Question 1.

Recall that one way to do intraprocedural dataflow analysis is as follows:

- Represent the program using a flowgraph.
- Define a lattice of dataflow facts, including a meet operator.
- For each different kind of flowgraph node, give a dataflow function that reflects the effect of executing that kind of node.
- Give equations that define, for all flowgraph nodes n, n_{in} (the dataflow fact that holds before node n) and n_{out} (the dataflow fact that holds after node n).
- Find the greatest fixed point solution to the equations.

Part (a)

Assume for now that programs are "monolithic" (do not contain subprograms or subprogram calls), and that all variables are of type int.

Define a dataflow-analysis problem (by specifying a lattice, a set of functions, and a set of equations) to provide information about the *range* of values each variable might have at each point in the program. For example, after the following code

```
read x;
if (x < 0) then y = 0 else y = 10;
```

variable x should have the range [-infinity, +infinity], and variable y should have the range [0, 10].

Assume that there are (only) the following kinds of flowgraph nodes (so you should define 7 kinds of dataflow functions):

1. A (unique) enter node.
2. Read nodes of the form: **read** *variable*.
3. Write nodes of the form: **write** *variable*.
4. Predicate nodes of the form: **if** *condition*.
5. Assignment nodes of the form: *variable* := *integer constant*.
6. Assignment nodes of the form: *variable* := *variable*.
7. Assignment nodes of the form: *variable* := *some other expression*.

Define a single kind of dataflow function for node kinds 4 and 7 (i.e., do not try to take the particular condition or the particular expression into account).

Part (b)

Now assume that programs can include global variables, procedures, and procedure calls, and that parameters can be passed either by value or by reference (note: procedures do not return values, and as before, all variables are of type int). Discuss at least two ways to do (safe) range analysis given these assumptions. Be sure to consider a simple but possibly not very accurate approach as well as a more accurate approach.

Question 2.

In Java, arrays are implemented as pointers to heap-allocated storage. In Pascal, arrays can be stack allocated. A third possibility is to implement an array as a pointer to stack storage.

Part (a)

Discuss the relative advantages and disadvantage of each of these three approaches.

Part (b)

Under what conditions would it be possible for a Java array to be directly allocated on the stack, or using a pointer to an area of stack storage?

Part (c)

Assume that those Java arrays that meet the conditions you described in Part (b) are stored directly on the stack, or are implemented using pointers to stack storage. How does this affect the way array parameters must be handled (both on the calling side and the called side)? (Remember that a method with an array parameter might be called from multiple places, so the actual parameters at different call sites might be implemented in different ways.)

Question 3.

This question concerns how to analyze a program to determine which procedures have *side effects* (i.e., whether an invocation of the procedure introduces a change in the state that is observable when the procedure returns). That is, side-effect-free procedures are pure functions in the mathematical sense.

Note that for procedure *f* to be "side-effect free", the procedures called transitively from *f* must also be side-effect free.

Part (a)

Explain an optimization that can be performed on side-effect-free procedures, and describe what the savings would be.

Part (b)

In this part, assume that the language in question is C without pointers. (In particular, parameters are passed by value, and there are no constructs that introduce aliasing.)

One way to identify (a superset of) a program's side-effect-free procedures is to solve a set of equations: For each procedure *f*, there is a Boolean variable X_f that should have the value true if *f* is side-effect free.

(b)(1) Explain how to generate an appropriate system of equations for a program. (Your equations should do as good a job as possible of identifying the side-effect-free procedures.)

Illustrate your construction by means of an example.

(b)(2) Your solution to Part (b)(1) should have the property that it can generate some sets of equations that have more than one solution. Give an example program (and its set of equations) such that the program's equations have more than one solution. (Note: For your program to be an acceptable answer, all procedures should be callable (transitively) from the distinguished procedure *main*.) List all the solutions to the equations you give.

(b)(3) Recall that if you have a complete partial order *D* with a least element $\perp$, and *F* is a continuous function from *D* to *D*, a recursive equation of the form $X = F(X)$ has a least solution $\bar{X}$ (where "least" is defined with respect to the ordering relation $\sqsubseteq$ of *D*) that can be obtained as

$$(*) \quad \bar{X} = \bigsqcup_{i=0}^{\infty} F^i(\perp).$$

Given the set of equations that arise from a program *P*, as defined by your answer to Part (b)(1), define an appropriate partial order *D* and a function *F* such that the least fixed-point of *F* indicates (a superset of) the side-effect-free procedures in *P*. Be sure to define explicitly the ordering on elements of *D*, because the notion of "least" is defined with respect to this ordering.

(Hint: Recall that if a partial order *D* is finite, then (i) every monotonic function from *D* to *D* is continuous and (ii) *D* is a *complete* partial order.)

(b)(4) Recall that if *D* has no infinite ascending chains, equation (*) has operational significance—it corresponds to an iterative process for finding $\bar{X}$. For your set of equations from Part (b)(2), trace to completion the iterative process that would arise from your answer to Part (b)(3).

Question 4.

Some modern processors support *instruction predication*. That is, an instruction may be tagged with a control register. The instruction is executed if and only if the register is true (non-zero).

Part (a)

Normally three basic blocks are generated for an *if* statement (for the condition, the *then* part, and the *else* part). Describe how, using predicated instructions, an *if* statement can be flattened into a single basic block (illustrate your explanation with a small example).

Part (b)

How must your solution be extended if nested *if*s (*if*s within *if*s) are allowed (as they usually are)?

Part (c)

Under what circumstances is it safe to leave a statement in the *then* part or in the *else* part of a "flattened" *if* statement un-predicated (i.e., so that it will be executed regardless of the value of the *if* condition)?

Part (d)

Why might it be a good idea for a compiler to generate "flattened" *if* statements? Consider issues such as code scheduling and pipelining, superscalar machines that can execute many instructions per cycle, and think about whether it is a good idea to have all statements in the *then* and *else* parts be predicated, or if there are more opportunities for optimization if some statements are not predicated.

Question 5.

Consider the recursive data type *tree*, defined as follows:

A *tree* is either *empty*, or it is a *node* containing two subtrees and an integer.

Now consider the two versions of function *flatten*, given below. Both produce a list that contains the integers from tree T in preorder. (The notation k::L means cons(k, L).)

VERSION 1

```
let rec flatten1(T) =
   cases (T) of
   empty:              nil
   node(T1, k, T2): concat(flatten1(T1), k::flatten1(T2))
```

VERSION 2

```
let flatten2(T) = flat(T, nil)

let rec flat(T, L) =
   cases (T) of
   empty:              L
   node(T1, k, T2): flat(T1, k::flat(T2, L))
```

Part (a)

Assume that the running time of function *concat* (which returns the concatenation of its two list parameters) is linear in the length of its first parameter, and that the running time of *cons* is constant. What are the worst-case running times of functions *flatten1* and *flatten2*, for a tree with N nodes? (Be sure to justify each answer. If one or both of the *flatten* functions achieves its worst-case time only for certain shapes of trees, be sure to explain this, and to give an example.)

Part (b)

Recall that function *concat* can be defined as follows:

```
let rec concat(L1, L2) =
   cases (L1) of
   nil:   L2
   x::L: x::concat(L, L2)
```

and that *concat* has the following property:

for all lists A, B, and C, *concat*(*concat*(A, B), C) = *concat*(A, *concat*(B, C)).

Prove, using structural induction on T, that for all trees T, *flatten1*(T) = *flatten2*(T). Be sure to justify each step of your proof. Hint: You may find it useful to prove that

for all T, L: *flat*(T, L) = *concat*(*flat*(T, nil), L)