}{\langle e_1 + e_2, \sigma \rangle \Downarrow i_1 + i_2}$$

The inference rules for statements allow us to deduce judgments of the form $\langle s, \sigma_1 \rangle \Downarrow \sigma_2$, reflecting state transitions.

$$\frac{}{\langle \textbf{skip}, \sigma \rangle \Downarrow \sigma} \qquad \frac{\langle e, \sigma \rangle \Downarrow i}{\langle x := e, \sigma \rangle \Downarrow \sigma[x \mapsto i]}$$

$$\frac{\langle s_1, \sigma_1 \rangle \Downarrow \sigma_2 \quad \langle s_2, \sigma_2 \rangle \Downarrow \sigma_3}{\langle s_1; s_2, \sigma_1 \rangle \Downarrow \sigma_3} \qquad \frac{\langle e, \sigma_1 \rangle \Downarrow 0 \quad \langle s_1, \sigma_1 \rangle \Downarrow \sigma_2}{\langle \textbf{if}0\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2, \sigma_1 \rangle \Downarrow \sigma_2}$$

$$\frac{\langle e, \sigma_1 \rangle \Downarrow i \neq 0 \quad \langle s_2, \sigma_1 \rangle \Downarrow \sigma_2}{\langle \textbf{if}0\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2, \sigma_1 \rangle \Downarrow \sigma_2} \qquad \frac{\langle \textbf{if}0\ e\ \textbf{then}\ (s;\ \textbf{while}0\ e\ \textbf{do}\ s)\ \textbf{else}\ \textbf{skip}, \sigma \rangle \Downarrow \sigma'}{\langle \textbf{while}0\ e\ \textbf{do}\ s, \sigma \rangle \Downarrow \sigma'}$$

Note that for an assignment $x := e$, we evaluate e to an integer i and then update the state σ so that if we look up the value of x, we will get the value of i (denoted $\sigma[x \mapsto i]$). That is, the resulting state is the partial function given by:

$$\lambda v.\textbf{if}0\ v = x\ \textbf{then}\ i\ \textbf{else}\ \sigma(v)$$

Clearly, it is an error to attempt to use a variable before we assign it a value. Some languages, notably Java, perform a static analysis to ensure that variables are definitely assigned before they are used. In fact, Java performs such an analysis in a single pass over the code.

(a) [15 pts] Describe a one-pass analysis that determines whether or not a variable has been definitely assigned before it is used. To receive full credit, you should carefully describe what information you are computing for each expression or statement form.

(b) [10 pts] The one-pass definite-assignment analysis of Java is simplified by the fact that control flow is (largely) structured. Briefly explain how to perform definite-assignment analysis in the presence of unstructured control-flow operators such as **goto**.

3. Typing and the Lambda Calculus [25 pts]

The types and terms of the simply-typed lambda calculus can be described by the following BNF definitions:

$$\begin{array}{llcl} \text{(types)} & \tau & ::= & B \mid \tau_1 \to \tau_2 \\ \text{(terms)} & e & ::= & x \mid \lambda x.e \mid e_1\, e_2 \\ \text{(values)} & v & ::= & \lambda x.e \end{array}$$

where we use x to range over a denumerable set of variables, and B ranges over a set of base types (*e.g.*, int).

We can describe call-by-name evaluation of lambda calculus terms using the following inference rules:

$$\frac{}{\lambda x.e \Downarrow \lambda x.e}$$

$$\frac{e_1 \Downarrow \lambda x.e \quad e\{e_2/x\} \Downarrow v}{e_1\, e_2 \Downarrow v}$$

where the notation $e\{e_2/x\}$ denotes capture-avoiding substitution of the expression e_2 for the free occurrences of the variable x within e.

The well-typed terms of the simply-typed lambda calculus can be described by the following typing rules, where Γ is a type assignment mapping variables to types:

$$\frac{}{\Gamma \vdash x : \Gamma(x)}$$

$$\frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\, e_2 : \tau_2}$$

It is a fact that evaluation of any well-typed term terminates under call-by-name. That is, if $\vdash e : \tau$, then there exists a v such that $e \Downarrow v$, and furthermore, $\vdash v : \tau$.

Yet, there are terms that do not terminate when evaluated. For instance, the term Ω defined as:

$$(\lambda y.(y\, y))(\lambda x.(x\, x))$$

diverges when evaluated.

(a) [10 pts] Using the evaluation rules, show that there is no v such that $\Omega \Downarrow v$.

(b) [10 pts] Prove that there is no τ such that $\vdash \Omega : \tau$.

(c) [5 pts] Suggest an extension to the type system that would allow terms such as Ω. (You need not describe this formally—an informal explanation will suffice.)

4. Modules and Separate Compilation [30 pts]

Consider the following module extension to the typed lambda calculus of Problem 3. In this extension, a program p is a collection of modules and and a top-level expression e that is evaluated with the modules in scope. Each module has a name y and contains some number of type definitions t and value definitions (fields) f. A type definition introduces a type identifier X; a value definition introduces a variable x. In BNF, the added syntax is the following:

$$\begin{array}{rcl} p & ::= & m_1 \ldots m_k\ e \\ m & ::= & \textsf{module}\ y\ \{\textsf{types}\ t_1 \ldots t_m\ \textsf{values}\ f_1 \ldots f_n\} \\ t & ::= & X = \tau \\ f & ::= & x : \tau = v \end{array}$$

Legal types τ and terms e are extended to allow the selection of an identifier from a module named y.

$$\begin{array}{rcl} \tau & ::= & \ldots \mid y.X \\ e & ::= & \ldots \mid y.x \end{array}$$

A type of the form $y.X$ is abstract: it is not known to be equal to any type other than itself, except in the type-checking of the module y itself, where $y.X$ is equivalent to the type to which the identifier X is bound in the types section of module y. Each module name is in scope in every module in the same program and in the top-level expression of that program. (For simplicity, the name of a module is not in scope in the types section of the module itself; however, this is not essential to the problem.)

(a) [5 pts] Structuring a program into modules facilitates separate compilation because the modules can be compiled separately. How does it also help programmers write programs correctly? Discuss briefly (2-3 sentences).

(b) [5 pts] Assuming modules in this language are compiled separately, what is the least amount of information about other modules in the program that is needed in order to type-check a given module in isolation?

(c) [10 pts] Suggest an extension to the type-checking context Γ of Problem 3 that allows the typing rules for the typed lambda calculus to be extended to this new language. You do not need to work out the typing rules.

(d) [5 pts] Suppose the compiler for this language generates code that uses a standard stack discipline, and also stack-allocates local variables. If compiling a module separately in this style, what is the least information about other modules that the compiler needs?

(e) [5 pts] Modules are more reusable if they can be used polymorphically. One possible kind of parametric polymorphism is what we might call *template modules*, with the following syntax:

$$m ::= \ldots \mid \textsf{module}\ y[X'_1, \ldots, X'_l]\ \{\textsf{types}\ t_1 \ldots t_m\ \textsf{values}\ f_1 \ldots f_n\}$$

The formal type parameters X'_i can be used as types within the module definition. An element of a template module is selected by first instantiating the module on a list of l types, binding the identifiers $X'_1 \dots X'_l$. Formal type parameters and instantiation extend the legal type and term expressions:

$$
\begin{array}{rcl}
\tau & ::= & \dots \mid y[\tau'_1, \dots, \tau'_l].X \mid X \\
e & ::= & \dots \mid y[\tau'_1, \dots, \tau'_l].x
\end{array}
$$

Suppose that the expressions involving instantiations of template modules are well-formed exactly when the body of the module is well-formed after substitution of the actual parameters $\tau'_1, \dots, \tau'_l$ for the formal parameters $X'_1, \dots, X'_l$. What are the implications for separate compilation? Explain in 2–3 sentences.