Due **Thursday, March 15, 10:30am**. Turn in to Craig Chambers or slide it under his office door.

Take over a total of **6 hours**, self-timed. Time begins when you look at the next page. You may have a **single pause period** in the middle of your 6-hour block, of arbitrary length. During your pause, you are welcome to think about the exam in your head, and you can ask the course staff for clarifications of questions, but you cannot look at the exam, your answers, or any other materials related to this class, nor can you write down anything related to your exam, aside from taking note of clarifications.

During your 6-hour period, you may refer to any of your notes, handouts, lecture slides, course readings, or sample solutions to this year's homework. You may not discuss these questions with anyone else (other than asking the course staff for clarification), nor may you try to find solutions to these problems elsewhere, e.g., on the web or from previous years' exams or homeworks.

For short-answer questions, you shouldn't need to write more than 100 words or so, and for most questions a few dozen words in telegraph-ese should be sufficient.

You should turn in a paper copy of your solutions. You may develop all or part of your solutions on-line, as long as you turn in a print-out. Make sure you do not develop your solutions using software that does not reasonably word-wrap long lines.

100 points total.

Good luck!

1) [3 pts] The Alpern et al. paper and the VDG paper both discuss adding selector operands to ϕ nodes (called ϕ_{if} nodes in the Alpern et al. paper and γ nodes in the VDG paper). Explain how this additional operand makes these nodes amenable to optimization by common subexpression elimination and by loop-invariant code motion.

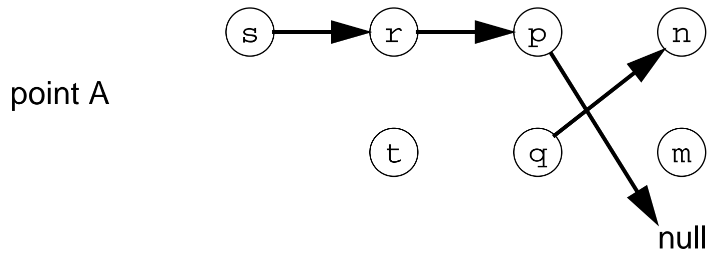
   Phi nodes with selector operands can be treated like regular nodes, not oracles whose behavior is mysterious. So two phi nodes with the same operand dataflow edges (including the same selector) are known to compute the same result, i.e. are common subexpressions. Such extended phi functions also can be treated as loop invariant expressions under the same conditions as other pure, side-effect-free operators.

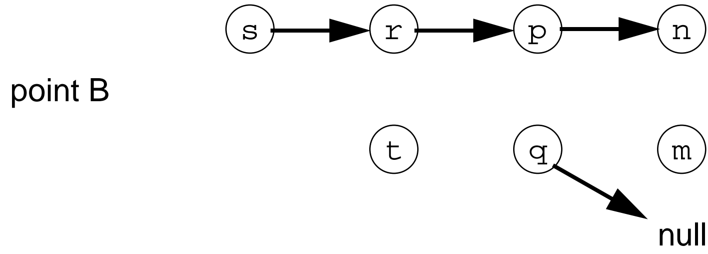
2) [2 pts] What is the difference between a strong update and a weak update? When is it safe to perform a strong update? When is it safe to perform a weak update?

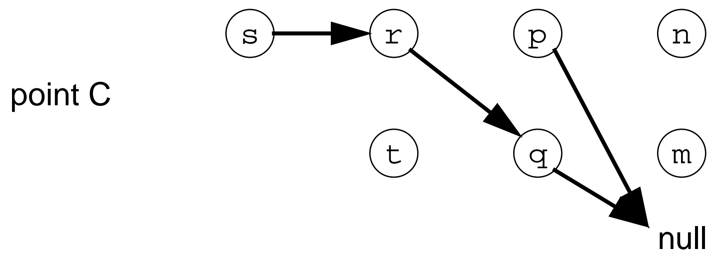
   A strong update (to a variable, or a memory location, or ...) allows old information about that variable etc. to be thrown away before the new information is added, while a weak update requires the old information to be retained and the new information only added. A strong update is safe only when it is known that the variable etc. is definitely being updated. A weak update is always safe.

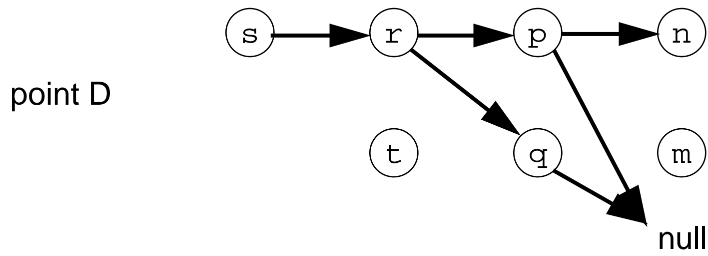
3) [6 pts] Draw the may-point-to graphs computed for each of the five marked program points in the following program. You should track explicitly which pointers might be null at each program point.

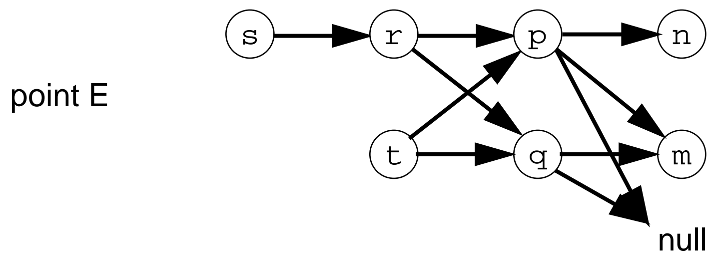
```
int n = 3;
int m = 4;
int* p = null;
int* q = &n;
int** r = &p;
int*** s = &r;
// point A
if randomBool() then
   *r = q;
   q = null;
   // point B
else
   q = null;
   r = &q;
   // point C
end
// point D
*r = &m;
int** t = *s;
// point E
```


point A
s
r
p
n
t
q
m
null


point B
s
r
p
n
t
q
m
null


point C
s
r
p
n
t
q
m
null


point D
s
r
p
n
t
q
m
null


point E
s
r
p
n
t
q
m
null

4) You are asked to modify your intraprocedural constant propagation and may-point-to analyses to handle pointers to structures with fields, in place of regular pointers. Your analyses should be defined over a CFG. Programs use the following instructions to manipulate structures and pointers:

```
p := &s        // take the address of a local variable s, which is a structure
p := new      // allocate a new structure on the heap, and return its address
x := p->f      // read from f field of the structure pointed to by p
p->f := x      // assign into the f field of the structure pointed to by p
```

You can assume that programs will only assign into fields that the target structures have, and will only read the contents of fields that have been previously assigned. Allocation leaves all fields undefined (so an assignment to a field of the newly allocated structure is required before it can be read). There are no null pointers in this language, only undefined pointers that you can assume won't be referenced before assignment. Also, there is no way to assign to a field of or otherwise reference a local variable structure directly, only indirectly through its pointer; i.e., there are no `p.f` references, only `p->f`, nor are there any direct assignments of whole structures.

a) [7 pts] Define the domain of your may-point-to analysis, in lattice-theoretic terms, and also indicate the top and bottom lattice elements (if they exist), the meet function, and the height of the lattice. You do not need to be able to distinguish heap-allocated structures. (You might want to follow Sorin's checklist to make sure you've defined your domain fully.)

At each program point, we want to capture the set of structures each pointer might point to, plus the set of things each field of the target structures might point to. So we'll define a domain for the sources of pointers (Src) as the union of the set of variables (Var) and the set of Struct $\times$ Field pairs, where an element of Struct is either a variable (for a stack structure) or the distinguished value HeapStruct, which is a summary node standing for all heap-allocated structures, and where Field is the set of field names. The domain for the targets of pointers (Dest) is Struct. Then at each program point we compute a map from Src to set of Dest.

More precisely:

Assume Var is the set of variables in the program and Field is the set of field names in the program.

Let set Struct = Var $\cup$ {HeapStruct}

Let set Src = Var $\cup$ (Struct $\times$ Field)

Let set Dest = Struct

Then domain MPT = $\langle Src \rightarrow pow(Dest), \leq_{MPT} \rangle$

To make it easy to define our functions on this domain, we require the map to be a total map, where every element of Src maps to some set, possibly the empty set.

One may-point-to set m1 is more conservative than another m2 if m1 maps each source s to a superset of what m2 maps s to:

$m1 \leq_{MPT} m2 \Leftrightarrow \forall s \in dom(m1).\ m1(s) \supseteq m2(s)$

The top element of the MPT domain is the map from every variable and struct/field pair to the empty set, and the bottom element is the complete map, mapping every variable and struct/field pair to the set of all structs. The merge function computes a map that maps each source to the union of the targets of that source in the merging maps. The height of the lattice is |Dest| * |Src| + 1, or (|Var|+1) * (|Var|+(|Var|+1) * |Field|) + 1, which is $O(|Var|^2\ |Field|)$.

b) [5 pts] Give flow functions, expressed as constraints relating `pred` and `succ` domain values, for your may-point-to analysis for the four instructions listed above. Of course, your flow functions must be monotonic.

$MPT_{\texttt{p := \&s}}$:
```
   succ = pred - {p → *} ∪ {p → s}
```

$MPT_{\texttt{p := new}}$:
```
   succ = pred - {p → *} ∪ {p → HeapStruct}
```

To analyze a reference from a structure field, find out what the structures can be, and then find out what their f fields can point to, and update the lhs to point to the union of all that:

```
MPTx := p->f:
   succ = pred
      - {x → *}
      ∪ {x → y | s ∈ pred(p) ∧ y ∈ pred((s,f)) }
```

To analyze an assignment to a structure field, if it's a strong update (i.e., p points to a single structure, which isn't a heap summary node), then forget what we used to know about the f field of this structure. Then add in the info that the f field of the structures that p may point to now can point to what x points to:

```
MPTp->f := x:
   succ = pred
      - (if pred(p)={s} and s != HeapStruct
           then {(s,f) → *} else {})
      ∪ {(s1,f) → s2 | s1 ∈ pred(p) ∧ s2 ∈ pred(x) }
```

c) [5 pts] Define the domain of your constant propagation analysis, in lattice-theoretic terms, and also indicate the top and bottom lattice elements (if they exist), the meet function, and the height of the lattice.

At each program point, we want to record not only a map from variables to the standard 3-level constant propagation lattice, but also a map from fields of structures to this lattice.

Let domain 3Level = <Constant $\cup$ {T_{3Level}, $\perp_{3Level}$}, $\leq_{3Level}$> be the standard constant propagation lattice, with a top, bottom, and infinitely wide set of incomparable constants drawn from the set Constant.

Then domain RC = <Src $\rightarrow$ 3Level, $\leq_{RC}$>

To make it easy to define our functions on this domain, we require the map to be a total map, where every element of Src maps to some element of 3Level (using top for an element of Src that is undefined, and bottom for an element of Src that is defined to something that's not known to be a constant).

One reaching-constant map m1 is more conservative than another m2 if m1 maps each source s to a more conservative 3Level element than m2:

$m1 \leq_{RC} m2 \;\Leftrightarrow\; \forall\, s \in dom(m1).\; m1(s) \leq_{3Level} m2(s)$

The top element of the RC domain is the map from every variable and struct/field pair to T_{3Level}, and the bottom element is the map from every variable and struct/field pair to $\perp_{3Level}$. The merge function computes a map that maps each source to the meet of the 3Level lattice elements of that source in the merging maps. The height of the lattice is 2 * |Src| + 1, or 2 * (|Var|+(|Var|+1) * |Field|) + 1, which is O(|Var| |Field|).

d) [5 pts] Give flow functions, expressed as constraints relating `pred` and `succ` domain values, for your constant propagation analysis for the four instructions listed above. Your analysis should be able to track the flow of constants through structure fields, at least for local variable structures. Your analysis can refer to the results of the may-point-to analysis using the variables $\texttt{pred}_{MPT}$ and $\texttt{succ}_{MPT}$.

$RC_{p := \&s}$:
  succ = pred - {p → *} ∪ {p → $\bot_{3Level}$}

$RC_{p := new}$:
  succ = pred - {p → *} ∪ {p → $\bot_{3Level}$}

To analyze a reference to field f of structures pointed to by p, find out what the structures might be, then the constants that might be in their f fields, then update the lhs with the meet of all these possible constants:

```
RCx := p->f:
   succ = pred - {x → *} ∪ {x → k}
   where k = Meet { k' | s ∈ predMPT(p) ∧ k' = pred((s,f)) }
```

To analyze an update of field f of structures pointed to by p, replace the old info about the structure field contents with new info. In the case of a strong update, the new info is just the constant info for the rhs. For a weak update, we have to meet the new constant info with the previous constant info for that field:

```
RCp->f := x:
   succ = pred
      - {(s,f) → * | s ∈ predMPT(p) }
      ∪ {(s,f) → k | s ∈ predMPT(p) ∧
           k = (if predMPT(p)={s'} and s' != HeapStruct
                then pred(x) else pred(x) meet pred((s,f))) }
```

e) [4 pts] Show the may-points-to and constant propagation information constructed by your analyses at each of the four program points labeled in the following program:

```
// s1 and s2 declared as local variable structures

p1 := &s1
p1->x := 4
p2 := &s2
p1->n := p2
p2->x := 5
p3 := p2

// point A

while randBool() do

   // point B (t and x not in scope)

   t := new
   x := p3->x
   t->x := x * 2
   p3->n := t
   p3 := t

   // point C (t and x are in scope)

end
```

```
// point D (t and x not in scope)
z1 := p1->x * p1->x
z2 := p2->x * p2->x
z3 := p3->x * p3->x
```

(I have omitted from MPT any Src elements that map to the empty set, and from RC any Src elements that map to T.)

point A:

MPT = { p1→{s1}, p2→{s2}, p3→{s2}, (s1,n)→{s2} }

RC = { p1→⊥, p2→⊥, p3→⊥, (s1,x)→4, (s1,n)→⊥, (s2,x)→5 }

point B:

MPT = { p1→{s1}, p2→{s2}, p3→{s2,HeapStruct},
(s1,n)→{s2}, (s2,n)→{HeapStruct}, (HeapStruct,n)→{HeapStruct} }

RC = { p1→⊥, p2→⊥, p3→⊥,
(s1,x)→4, (s1,n)→⊥, (s2,x)→5, (s2,n)→⊥,
(HeapStruct,x)→⊥, (HeapStruct,n)→⊥ }

point C:

MPT = { p1→{s1}, p2→{s2}, p3→{HeapStruct}, t→{HeapStruct},
(s1,n)→{s2}, (s2,n)→{HeapStruct}, (HeapStruct,n)→{HeapStruct} }

RC = { p1→⊥, p2→⊥, p3→⊥, t→⊥, x→⊥,
(s1,x)→4, (s1,n)→⊥, (s2,x)→5, (s2,n)→⊥,
(HeapStruct,x)→⊥, (HeapStruct,n)→⊥ }

point D: same as point B

f) [1 pt] Which of the `z1`, `z2`, and `z3` calculations above can be folded based on your analysis results?

z1 and z2

5) You are building a compiler that does inlining, in a top-down fashion (i.e., when a call site is encountered during compilation of a procedure, the inliner will decide whether to inline the callee procedure). Since the language you're compiling supports arbitrary recursion, you need to figure out how to prevent infinite inlining through recursive procedures.

a) [3 pts] What is a clean way to do this, while still allowing useful inlining? (An arbitrary cut-off after a fixed amount of inlining is not a clean way.)

Keep track at each program point the stack of functions that have already been inlined, plus the outer function being compiled. Then don't allow inlining of any function that is already present in this stack.

b) [3 pts] How can you extend your solution to allow a limited amount of inlining of recursive functions, akin to loop unrolling?

Allow up to k occurrences of the function in the inlining stack before blocking inlining, where k is the unrolling factor of the recursion. k=1 is the no-inlining-of-recursive-calls case above.

6) The standard definition of intraprocedural class analysis generates singleton class sets for instructions like `x := new Class`, initializes other variables to the universal set of all classes, takes the union of class sets at merge points, narrows class sets along the successor branches of instructions like `if x instanceof Class goto L`, and uses class sets when trying to optimize method calls or `instanceof` tests. This analysis is typically defined over the CFG.

a) [3 pts] In what way(s) would the algorithm perform better over def/use chains?

As with constant propagation, each edge directly connects defs to uses, so the analysis will propagate less information on each edge and more directly to the places that use it.

b) [3 pts] In what way(s) would the algorithm perform more poorly over def/use chains?

It would lose the ability to narrow class sets at instanceof tests, since such points are not explicit in the def/use chains.

c) [3 pts] Where have you seen this weakness of def/use-chain-based analyses before? What general property of an analysis leads to this weakness?

We saw it on the midterm, w.r.t. range analysis. The general situation is an analysis where the outcome of a conditional branch, or really any computation other than an assignment to a variable, has an indirect effect that improves the information known about the variable.

7) You want to develop a tool that makes it easier to understand Java programs. In particular, you'd like to be able to click on a method call, and have your tool pop up a menu of the set of methods that might be invoked by that call. You want to have as few false positives as possible.

a) [3 pts] Briefly describe how you would use the results of class hierarchy analysis to compute the set of possible called methods at a call site.

CHA will tell me the set of classes that the receiver might be, based on the set of classes in the program that are subclasses of the receiver's declared type. For each of these classes I can find the method that will be called by this method call site. I take the union of these methods, and report them to the user.

b) [3 pts] Would adding interprocedural, context-insensitive class analysis be useful? If so, briefly describe how you would exploit the results of such an analysis in your tool.

Yes, since it could be used to narrow down the set of possible receiver classes. I'd run the analysis, and then use the (hopefully smaller) set of possible receiver classes to refine the set of possible callee methods.

c) [3 pts] Would adding context-sensitivity be useful? If so, briefly describe how you would exploit the results of such an analysis in your tool.

Yes, in exactly the same way that interprocedural analysis would. I also could build a more sophisticated tool that lets the user navigate the calling-context-based call-graph of the program, in part 8a below, and ask questions about call sites within a particular version of the program.

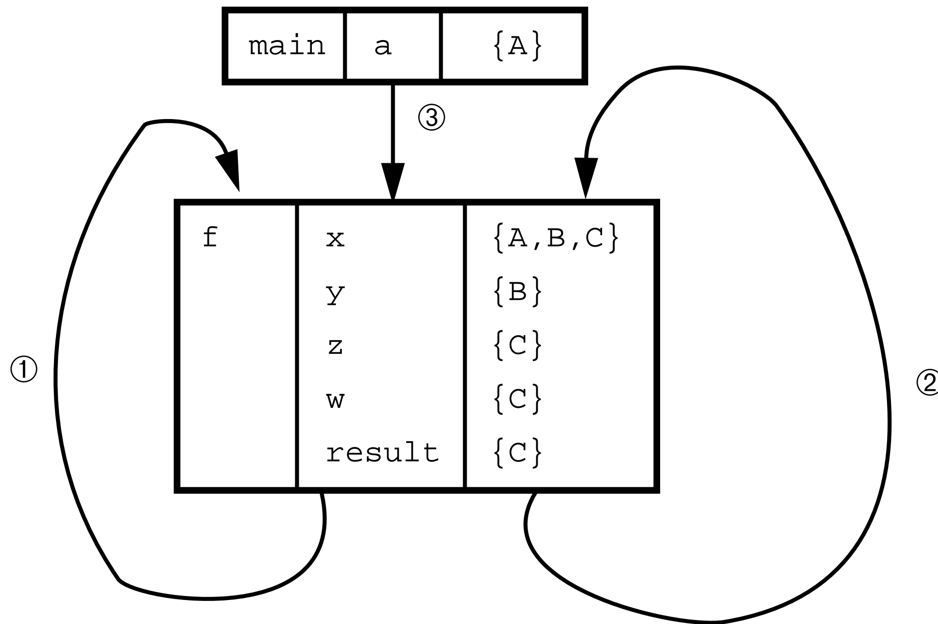
d) [3 pts] Would adding procedure specialization be useful? If so, briefly describe how you would exploit the results of such a transformation in your tool.

No, procedure specialization wouldn't help, except as a way to visualize and manipulate the calling-context-based call-graph as in part 7c above.

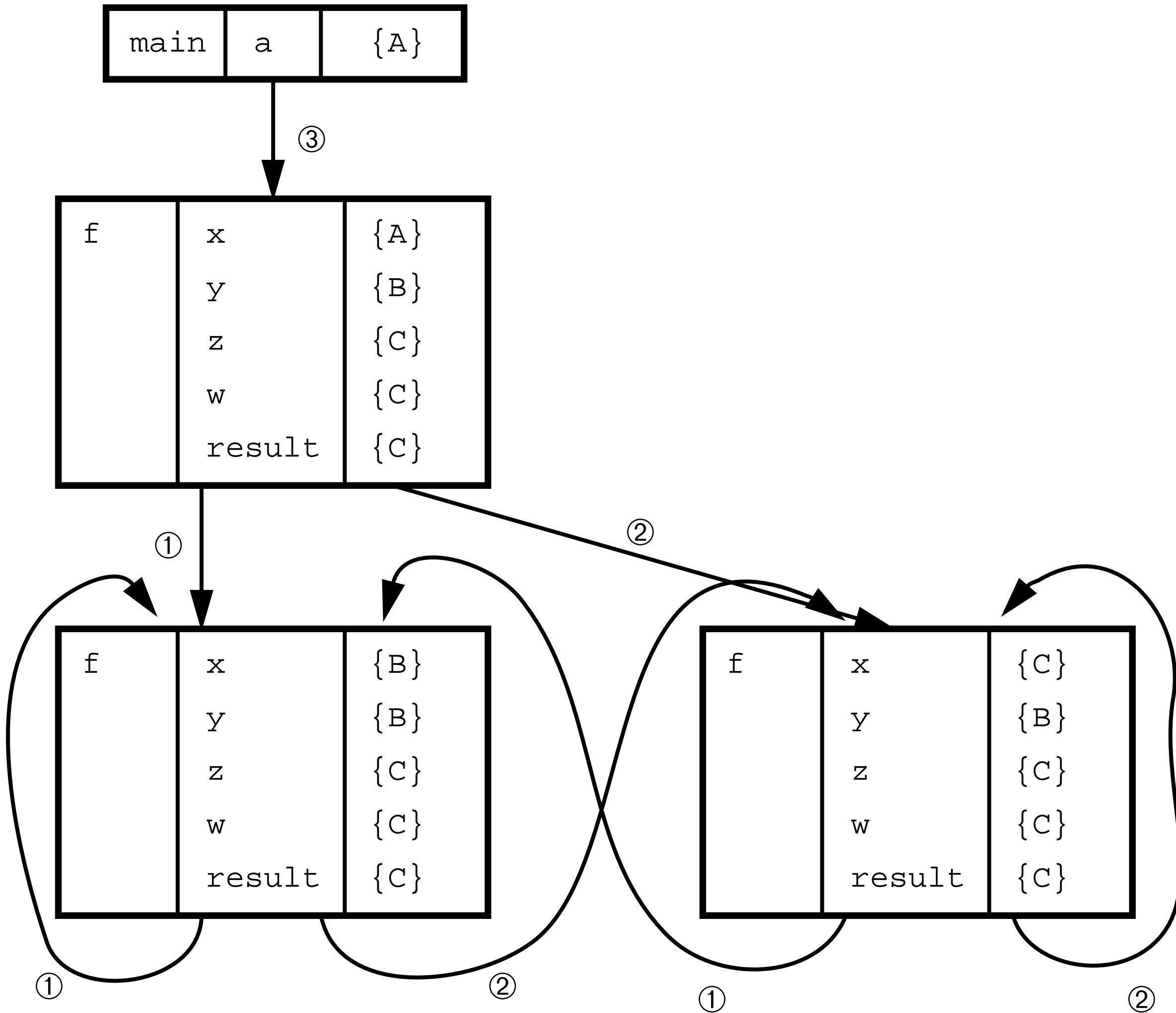
8) Consider the following Java program:

```
class A {
   public static A f(A x, int n) {
      A y = new B();
      A z = new C();
      if (n > 0) {
         A w = f(y, n-1);  // call site 1
         z = f(w, 0);      // call site 2
      }
      return z;
   }
   public static void main(String[] args) {
      A a = new A();
      f(a, 5);              // call site 3
   }
};
class B extends A {};
class C extends A {};
```

a) [4 pts] Show the results for interprocedural context-insensitive class analysis of this program. Show the call graph, with all edges labeled by call site number, and show the contents of the argument, result, and local variable class sets (ignoring primitive types and arrays). Ignore constructor calls.

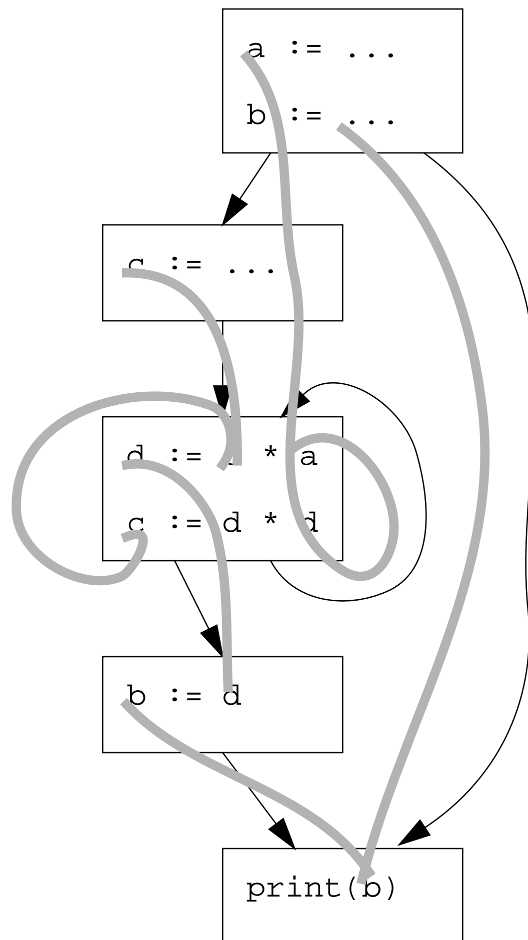


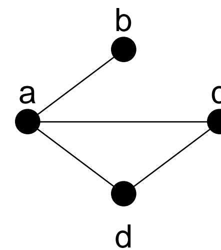
b) [5 pts] Repeat, but using the context-sensitive Cartesian Product Algorithm. Nodes in your call graph should correspond to procedures under some particular calling context; there can be multiple nodes for a single source procedure in this context-sensitive call graph.



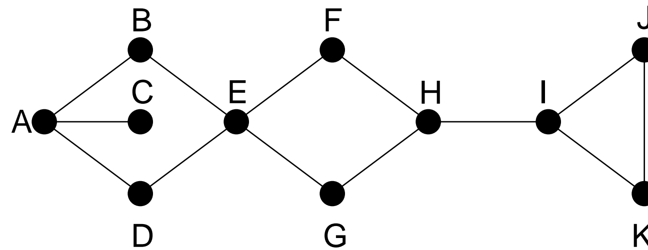
9) a) [5 pts] For the following program fragment, show the control flow graph, the live ranges of variables, and the interference graph, using live ranges as the units of allocation.

```
a := ...
b := ...
if ... then
   c := ...
   do
      d := c * a
      c := d * d
   until ...
   b := d
end
print(b)
```





b) [5 pts] For the following interference graph, execute Briggs' extension to Chaitin's algorithm to allocate registers, assuming 2 registers are available. Assume all nodes are equally frequently referenced in the underlying program, so that out-degree is the sole criterion for spill node selection. IMPORTANT: To ensure that there is a single correct solution, when simplifying or spilling, if more than one node is equally applicable, pick the node with the first label alphabetically, and when picking registers for nodes, if both registers are available, assign registers pulling from the following not-so-random sequence: r1, r2, r1, r2, r1, r2, ..., i.e., the first time you have to pick a register with no constraints, pick r1, and the second time you have to pick a register with no constraints, pick r2, and so on. Show your work, as we did in class.



Stack (grows downwards)	Register Assignment (computed bottom-up)
C	r1
(blocked) E	r2
B	r1
A	r2
D	(unconstrained) r1
F	r1
G	r1
H	(unconstrained) r2
(blocked) I	spilled
J	r2
K	(unconstrained) r1

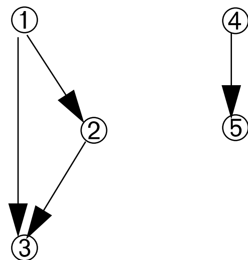
10) Consider the problem of scheduling the following basic block for a target machine where a load or a multiply takes 2 cycles, interlocking with the following instruction for a cycle if that instruction uses the result of the load or multiply.

```
① x := *p
② y := x*x
③ z := x+y
④ u := *q
⑤ v := u+5
// p, q, z, and v live at this point
```

a) [2 pts] Identify the interlocks (if any) in this schedule.

after instructions 1, 2, and 4

b) [3 pts] Show the data dependence graph for these statements.



c) [4 pts] Perform the list scheduling algorithm to construct a different schedule. At each step in the algorithm, identify the candidate instructions, the best instruction selected, and the heuristic rules (if any) that pruned instructions out of the candidate set on the way to selecting the best instruction. If more than one instruction is equally good (or bad), indicate the other instructions that could have been selected as best.

Candidates	Best	Reason(s)
1,4	1	3
2,4	4	1
2,5	2	1
3,5	5	1
3	3	--

d) [1 pt] Show the new schedule (writing out the instructions explictly), and identify the interlocks (if any) in the new schedule.

```
① x := *p
④ u := *q
② y := x*x
⑤ v := u+5
③ z := x+y
```

No interlocks!

11) a) [3 pts] Programs in my newly designed language have a high allocation rate and death rate. Programs are heavily interactive, and I don't want to be embarrassed by pauses due to garbage collection. What kind of automatic GC should I use, and why?

A generational copying GC, e.g. generation scavenging, should be good, since it typically has low pause times, it has good memory locality and supports fast allocation due to copying (and thereby compacting) live data, and its generational nature will enable it to collect the space of the dying objects quickly. (This presumes that the objects that are dying are young objects, which might not be the case, e.g. if there was some FIFO pattern of use of objects.)

b) [3 pts] Programs in my other language have very low allocation rates, and relatively small heap memory demands, but programs run a long time and must have no storage leaks. I want to minimize the overall execution time of my programs, but I am not particularly concerned if there are periodic pauses for garbage collection (programs in my language are batch programs, not interactive ones). What kind of automatic GC should I use, and why?

A stop-the-world mark/sweep collector seems best, since it has low overhead and doesn't suffer from storage leaks in the face of cycles (unlike reference counting).