Question 1.

In an environment in which computer programs are freely transmitted across the Internet, porting and security issues are becoming increasingly important.

Part (a)

Define at least three classes of portability and/or security problems that a program (in source or object form) imported from an external site may be subject to.

Part (b)

Assume that we have the complete source for an imported program, including instructions for its configuration (*e.g.*, a Makefile including compiler options). What kinds of compile-time analyses can be used to detect possible occurrences of the problems you defined in part (a)?

Part (c)

Instead of compile-time analyses, what kinds of run-time actions can be used to detect or prevent the classes of problems you defined in part (a)?

Question 2.

Recall that among the possible ways to represent the statements in a basic block are:

- as a sequence of abstract-syntax trees (one for each statement)
- as a DAG.

Part (a)

Give an algorithm for building a DAG representation of a basic block. (Assume that a basic block consists of a sequence of 3-address statements.) Illustrate your algorithm using the following sequence of statements (please give illustrations of some intermediate stages of your algorithm, not just the final DAG).

```
a = 4 * k
b = a
c = b * k
d = 4 * k
b = b + 1
```

Part (b)

What are some advantages of the DAG representation over the sequence of trees representation?

Part (c)

Suppose the sequence of statements in the basic block includes array references (*e.g.*, "A[k] = 0" or "x = A[k]"). How does this complicate the process of building the DAG representation of the basic block (and what must be done to handle such array references)?

Question 3.

In this question, we explore how a program optimizer might take into account information about whether the condition of an if-then-else statement or a while loop is always true or always false. We consider only a limited version of the problem, which has the following features:

- Programs are assumed to consist of a single procedure.
- The program is assumed to have no aliasing.
- The analysis only needs to track state changes for certain kinds of assignment statements:
  - assignments of constant values, *i.e.*, statements of the form $x = c$, where c is a constant,
  - copy statements, *i.e.*, statements of the form $x = y$, where y is another program variable.

  The analysis should assume that nothing is known about the value of x after other kinds of statements that assign to x (*e.g.*, $x = y + z$).
- All expressions in conditions are of the form $x == 0$, where x is a program variable.

Assume that we have an analysis that determines (safely), for each condition with an expression of the form $x == 0$, whether x is always 0, always non-0, or of unknown value. One approach to optimizing the program would be to iterate between phases of analysis and transformation:

```
[1] Analyze the program (to discover information about the values of variables used in conditions)
[2] while there are conditions with statically determinable values do
[3]     Remove the non-executable branches and their controlling conditions
[4]     Analyze the program (to discover information about the values of variables used in conditions)
[5] od
```

For Parts (a) and (b), assume that the analysis algorithm used in steps [1] and [4] does not interpret conditions. (Step [2] interprets conditions—with respect to the information gathered in steps [1] and [4]—however, this is done as a separate stage in between invocations of the analysis algorithm proper.)

Part (a)

Give an example program for which more than one iteration is necessary to produce the best results. Explain what code is removed on each iteration, and why.

Part (b)

Give an example program in which there is a condition with the following properties: (i) the condition depends only on assignments from constants and copies, (ii) the condition is always true on any actual execution, yet (iii) the iterative algorithm given above would not detect this fact. [Hint: Consider while loops.]

Part (c)

Part (b) suggests that we need an analysis that accounts for the values of conditions *as part of the analysis*. Define a dataflow analysis (for statically determining the values of conditions of the form $x == 0$) that incorporates the notion of not propagating information down a branch until there is "evidence" that the branch will be taken.

Recall that a dataflow analysis can be defined by specifying

- A lattice of dataflow values, together with the lattice's meet operation.
- A dataflow function for conditions and for each kind of statement. (Each function maps the dataflow value that characterizes the state before the statement/predicate executes to the dataflow value that characterizes the state after it executes.)

[Hint: Put the dataflow functions on the edges of the control-flow graph. The function on a condition's outgoing true edge need not be the same as the function on the condition's outgoing false edge.]

Part (d)

Illustrate your answer to Part (c) using your example from Part (b).

Question 4.

Consider a generalized kind of constant propagation that determines, for each program point and for each variable, whether the variable is *N-limited*; that is, whether it contains one of a set of up to N values. (Normal constant propagation determines whether a variable is 1-limited.)

Part (a)

Define a dataflow framework that determines which variables at which points are N-limited, for a fixed N. Assume the usual simple imperative programming language (a program is a single procedure, there is no aliasing, *etc*).

Part (b)

How can the knowledge that a variable is N-limited at a particular point be used by an optimizing compiler?

Part (c)

What are the advantages and disadvantages of this dataflow problem compared with normal constant propagation?

Question 5.

Consider the simple imperative language defined below.

program → *cmd*
cmd → Id := *intexp* | repeat *cmd* until *boolexp* | *cmd* ; *cmd* | switch (*intexp*) *cases*
cases → case *intexp* : *cmd* | case *intexp* : *cmd* ; *cases*
intexp → IntLit | Id | *intexp* + *intexp*
boolexp → BoolLit | *intexp* == *intexp*

That is, a program is a command, and a command is an assignment, a repeat-loop, a command followed by another command, or a switch, and commands contain simple integer and boolean expressions.

A partially defined denotational semantics for this language is given below. A *State* is a mapping from identifiers to values; initial state σ_0 is the state that maps all identifiers to zero. The meaning functions **I** and **B**, used to define the meanings of integer and boolean literals, simply return the values of their arguments. The function "update" used to define the meaning of the assignment command takes three parameters: a state σ, an identifier *x*, and an integer value *v*, and returns a state that is the same as σ except that it maps *x* to *v*.

Meaning Functions

P: Command → State
C: Command → State → State
IE: IntExpression → State → Integer
BE: BoolExpression → State → Bool

P[[*C*]] = **C** [[*C*]] σ_0
C[[*Id* := *intexp*]] = $\lambda\sigma$. update(σ, *Id*, **IE** [[*intexp*]]
C[[C_1 ; C_2]] = $\lambda\sigma$. **C** [[C_2]] (**C** [[C_1]] (σ))
IE[[*IntLit*]] = $\lambda\sigma$. **I** [[*IntLit*]]
IE[[*Id*]] = $\lambda\sigma.\sigma$(*Id*)
IE[[$intexp_1$ + $intexp_2$]] = $\lambda\sigma$.(**IE** [[$intexp_1$]] σ) + (**IE** [[$intexp_2$]] σ)
BE[[*BoolLit*]] = $\lambda\sigma$. **B** [[*BoolLit*]]
BE[[$intexp_1$ == $intexp_2$]] = $\lambda\sigma$. (**IE** [[$intexp_1$]] σ) == (**IE** [[$intexp_2$]] σ)

You are to supply the definitions of the meaning functions for the repeat loop:

C [[*repeat C until boolexp*]]

and the switch:

C [[*switch intexp cases*]]

In writing these definitions you may use the *fix* operator (which returns the least-fixed-point of its functional argument), as well as the usual functional constructs (*e.g.*, *let*, *if-then-else*). If you need to change any of the types of the meaning functions (**P**, **C**, **IE**, or **BE**) be sure to write down the new types; if you need to add a new kind of meaning function, write its type, too.

Question 6.

Consider a DFA (deterministic finite automaton) that accepts the set of tokens of a programming language. For purposes of this question, it is convenient to think of the DFA's transition function δ as defining a labeled directed graph (or state-transition diagram) in the usual way: The nodes are the states Q; each transition $\delta(q, a) = q'$ corresponds to an edge $q \rightarrow^a q'$, labeled with a.

In addition, however, it is convenient to assume that the graph is augmented with an explicit failure node, q_{fail}, which represents a new non-final state, and that the graph is normalized as follows:

(i) Nodes (states) from which there is no path to a final-state node are said to be *useless*. All useless nodes are condensed to q_{fail}. That is, if node m is useless, edges of the form $m \rightarrow^a q'$ and $q \rightarrow^b m$ are replaced by edges of the form $q_{fail} \rightarrow^a q'$ and $q \rightarrow^b q_{fail}$, respectively. (Some of these edges may be removed by normalization-step (iii) below.)

(ii) The graph is made into a "total representation" of δ: An edge of the form $q \rightarrow^c q_{fail}$ is added to the graph for each undefined transition $\delta(q, c)$.

(iii) q_{fail} is made into a sink node: All edges of the form $q_{fail} \rightarrow^a q_{fail}$ are removed from the graph.

Let us call a state s an *unbounded state* iff

- It is a non-accepting state, and
- There are an infinite number of paths that start from s and do not include a final state.

That is, the DFA contains unbounded states iff there are arbitrarily long sequences of characters that are prefixes of valid tokens, without themselves being valid tokens. (Note that q_{fail} is not an unbounded state.)

Part (a)

Give a regular expression that defines a token that might reasonably be part of some programming language and for which the DFA has an unbounded state. Show the DFA for the token, and indicate which state or states are unbounded.

Part (b)

For this part, either

(i) Give an algorithm to identify the set of unbounded states of a DFA, or

(ii) Explain how to define a collection of equations that identify the set of unbounded states of a DFA. Can a set of equations as you have defined them have more than one solution? If so, explain how would you go about solving the equations to ensure that the final solution obtained identifies exactly the unbounded states. If not, explain why they have a unique solution.

(Whichever approach you choose, you should address the general case, not just your example from Part (a).)

Part (c)

For most programming languages, the DFA for the language's tokens never has a path in it from a final state to an unbounded state. Why would it be a bad thing if there were such a path?