1) [10 pts] On homework 1, I asked about dead assignment elimination and gave the following sample solution:

   8. Give an algorithm for dead assignment elimination that exploits def/use chains to work faster than the propagation-based algorithm presented in class. What is the time complexity of your algorithm, assuming def/use chains are already constructed? What optimization opportunities, if any, are missed by your algorithm, compared to the best propagation-based algorithm presented in class?

   Given def/use chains, I would start at the ends of the chains (those defs with no downstream uses), and remove these defs (if they had no side-effects). When a def is removed, its uses on earlier defs get removed, which may make new defs (its operands) now dead, so I recursively walk backwards up the def/use chains removing nodes until I find some with side-effects or uses. This algorithm takes linear time. However, in the presence of cycles in the def/use graph, I won't necessarily find as many dead statements as the propagation-based algorithm in class, e.g. I won't discover that a dead `x := x + 1` statement in a loop is dead. This is essentially because this algorithm is pessimistic rather than optimistic: I assume that every statement is live, rather than dead, until I prove it has no uses, which means that I can reach a worse fixpoint.

   Extend my solution to work just as well as the propagation-based algorithm from class, while still working from def/use chains. What is the time complexity of your improved algorithm?

   The main opportunity missed by the basic def-use chain based algorithm is cyclic reference patterns that do not connect to other def-use chains. To identify these cycles we associate a single bit live/dead value with each definition. All defs are initially dead. Starting from all nodes with side-effects (calls, returns, assignments to globals, etc.), we march backwards from uses to defs, marking each reached definition as live. After we're done, all statements whose live/dead bit is dead can be removed. Complexity: O(V+E). We accepted linear complexity as well, since the question incorrectly asserted that the complexity of the basic algorithm was linear (it is really O(E+V)).

2) Consider a block-structured language that supports exceptions and exception handlers. Each begin-end block can have an associated exception handler, for example:

```
begin
  <statements>
  raise larry;
  <statements>
  raise the_dead;
  <statements>
except
  when foo, bar:
    <statements>
  when larry, moe, curley:
    <statements>
  when others:
    <statements>
end
```

Exceptions are only raised explicitly via the `raise` statement, as shown above. Exceptions are handled by the `when` handler attached to the nearest lexically-enclosing block; the `when others` handler handles all otherwise unhandled exceptions. After the exception is handled, execution continues with the statement following the begin-end block to which the handling handler is attached.

Note: for this question, you need not worry about how the exception handling facility would be implemented, only about how its effects on control flow would be represented.

a) [10 pts] Assuming that exceptions are always handled within the procedures where they are raised, how would you extend the representation of a procedure to model the control flow effects of `raise` statements and exception handlers, so that iterative dataflow analysis can still be performed?

Once the control flow is made explicit in the CFG, everything is fine. We introduce a handler merge and block-done-merge. Connect each raise statements to the handler merge instead of to its textual subsequent statement. After the handler merge insert code for the handlers. Also insert branches from the end of the "normal" block to the block-done-merge and from each of the handlers to the block-done-merge.

b) [10 pts] Now consider the case when an exception might not be handled by the procedure in which it originates. In this case, the exception is implicitly re-raised at the dynamically-enclosing call site. How would you extend your solution to handle this case as well, both on the callee and caller side?

We'll assume that their is some mechanism (such as returning an extra/special value) that lets the caller know that the callee has raised an exception. On the callee-side, each procedure introduces an outermost begin/except/when block that propagates all exceptions of its caller via this mechanism. On the caller-side, each call-site is immediately followed by a test for a normal vs. exceptional return. On a normal return control flow passes to the subsequent statement, exceptional returns branch to the handler-merge for the innermost handler.

c) [10 pts] What are some of the direct and indirect benefits of knowing the set of exceptions that might be raised by a procedure call?

We can get a direct benefit of streamlining the calling convention (eliminate check for exceptional return in caller, possibly eliminate the need to return an extra value from callee). Indirect benefits due to simpler control flow downstream of the call.

d) [15 pts] Define an interprocedural analysis to compute the set of exceptions that may be raised by a procedure, as well as which procedures are guaranteed to return with an exception. Discuss your method of summarization, your approach to context sensitivity, and the like. Describe the time and space complexity of your summarization process. Explain complications that arise due to recursion, if any.

We do a context-insensitive interprocedural analysis to compute for each procedure the set of exceptions which may be raised when that procedure is called. We also include a special exception _return to indicate that the procedure returned normally. To handle recursion we initialize all summaries to the empty list (best possible information) and then use a work-list based algorithm to iterate until we reach fixedpoint. Time complexity $O(n^3)$, space complexity $O(n^2)$ (assuming that the number of possible exceptions is O(n)). The intraprocedural analysis for each procedure is a linear pass over the procedure's CFG. As we encounter raise/ return statements we add the appropriate value to the procedures summary. For a procedure call we add in that procedure's summary - _return (since whether or not the callee can return normally does not impact whether or not its caller can return normally). An improvement would be to keep track of what handler block(s) enclose each statement and filter the set of exceptions appropriately.

3) a) [15 pts] Define, in lattice-theoretic terms, an intraprocedural analysis for computing the set of variables that may be used before they are defined. Define your domain of analysis, including the interesting aspects such as the top and bottom elements and the greatest-lower-bound function. Indicate the direction of analysis, the initial conditions at the start of analysis, and the key flow functions. Explain how to use the results of analysis (i.e. the information computed at program points) to report to the user which variables may be used before they're defined.

One can solve this problem with either a forward 'must-be-defined' analysis or by using a slight modification of live variables (a backwards pass). We'll take the second alternative.
Domain: set of variables
Top: empty set
Bottom: universal set
$\leq$: $\subseteq$
glb: set union
flow function: $OUT_n = IN - defs(n) \cup uses(n)$

Any variables remaining in the set on procedure entry are potentially used before defined.

b) [15 pts] How would you extend this analysis to operate interprocedurally, so that e.g. potential uses of global variables before they are assigned could be determined (assume that globals are not all initialized by default at program start-up)? Discuss your method of summarization, your approach to context sensitivity, your modification to the flow function for procedure calls to exploit interprocedural information, and the like. Describe the time and space complexity of your summarization process. Explain complications that arise due to recursion, if any.

One cheesy way out is to make a supergraph and do part a. You only got partial credit for this.

We'll do a context-insensitive interprocedural analysis using a work-list based approach to deal with recursion. We first compute for each procedure the set of global variables that it definitely defines, using the definitely-must-def analysis that I didn't define in part a). We extend its flow function for procedural calls to union in the definite-defs from the summaries of its callee procedures. Given these summaries we then use the same work-list-based, context-insensitive IP algorithm to run the live variables analysis I did define in a). Extend its flow function to kill the definitely defined variables of callee procedures and add in their used-before-defined variables. Any variables live at the entry to main are those that we want to warn the user about.

4) [10 pts] Why has no one developed a context-sensitive interprocedural MOD analysis?

Because knowing the calling context has no impact on the set of global variables modified by the procedure. (Unless as some of you pointed out, we are also doing IP constant prop, or IP alias analysis. But you didn't have to get this tricky to get full credit)

5) [10 pts] Describe a program transformation that can hoist loop-invariant conditional tests out of loops. Illustrate its effect on an example. What is its benefit? What is its cost?

Key idea is to hoist the test out of the loop and then create two copies of the loop, one specialized for the test being true, one specialized for the test being false. Benefits are runtime savings of executing the test only once, and that each loop body is smaller, thus possibly helping i-cache behavior. Costs are that there are two copies of the loop, thus increasing overall code space, and the minor cost of executing the test once (only matters if the test didn't originally dominate all loop exits). Also, this scheme is exponential in the number of tests hoisted.

6) a) [10 pts] What are some optimizations that can be done solely with may-alias information?

One really does optimization based on the converse of may-alias information (must-not-be-aliased). This info allows one to be less conservative at pointer assignments in analyses that map variables to values like available expressions and available constants; one only has to kill info about variables that might be aliased.

b) [10 pts] What are some optimizations that require must-alias information?

Strong updates of information CSE, constant folding etc. through pointers.

7) [10 pts] The presence of pointer assignments can hurt the quality of information computed by dataflow analyses. We've discussed several compiler analyses that can reduce the negative impact of such assignments. What can language designers do to reduce the impact of such pointer assignments?

A number of possible answers including static type safety, disallowing taking the address of local variables, disallowing user-level pointers entirely.

8) a) [10 pts] The Steensgaard near-linear-time alias analysis paper uses a type-inference framework based on unification to produce OK results quickly. In order to get this faster time bound, what are the key sources of loss of precision that are incurred?

A number of possible answers. What we were really looking for was imprecise treatment of structures and backflow along dataflow edges. (formals to actuals, lhs to rhs).

b) [10 pts] Computing a flow-insensitive interprocedural summary is also near-linear-time (assuming processing a statement takes near-constant time). What is the relationship between regular flow-insensitive summaries and Steensgaard's work?

It took us several tries to figure this our ourselves, so we accepted almost anything that seemed plausible and didn't say something completely false (it was OK to believe the question's incorrect assertion that flow-insensitive IP summaries are also near-linear time). We believe that the relationship is that flow-insensitive IP summary based algorithms are more precise since they don't have the back-flow problem. However in the presence of recursion the summary based analysis actually has a complexity of something like $O(n^3)$ while Steensgaard remains nearly-liner time.

c) [15 pts] Pick another flow-sensitive dataflow analysis problem that we've discussed in class that might profitably be cast in this near-linear-time analysis framework. How would you model it? What kinds of programs would be analyzed effectively by this alternative framework, and what kinds of programs wouldn't?

Constant propagation is probably the best answer here. This might work well for programs using an array library like LINPACK, where strides/sizes are passed as parameters or if routines called with constants are only called from one call site.

9) [10 pts] The Wilson & Lam paper describes a context-sensitive pointer analysis. How might this context-sensitive analysis be used to drive procedure specialization? What would be the benefit?

Note that procedure specialization means creating multiple compiled versions of a procedure, each one specialized to a particular set of callers.

We use the partial transfer functions to tell us what specializations to create and how to connect callers to callees. The benefit is that each specialized procedure has more precise alias information about its formals. We probably want to try and be selective about which specializations are actually generated, either by using profile data or static estimates of execution frequency of each procedure or by estimating the benefit of the specialization.