1) [3 pts] What's the point of modeling dataflow analyses using lattices? What are **three** different things that are gained by doing this?

   Can view many different analyses in a uniform manner, helping understanding of what's unique and common to each

   Can reason about correctness, termination, time-complexity of analyses by formalizing analyses

   Can base implementation frameworks (e.g. Sharlit, Vortex) on the theory

2) [3 pts] What is the difference between an optimistic iterative analysis and a pessimistic iterative analysis? What is the main advantage of an optimistic analysis? What advantage might a pessimistic analysis have?

   An optimistic analysis starts by assuming the top domain element (the best possible information) and then iterates to weaken the information until a fixpoint is reached. A pessimistic analysis starts by assuming the bottom domain element (the most conservative possible information) and then iterates to strengthen the information until a fixpoint is reached. An optimistic analysis will reach at least as good and sometimes better fixpoint than a pessimistic analysis. A pessimistic analysis is a safe solution at all intermediate stages, even before fixpoint is reached, so it can be stopped in the middle of iteration if analysis is taking too long without giving up all information. Another benefit of pessimistic analysis, identified by one of the students, is that transformations can be performed eagerly during analysis w/o undo support, since their preconditions will never be violated by later iterations.

3) [3 pts] In what sense does a context-insensitive analysis still provide interprocedural information about calling context?

   A context-insensitive analysis still computes a summary of all caller context information. It simply computes a single summary for all callers, as opposed to separate summaries for different (groups of) callers.

4) Both the Vortex compiler and the Sharlit system provide a toolkit for constructing data flow analyzers, based partially on the model of defining the important lattice operations and the flow functions for the problem being solved.

   a) [3 pts] What does Sharlit provide in addition to this basic framework to improve the performance of the generated analyzers?

      Sharlit includes path compression facilities, to gain the advantages of basic block summaries and even interval analysis, but in a nicely modular way.

b) [3 pts] Why is it beneficial to be able to interleave transformations with analysis, as supported by the Vortex framework? Why is this hard to do, in general?

Transformations can improve the information computed by analysis (e.g. constant folding improves constant propagation, and branch folding improves many analyses), without having to simulate the effect of the transformation as part of the analysis's flow functions.

It's hard to do transformation while trying to reach fixpoint of an analysis, as the transformations have to be either simulated or undoable. Non-local transformations or transformation that have non-local effects on the control flow graph (e.g. branch folding, code motion) are hard to handle.

5) Consider Callahan et al.'s interprocedural constant propagation algorithm to be context-sensitive.

a) [3 pts] Explain how return jump functions are a kind of total transfer function supporting context-sensitive analysis.

The jump function computes the return constant info given the argument constant infos, which is a single function that can be used by all callers to compute the constant returned from the constants passed in, for each call site separately.

b) [3 pts] Explain how you would change the algorithm to use partial transfer functions instead.

I would maintain a mapping from argument constant infos to result constant infos, reanalyzing the callee for each different combination of argument constant infos encountered in the program at a call site to the procedure being summarized.

c) [3 pts] Which model is able to produce more precise summaries, or can both models achieve the same precision (at least in theory)? Explain your answer.

The partial transfer approach can do symbolic analysis using the actual values of the constants, computing very precise result constant info for each different tuple of argument constant infos encountered in the program. But the total transfer function can emulate this, e.g. producing a summary function that is essentially the source code which can then be run on all argument constants. And the summary can be simplified to only those parts that have an impact on the result constant info, in most cases producing a summary function that's much smaller than the procedure itself.

d) [3 pts] Which approach would you choose if you wanted to support modular interprocedural analyses, where procedures could be summarized given only the summaries of their callees?

Total transfer functions, as these can be computed without knowing about callers.

e) [4 pts] Explain how you could adapt Steensgaard-style near-linear-time analysis analysis to build near-linear-time interprocedural constant propagation. What kinds of program structures would this analysis handle well, and what would it do poorly on, relative to a context-insensitive interprocedural constant propagator?

I would construct the same sort of dataflow graph as Steensgaard does. I'd introduce nodes (type variables) for variables, constants, binops, and other non-call expressions. Variables initially are labeled with top, constant expressions are labeled with that constant, and other non-constant r.h.s. calculations are labeled with bottom. Then I'd put edges between nodes, unifying them, whenever I assigned one node to another. Procedure calls cause actual nodes to be assigned to the corresponding formal nodes, and the callee's result assigned to the l.h.s. variable of the call instruction. When unifying two nodes, I'd take the meet of their labels to compute the label of the unified node. When unification was done, for unified nodes that have constant labels, I'd propagate that constant to all nodes that got unified together in that one blob. This algorithm should be near-linear time.

This would work fine on simple constant propagation, and even interprocedural constant propagation. But it could mess up if one caller passes in a non-constant argument, which unifies with the actual parameters of all other callers; if the actual parameters of the other callers were variables that happened to contain constants and were used after the call, the merging with other non-constant callers will pollute the constant info.

f) [3 pts] Which of these various approaches to interprocedural constant propagation would you recommend using in practice, to compile a C or Java program?

For simple sorts of interprocedural constant propgation, e.g. where a literal constant is passed as an argument or result, Steensgaard should work pretty well. If it fails, then it seems that a bottom-up total-transfer-function-based context-sensitive algorithm would have good modular analysis properties.

6) a) [3 pts] What are some optimizations that can be done solely with may-alias information?

Bounding the set of variables potentially affected by a store through a pointer (better side-effect analysis, for lots of analyses). Similarly, doing dead-store elimination, if all possible targets of a stored-through pointer are dead.

Performing more precise analysis of the result of a load instruction (e.g. for reaching definitions, constant propagation, and copy propagation), by limiting the results to the values stored in pointed-to variables. Similarly, replacing a load with the result of the load, if all the possible targets of the load contain the same variable/value.

Constant-folding pointer comparison operators, for the case where two pointers are known not to alias.

b) [3 pts] What are some optimizations that require must-alias information?

Doing a strong update of the target of a pointer store.

A load or store through a pointer can be replaced by a use or definition of the corresponding pointed-to variable, effecting a kind of copy propagation. If replacing a store, then this will enable strong updates.

Additionally constant-folding pointer comparison operators for the case where two pointers are known to alias.

Of course, must-alias info can do everything that may-alias info can do, but more precisely. Also, may-alias information where a pointer may-alias only one other thing is essentially must-alias information, and enables all the must-alias-specific optimizations.

7) a) [6 pts] The standard analysis for identifying loop-invariant calculations might determine that some conditional branch tests are loop-invariant. 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?

After hoisting/copying all loop invariant calculations to the loop preheader, the loop body can be duplicated for each invariant branch in the loop body (up to 2^N loops for N invariant branches). The loop-invariant branches are hoisted out of the loops, turning into a decision tree to select a copy of the loop. Each loop copy would be specialized to the particular outcomes of the invariant branches leading to that copy, folding away the branches in the appropriate manner.

For example:

Original program:

```
while(...) {
   A;
   if (invar1) {
      B;
      if (invar2) {
         C;
      } else {
         D;
      }
      E;
   } else {
      F;
   }
}
```

Transformed program:

```
if (invar1) {
  if (invar2) {
    while(...) {
      A;
      B;
      C;
      E;
    }
  } else {
    while(...) {
      A;
      B;
      D;
      E;
    }
  }
} else {
  while(...) {
    A;
    F;
  }
}
```

The benefit of this transformation is both removing a conditional branch from the loop and also optimizing each loop copy with more precise information about that particular path through the loop-invariant conditionals.

The cost is code space blowup.

b) [6 pts] In languages with heavy use of tightly recursive functions, “recursion-invariant” calculations can occur, which are recomputed on each recursive call but invariant over those calls. How might you develop an interprocedural recursion-invariant code motion optimization? Discuss the sort of analysis you’d build, as well as how you’d transform programs to exploit your analysis and accomplish the optimization. Would you recommend adding your optimization to high-performance Scheme compilers?

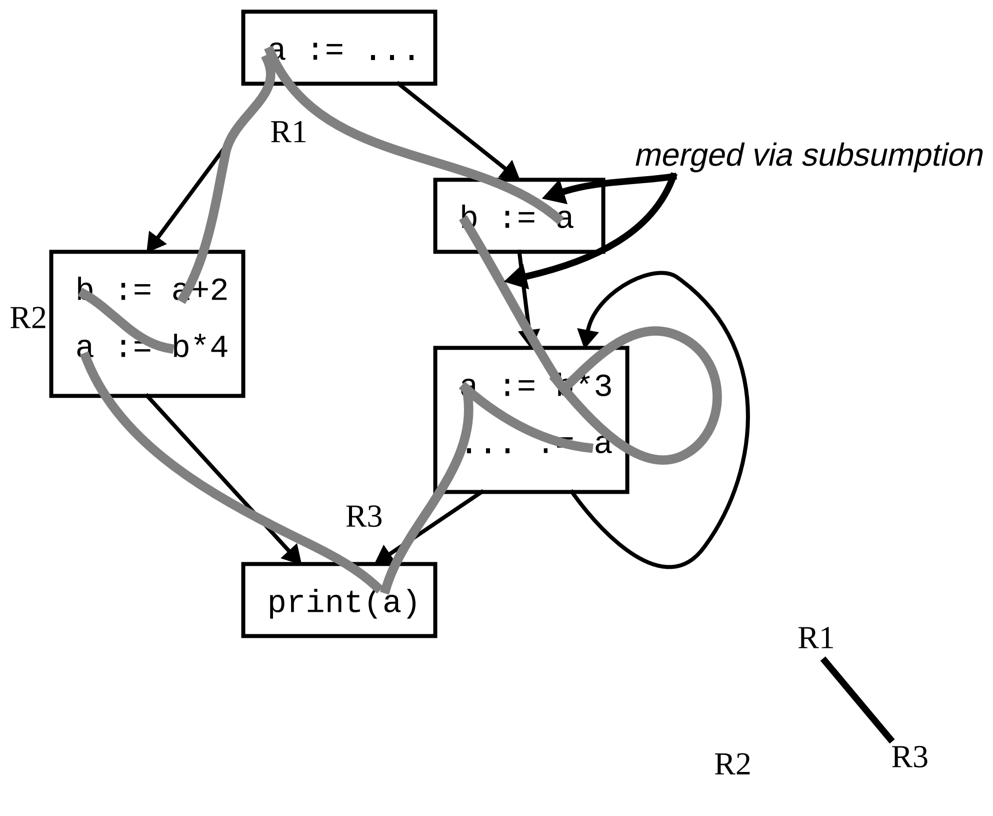
For analysis, I’d detect recursion-invariant calculations by first building a call graph. Then a recursion-invariant calculation is a pure, idempotent calculation all of whose operands are (base case) outside the recursive cycle in the call graph, or (inductive case) are themselves recursion-invariant. (The inductive case includes variables passed down as parameters from outside the recursive cycle down to the recursion-invariant calculation.)

Once I identified recursion-invariant calculations, I’d hoist them outside the recursion. I’d probably do this by adding a duplicate version of the calculation and then running interprocedural common subexpression elimination (recursion-invariant code copying). What’s that, you say? Well, it’s an optimization I just made up which passes in additional arguments corresponding to the common subexpressions that are to be reused in a callee. For each common subexpression that is to be reused in the callee, add an extra argument to the callee where the common subexpression’s result is passed in. Alternatively, I could allocate global variables to hold the common subexpression’s result.

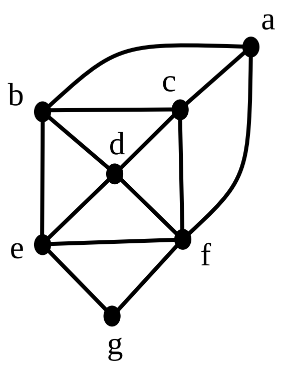
I’m not sure this is worth it, given all the parameter passing that can ensue. It would have to be a really expensive calculation to be worth the overhead.

8) a) [5 pts] For the following program fragment, draw the control flow graph, illustrate the live ranges for this graph, show which live ranges would be merged via subsumption, and draw the final interference graph for the live ranges after subsumption.

```
a := ...;
if ... then
   b := a+2;
   a := b*4;
else
   b := a;
   do
      a := b*3;
      ... := a
   while ...;
end
print(a);
```



b) [5 pts] For the following interference graph, apply Briggs's extension to Chaitin's algorithm to perform register allocation with registers `r1`, `r2`, and `r3` available for allocation. Assume references to all nodes are executed with the same frequency. Show the order in which nodes are removed from the graph during the simplification phase, and the final allocation of each node to a register or the stack. (Whenever more than one node is equally good for removal, pick the node with the lowest letter name.) For this example, does Briggs's extension avoid any spills that Chaitin's original algorithm would incur? If so, which one(s)?



Remove g (< 3 neighbors)

Remove b (max out degree)

Remove a (< 3 neighbors)

Remove c (< 3 neighbors)

Remove d (< 3 neighbors)

Remove e (< 3 neighbors)

Remove f (< 3 neighbors)

Allocate f to r1

Allocate e to r2

Allocate d to r3

Allocate c to r2

Allocate a to r3

Allocate b to r1 **** this would have been spilled in Chaitin's algorithm

Allocate g to r3

9) Consider doing register allocation for a machine which had independent integer and floating point register banks. On this machine, integer and floating point arithmetic instructions still required their operands and results to be in integer and floating point registers, respectively. Each variable in the source program is known to be either an integer or a floating point number.

a) [4 pts] How would your register allocation algorithm change to compile for this machine?

I'd essentially have two separate interference graphs and register allocation problems. Integer values go in one graph, and floats in the other. There is no interaction between the two problems.

b) [4 pts] Imagine that this machine had instructions for moving the contents of an integer register to a floating point register and vice versa, which were as cheap as regular register move instructions. How would your register allocation algorithm change?

A cheap and easy way would be to replace "spills" with attempted moves into the other register bank. But it's not clear how to account for the positions of the spills in the other bank's interference graph. Perhaps a solution would be to use a Chaitin-style algorithm where after inserting spill code (e.g. moves from one bank to another), the allocation problem is restarted. Maybe first trying spills by moving to the other bank, then in the second pass putting all spills into the stack, would work OK.

10) Consider the following program fragment:

```
r = b * b - 4 * a * c
```

Under local register allocation, assuming `r`, `b`, `a`, and `c` are in memory before & after the fragment, the following assembly code may be generated (in this assembly code syntax, destination registers are the last operand):

```
ld  [fp+offset(b)], r1
mul r1,r1,r1
ld  [fp+offset(a)], r2
shl r2,2,r2
ld  [fp+offset(c)], r3
mul r2,r3,r2
sub r1,r2,r1
st  r1,[fp+offset(r)]
```

a) [4 pts] Annotate these instructions with register actions as in Wall's algorithm.

```
ld  [fp+offset(b)], r1   REMOVE(b)
mul r1,r1,r1             OP1(b),OP2(b)
ld  [fp+offset(a)], r2   REMOVE(a)
shl r2,2,r2              OP1(a)
ld  [fp+offset(c)], r3   REMOVE(c)
mul r2,r3,r2             OP2(c)
sub r1,r2,r1             RESULT(r)
st  r1,[fp+offset(r)]    REMOVE(r)
```

b) [4 pts] Assuming that the linker decided to allocate `b` to `r7` and `r` to `r8`, show the result of applying your register actions.

```
mul r7,r7,r1
ld  [fp+offset(a)], r2
shl r2,2,r2
ld  [fp+offset(c)], r3
mul r2,r3,r2
sub r1,r2,r8
```

11) Consider the following program fragment:

```
**p + (*(q+offset) << 2)
```

which is translated into the following assembly code, after instruction selection and register allocation (`p`, `q`, and `offset` are initially in registers `r1`, `r2`, and `r3`, respectively, and the result is in register `r4`):

s_1: `ld  r1,0,r4`
s_2: `ld  r4,0,r4`
s_3: `add r2,r3,r5`
s_4: `ld  r5,0,r5`
s_5: `shl r5,2,r5`
s_6: `add r4,r5,r4`

a) [3 pts] Assuming a simple machine model where loads interlock with the following instruction if it uses the result of the load, identify the interlocking instruction pairs in the program. Assuming cache hits, how many cycles does this sequence take to execute?

```
ld  r1,0,r4
ld  r4,0,r4   *** interlocks with previous instruction
add r2,r3,r5
ld  r5,0,r5
shl r5,2,r5   *** interlock with previous instruction
add r4,r5,r4
```

8 cycles

b) [3 pts] Construct the data dependence graph for this program fragment. You may assume that alias analysis has determined that none of the loads are aliased.

$$\begin{tikzcd}
s_1\text{:ld} \arrow[d] & & s_3\text{:add} \arrow[d] \\
s_2\text{:ld} \arrow[ddr] & & s_4\text{:ld} \arrow[d] \\
& & s_5\text{:shl} \arrow[dl] \\
& s_6\text{:add} &
\end{tikzcd}$$

c) [4 pts] Schedule these instructions using the Gibbons & Muchnick list-scheduling algorithm and heuristics. For each instruction chosen, show the list of candidates from which it was chosen, and indicate which heuristic rule was used to select the particular instruction from the candidates list, as was done in class.

```
Candidates:  Selection:        Reason:
{s1,s3}      s1: ld  r1,0,r4   interlocks w/ successor
{s2,s3}      s3: add r2,r3,r5  doesn't interlock w/ prev instr
{s2,s4}      s4: ld  r5,0,r5   on longest critical path
{s2,s5}      s2: ld  r4,0,r4   doesn't interlock w/ prev inst
{s5}         s5: shl r5,2,r5   no choice
{s6}         s6: add r4,r5,r4  no choice
```

d) [3 pts] What are the interlocking instruction pairs in the scheduled program? How many cycles does the scheduled program take to execute?

No interlocks. 6 cycles.

e) [5 pts] If loads required a 2-cycle delay to avoid an interlock instead of a 1-cycle delay, how would your algorithm change? Show the results of your revised algorithm on the original unscheduled code sequence above, identify the interlocks (& their duration), and report how many cycles the scheduled program takes to execute.

I'd add a "doesn't interlock w/ instruction 2 earlier" heuristic after the initial "doesn't interlock w/ previous instruction" heuristic and before all other heuristics.

The schedule doesn't change, only one of the reasons:

```
Candidates:  Selection:        Reason:
{s1,s3}      s1: ld  r1,0,r4   interlocks w/ successor
{s2,s3}      s3: add r2,r3,r5  doesn't interlock w/ prev instr
{s2,s4}      s4: ld  r5,0,r5   doesn't i-lock w/ 2 prev instr
{s2,s5}      s2: ld  r4,0,r4   doesn't interlock w/ prev inst
{s5}         s5: shl r5,2,r5   no choice
{s6}         s6: add r4,r5,r4  no choice
```

```
s1: ld  r1,0,r4
s3: add r2,r3,r5
s4: ld  r5,0,r5
s2: ld  r4,0,r4
s5: shl r5,2,r5   **** one-cycle interlock w/ s4
s6: add r4,r5,r4  [would have an interlock w/ s2, but previous
                   interlock put in enough delay]
```

7 cycles

12) [4 pts] To provide garbage collection for a system that compiled Scheme to C, Joel Bartlett developed a partially-conservative garbage collector where pointers in the heap were known (via explicit tagging), but pointers on the stack and in registers were ambiguous (since the actions of the C compiler were unknown). Bartlett's collector treated possible pointers in registers and on the stack conservatively, but used non-conservative techniques to deal with pointers once it started scanning the heap.

Bartlett's system is a "mostly-copying" collector. Why is the "copying" part surprising? Why is it only "mostly"?

It's surprising because conservative collectors aren't normally copying, since they can't change any pointers unless they're sure it's a pointer. Bartlett's system can copy objects pointed to only by heap objects, but not objects pointed to (possibly) from ambiguous roots on the stack or in registers. These objects must be pinned in place.