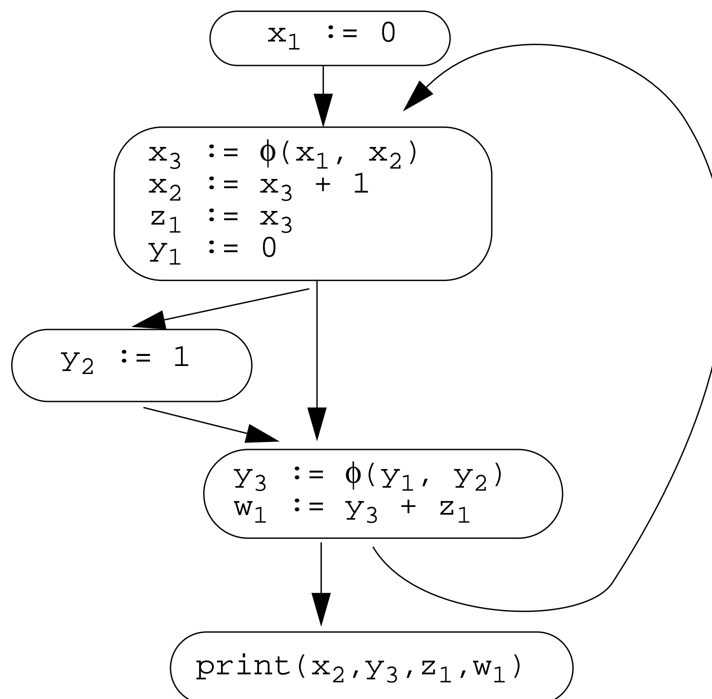
240 points total. Open book, open notes. Maximum 4 hours in a single block, self-timed on the honor system. One point per minute of test time.

1) a) [10 pts] Put the following program in SSA form (you may draw a control flow graph to illustrate your solution):

```
x := 0;
do {
   x := x + 1;
   z := x;
   y := 0;
   if (...) {
      y := 1;
   }
   w := y + z;
} while (...);
print(x, y, z, w);
```



b) [5 pts] Why would you want a program in SSA form?

Simplify analysis. Since each variable is only assigned once, each variable name only refers to a single value (which cannot be killed/redefined).

c) [5 pts] Why wouldn't you want to keep programs in SSA form?

Have to maintain SSA invariant across transformations

SSA is not executable; have to translate ϕ nodes before code gen

2) a) [10 pts] What is the data dependence graph, using dependence **distances**, for the following loop nest:

```
for i = 1 to N
   for j = 1 to N
      for k = 1 to N
         S1: a[i,j] := a[i,j] + b[j,k] * c[i,k];
         S2: c[i+1,k-1] := c[i,k] * 2;
         S3: b[j,k] := b[j+3,k-2] * 5;
```

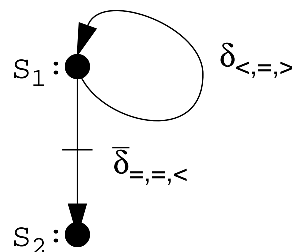
Tricky question due to 2D arrays acting like scalars with respect to one of the 3 loops. No one (including Dave) got this question completely right.

$S_1\ \bar{\delta}_{0,0,0}\ S_1$
$S_1\ \delta_{0,0,1+}\ S_1$
$S_1\ \delta^{o}_{0,0,1+}\ S_1$
$S_2\ \delta_{1,0,-1}\ S_1$
$S_2\ \delta_{1,0,-1}\ S_2$
$S_2\ \delta^{o}_{0,1+,1}\ S_2$
$S_1\ \bar{\delta}_{0,0,0}\ S_3$
$S_3\ \bar{\delta}_{0,3,-2}\ S_3$
$S_3\ \delta_{1+,0,0}\ S_1$
$S_3\ \delta_{1+,-3,2}\ S_3$
$S_3\ \delta^{o}_{1+,0,0}\ S_3$

b) [10 pts] For the following skeleton of a loop nest:

```
for i = 1 to N
   for j = 1 to N
      for k = 1 to N
         S1: ...
         S2: ...
```

assume the data dependence graph, using dependence directions, is the following:



how would you transform this loop, using loop interchange(s), to best parallelize this loop? Explain why your transformation is legal and works well, and why other possible transformations (including doing nothing) are worse or illegal.

Only the j loop can be executed in parallel (i and k have loop-carried dependencies), so we want to move the j loop to the most advantageous position. We'll assume that N># of available processors.

There are 6 possibilites (from outer to inner):

- k,i,j; k,j,i; j,k,i: These three are illegal due to lexographically negative deps.
- j,i,k: Best legal permutation for multi-processor. Deps are all lex. non-negative and we get N large chunks of parallel work.
- i,j,k: (unchanged). Could do this and run j loop in parallel but get smaller unit of parallelism, more synchronization barriers
- i,k,j: Best legal permutation for a vector machine (inner loop is parallelized), but lousy for multi-processor (many small chunks of parallel work).

3) Consider the following loop fragment:

```
for i = 1 to N
   S1: temp := ...;
   ...
   S5: ... := ... temp ...;
end
/* temp is dead here */
```

a) [5 pts] What data dependences exist between statements S_1 and S_5, based solely on the references to temp?

$S_1 \ \delta_= \ S_5$

$S_5 \ \bar{\delta}_< \ S_1$

$S_5 \ \delta^o_< \ S_1$ (not required for full credit)

b) [5 pts] Why can't this loop be fully parallelized directly?

loop-carried dependence

c) [10 pts] What is a transformation to this program that will enable full parallelization? What is the data dependence graph for your transformed loop?

Padua and Wolfe call it "node splitting." We introduce a temporary array to remove the anti-dependence:

```
S1:  temp[i] := ...;

...

S5:  ... := ... temp[i] ...;
```

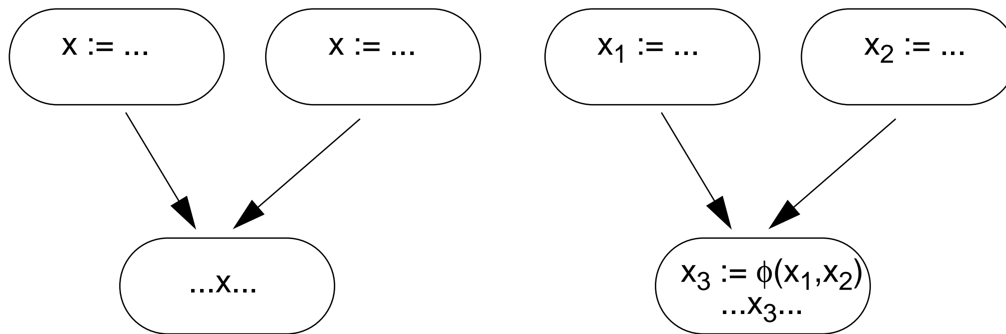
$S_1 \ \delta_= \ S_5$

4) a) [5 pts] Why is it impossible for two live ranges for the same source variable to interfere?

If the live ranges interfere, then they must be simultaneously live, which implies that the two live ranges are really part of the same (larger) live range.

b) [10 pts] A common choice for the basic unit of allocation in a register allocator is the live range. An alternative is to convert the program to SSA form and then use variables as the units of allocation. How do these choices differ? Give an example illustrating the difference.

merges with defs flowing in on multiple edges. In the example below, live ranges gives us 1 unit of allocation and SSA gives us three units.



c) [10 pts] Chaitin's original graph-coloring register allocator combined live ranges that were disjoint and linked by a simple assignment statement (called subsumption or coalescing), essentially running a kind of copy propagation to merge disjoint but adjacent live ranges. How can this merging produce better results than not merging? How can it produce worse results?

It can produce better results by eliminating register moves (by forcing the live ranges to be assigned to the same register). It can produce worse results by lengthing live ranges, leading to a more constrained interference graph (fewer nodes with higher degree), which leaves less flexibility, making it harder to avoid spills.

5) a) [10 pts] What is a good reason why you would want to perform register allocation before instruction scheduling?

To enable the loads/stores introduced as spill code to be scheduled

b) [10 pts] What is a good reason why you'd want to do the opposite, performing instruction scheduling before register allocation?

Register allocation introduces false dependencies (by reusing registers) that may inhibit scheduling.

6) a) [10 pts] How can mark/compact or copying garbage collection make allocation faster?

All free space is contiguous, so a simple pointer bump suffices to allocate an object

b) [10 pts] How can mark/compact or copying garbage collection make subsequent pointer dereferences in the application program go faster?

By improving locality (live objects get closer together)

7) Implementing first-class functions in a language like ML can be expensive, in general requiring heap-allocated closure objects and heap-allocated environment objects. Say you are a language designer who wants first-class functions but doesn't want to incur the cost of fully-general functions. [In this question, treat "ML" as refering to any statically-typed language with first-class function values.]

a) [10 pts] What is a minimal restriction on ML, statically enforceable, that would avoid having to create closures, instead enabling function values to be represented simply as a code address as in C? What capabilities would be sacrificed with this restriction?

The bodies of nested functions cannot contain references to non-global variables that are defined in lexically enclosing scopes. However, the bodies may still refer to global variables, and the function values are first class (can be stored in any variable/heap cell and returned from functions). Can easily enforce this during name resolution phase of compilation.

b) [10 pts] What is a minimal restriction on ML, statically enforceable, that would enable all environments to be stack-allocated? What capabilities would be sacrificed with this restriction?

Must enforce a LIFO-usage of function values. Function values cannot be stored into heap-allocated memory or variables defined in lexically-enclosing scopes, and function values may not be returned as the result of a function. This can be enforced by the typechecker. There are no restrictions on variable references in the function's body, and the function may still be passed as an actual parameter.

8) a) [10 pts] Say you had branch frequencies derived from profile data, giving the percentage of executions at which a branch went to each of its successors. Give a couple of examples of how you would exploit this information in a compiler.

In general, any part of the compiler that needs frequency guesses could benefit. For example the scheduler could ensure that the common successor of a branch instruction was fall-through thus improving i-cache behavior. It enables trace scheduling (scheduling across basic block boundaries). The register allocator could use this information for usage estimation.

b) [10 pts] Say you had counts for each call arc in a program, derived from profile data. Give a couple of examples of how you would exploit this information in a compiler.

To guide inlining, procedure specialization.

Reduce compile time by only using high levels of optimization on frequently executed procedures.

Improve Wall's link-time register allocation (better guess about which variables should be promoted to being in registers).

9) A basic analysis for object-oriented languages is class analysis, where a set of concrete classes is associated with each variable at each program point.

a) [15 pts] Formalize an intraprocedural version of this analysis using a lattice-theoretic framework, where you may assume you have complete knowledge of the whole program's class hierarchy. Define your domain (recall that a domain is defined by a set of elements and an ordering operator over elements) whose elements are associated with each program point, and specify (redundantly) the top & bottom elements of your domain and your lattice meet (glb) operator to use at merge points. Indicate the direction of analysis and the initial domain element at the start of analysis (taking into account that the procedure has formals f_1, ..., f_N). Specify a flow function for each of the following RTL instructions, derived from a Java-like language:

```
x := new C;      /* where C is a concrete class */
x := y;
x := y.msg(z1, ..., zk);
x := (C) y;   /* a checked narrow, succeeding if y is an instance of C or some
                  subclass of C, otherwise terminating the program */
```

First we'll define the lattice for a single variable.

set of domain elements: Powerset of classes defined in the program

$\leq$: $\supseteq$

top: empty set

bottom: universal set

meet: $\cup$

Then we define the domain for the analysis by applying the n-tuple operator to the domain I defined above (n = # of variables in procedure).

The analysis direction is forward, and the initial information maps each incoming formal *f* to $\perp$ if *f* is unconstrained and to {C|C is a subclass of specializer(*f*)} if *f* is constrained.

Flow functions. I'll use the notation Info_2 := Info_1[x := {a,b}] to denote that Info_2 is the same tuple as Info_1, except for the new value of its x component. Similarly Info(x) is the value in the x component of the Info tuple.

I'm also going to assume that we're compiling a dynamically typed language (or we're not going to take advantage of static type declarations if we aren't).

`x := new C:`

Out := In[x := {C}]

```
x := y;
```
Out := In[x := In(y)]

```
x := y.msg(z1, ..., zk);
```
Out := In[x := ⊥]

```
x := (C) y;
```
Out := In[x := {c∈In(y) | c is a subclass of C}]

b) [15 pts] Extend this analysis to be interprocedural, flow-sensitive, context-insensitive, and optimistic. Explain your initial conditions for analysis. Describe briefly how you organize your worklist, and describe briefly how you process a procedure, using (a possibly modified version of) your intraprocedural analysis above as a subroutine.

Have a summary for each procedure that contains lattice elements for local vars, formals, and the proc's return value. Each instance variable gets an associated lattice element as well. Represent global variables as locals of a 'top-scope" node. Initally all summaries have all variables bound to T. Put "top-scope" (to handle initialized global variables) and main nodes on worklist. Remove node from worklist and process it using modified intra analysis until the queue is empty.

Only major change is the flow function for msg sends/proc calls.

1) do compile time method lookup using current In(y). this gives us a set of potential callee procedures

2) Bind x to the union of the return values of the callee procedures. Add dependency links to force reanalysis (by adding dependents to the worklist) if a return value changes

3) Meet actuals at call-site with formals of each callee method, reanalyzing the method by putting it on the worklist (and updating its incoming formal info) if meet is lower in lattice than current value of formals.

Also change instance variable load/store functions to access associated lattice element (setting up dependency links on readers).

c) [5 pts] How might context-sensitive analysis improve the quality of this analysis?

The direct benefit is improved analysis of the caller procedure, since cardinality of the set of classes returned from the callee procedure may decrease. Indirectly we may get less smearing of formal parameters in the callee (smaller sets of classes for incoming formals). We might also improve the analysis of references to lexically-enclosing variables becuase there are now multiple sets of classes (one per contour) associated with each variable.

d) [5 pts] How might procedure specialization improve the resulting code quality even more?

Get more precise information (smaller class sets) for the formal parameters of the procedure, thus enabling more of the messages sent to formal parameters to be statically bound.

e) [10 pts] Some languages, such as Java, Smalltalk, and CLOS, allow new classes to be created or dynamically loaded at run-time. How would this complicate your analysis? What parts of your analysis could you keep, and what parts would you have to weaken or give up?

The simpliest approach would be to disable any parts of the analysis that make use of whole program imformation. The initial info binds formals to $\perp$. The flow function for checked narrow does nothing.

We could keep interprocedural class analysis almost unchanged, after some modifications to our intermediate language. We extend the new operation to allow expressions in addition to class literals. The lattice is extended to track each instance of class Class separately, with a special $Class_{\perp}$ that represents an unknown instance of class Class. We model the result of a class creation/load as $Class_{\perp}$. The result of a new operation on a class expression whose possible values include $Class_{\perp}$ is $\perp$. If we have a message whose receiver is $\perp$, this means that a possibly unknown callee procedure might be invoked. A conservative approximation of this is to assume that the procedure returns $\perp$, stores $\perp$ into all global variables, and that all instance variables reachable from the message sends actual parameters and all global variables are also bound to $\perp$. Unclear how much benefit we'd actually get from doing this analysis, but if class loading is rare, or very localized there might be some.

f) [5 pts] How might dynamic compilation as in the Self system be exploited to limit the impact of this language feature?

Keep dependency information about which code depends on what assumptions about the class hierarchy and recompile the code when its assumptions are violated.

g) [10 pts] Some languages, like Smalltalk and CLOS, allow the class of an object to be changed at run-time as a side-effect. For instance, the RTL statement

```
change_class(x, C);
```

would side-effect the object denoted by `x` to be an instance of class `C` instead of whatever it used to be. How would such a language feature complicate your analysis?

In the intra case, in addition to forgetting all information about exposed variables (vars defined in lexically enclosing environments), would also need to forget info about the actual parameters and their aliases. A conservative approach would be to forget everything at procedure calls. Could improve on this by doing IP analysis to determine if a particular call-site might actually do this thing. The interproc analysis probably just needs to be modified to allow backflow along dataflow arcs in this special case.

Actual flow function for the change_class node binds x (and all must-aliases of x) to {C} and adds C to the class sets of all may-aliases of x.

10) [10 pts] Why is multiple inheritance difficult to implement efficiently?

Cannot use efficient (no padding) object layout and still get the property that instance tables/virtual function tables are at a known offset.