1) a) [6 pts] Define an efficient, effective interprocedural analysis to compute the set of exceptions raised by each procedure (directly or indirectly) and not handled by the procedure (directly or indirectly), using the model of exceptions from the midterm exam. Describe the algorithm for analyzing the body of a procedure clearly, considering its raises, its handles, its calls, and its callers. Describe how your algorithm copes with recursion intelligently. Is your analysis (intraprocedurally) flow-sensitive or -insensitive? Is it context-sensitive or -insensitive?

A bottom-up summary-based analysis, computing a set of exceptions for each procedure. The set of exceptions for a procedure is computed in a single linear scan of the procedure, with each raise not surrounded by a corresponding handler adding to the procedure's raise set, and each procedure call adding all those exceptions in its summary that aren't handled by the calling procedure to the caller's summary. To handle recursion, each procedure's exceptions list is initialized to empty, and iteration is used to reach a fixpoint (callers are reanalyzed whenever a procedure's summary changes). The analysis is flow- and context-insensitive.

b) [6 pts] Define an efficient, effective interprocedural analysis to compute the set of exceptions possibly and definitely handled by callers of a procedure. Describe the algorithm for analyzing the body of a procedure clearly, considering its raises, its handles, its calls, and its callers. Describe how your algorithm copes with recursion intelligently. Is your analysis (intraprocedurally) flow-sensitive or -insensitive? Is it context-sensitive or -insensitive?

A top-down summary-based analysis, computing a pair of sets for each procedure summarizing the set of possibly and definitely handled exceptions. Initialize each procedure's possibly handled exception set to the empty set and definitely handled exception set to the universal set (except for main, whose definitely handled exception set is empty). To analyze a procedure, just process each call site as follows. Compute the call site's exception sets by taking the enclosing procedure's exception sets and adding in all the exceptions handled by blocks enclosing the call site. Then update the callee's possibly (resp. definitely) handled exceptions set by taking the union (resp. intersection) of the callee's old set and the call site's set. (If a set changes, reanalysis of the callee is required, which handles recursion in the normal iterative manner.) (Raises are ignored by this analysis.) The analysis is flow- and context-insensitive.

c) [3 pts] Explain how to use the analyses to report to the programmer potentially unhandled exceptions.

For each procedure, for all exceptions in its exceptions set (part a) not included in its definitely handled set (part b), report the exceptions as possibly unhandled. For all the exceptions not in the possibly handled set, the exceptions can be reported as definitely unhandled.

(It would be nice to avoid reporting the same exception as unhandled for each procedure it passes through.)

2) a) [5 pts] Java and all other type-safe languages automatically perform array bounds checking. Pulling together techniques discussed in class, describe an intermediate representation for array bounds checks and an analysis and transformation that can remove unnecessary bounds checks automatically. Illustrate your representation, analysis, and successful transformation on the following example:

```
int[10] a;
for (int i = 0; i < 10; i++) {
   ... a[i] ...
}
```

I would model a bounds check as a pair of explicit comparisons of the index against the low and high bounds of the array before the array reference. (Actually, if the low bound is 0, then a single unsigned comparison against the high bound will catch both negative and large indexes.) If either comparison fails, the error branch goes off to some call of a run-time error routine. (It would be nice to be able to assert to the compiler that the error routine doesn't return, to improve the quality of dataflow information after the test.

To optimize away some of these checks, I'd do integer range analysis as discussed in class. This analysis would track a low and a high integer constant for each integer expression. After comparisons, the outcome of the comparison can be used to narrow the range of possible values (e.g. after the `i<10` comparison above, the upper bound for `i` is known to be 9). The merge operator at loops needs to attempt to preserve as much useful information about ranges as possible while converging quickly; my idea would be to increase the high bound to infinity or the low bound to -infinity, based on the direction of growth from the loop entry's range info to the back edge's range info. E.g. if at loop entry the range is `[0..0]` and at the back edge the range is `[1..1]`, a non-generalizing merge would produce a combined range of `[0..1]`. But since the range appears to be growing in a positive direction after going around the loop, the high bound is extended eagerly to infinity, producing a new range at the loop head of `[0..infinity]`. The loop will then reach fixpoint.

Integer range information can be used to constant-fold comparisons that are known to always be true or false, such as the array bounds check before the `a[i]` reference above. The range of `i` will be computed to be `[0..9]` before the array bounds checks, which leads to both checks being constant-folded away.

b) [4 pts] Does your analysis successfully handle the following example, where the size of the array is dynamically determined? If not, how might it be extended to handle it?

```
int n = ...;
int[n] a;
for (int i = 0; i < n; i++) {
   ... a[i] ...
}
```

No, it doesn't. My analysis using integer range info only handles arrays of constant length, since it only tracks constant upper and lower bounds. It would need to be extended to track symbolic upper and lower bounds, which is much more complicated.

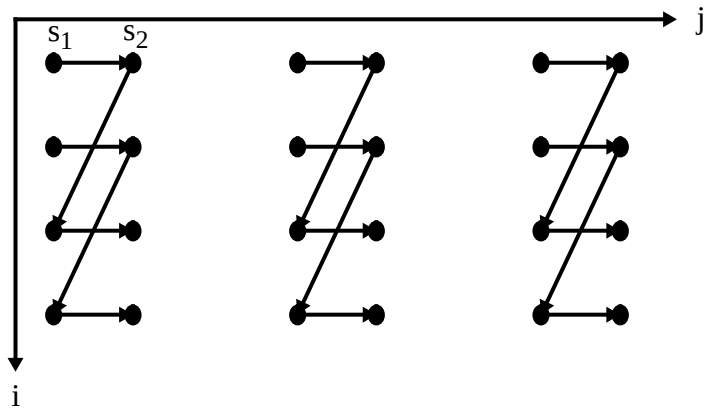
3) [5 pts] Imagine that you wish to perform context-sensitive interprocedural alias analysis, but to save analysis & implementation time you simply wish to analyze procedures under at most two possible input alias conditions: none of the formals and accessed non-local variables are aliased (best possible input alias information), and all of the formals and accessed non-local variables may be aliased (worst possible input alias information). What context sensitivity strategy would you choose for this algorithm (e.g. some variation on *k*-CFA, adaptive expansion, total transfer functions, partial transfer functions, or cartesian products)? Why is your choice a good one?

I would choose a partial transfer function-based strategy, with each procedure being analyzed under 0, 1, or 2 input contexts. This is a good strategy because it only analyzes an input context if it is used in the program (unlike a total-transfer function-based algorithm, and it shares the results of analysis across all callers with that same calling context (unlike a k-CFA-based algorithm). There's no need to do either adaptive expansion or compute cartesian products.

4) Consider the following program fragment:

```
for i := 2 to 5
   for j := 2 to 4
      s1: b[i-1,j+1] := a[i-1,j]
      s2: a[i+1,j]   := b[i-1,j+1]
   end
end
```

a) [4 pts] Draw the iteration space for this program.



b) [4 pts] Summarize the iteration space via dependences between the two statements in the loop body, using distance vectors.

$s_1\ \delta_{0,0}\ s_2$

$s_1\ \delta_{2,0}\ s_2$

c) [4 pts] Which if any of the loops can be parallelized? Why?

The inner (j) loop, because there are no loop-carried dependences on this loop (the subscripts are 0).

d) [4 pts] Can the loops be interchanged legally? Why or why not?

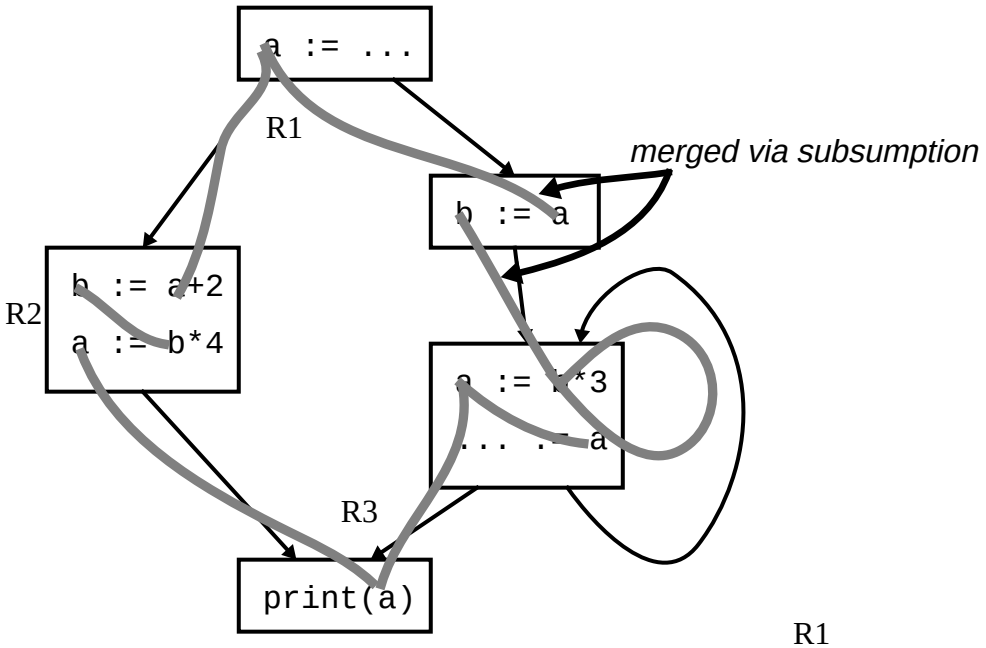
Yes, because the dependence vectors remain lexicographically non-negative after reordering the subscripts.

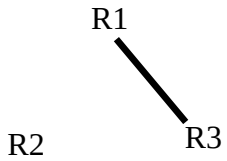
e) [3 pts] What is the most profitable sequence of loop transformations for this program, for compiling for a generic multi-processor as discussed in class?

Do loop interchange, and then parallelize the (now) outer j loop.

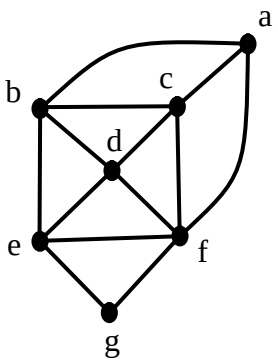
5) a) [6 pts] For the following program fragment, draw the control flow graph, illustrate the live ranges for this graph, show which live ranges would be merged via subsumption, and draw the final interference graph for the live ranges after subsumption.

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





b) [6 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

6) 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) [5 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
```

7) 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`):

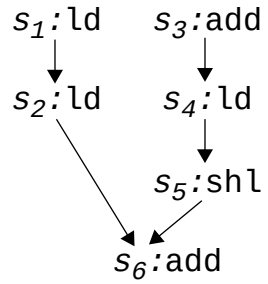
```
s1: ld  r1,0,r4
s2: ld  r4,0,r4
s3: add r2,r3,r5
s4: ld  r5,0,r5
s5: shl r5,2,r5
s6: add r4,r5,r4
```

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) [4 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.



c) [6 pts] Schedule these instructions using the Gibbons & Muchnick algorithm. 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:
{s_1,s_3}	s_1: ld r1,0,r4	interlocks w/ successor
{s_2,s_3}	s_3: add r2,r3,r5	doesn't interlock w/ prev instr
{s_2,s_4}	s_4: ld r5,0,r5	on longest critical path
{s_2,s_5}	s_2: ld r4,0,r4	doesn't interlock w/ prev inst
{s_5}	s_5: shl r5,2,r5	no choice
{s_6}	s_6: 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

f) [5 pts] Say you were scheduling for a very simple & idealized superscalar machine which can issue & execute two instructions in parallel in each cycle. The scheduler must choose pairs of instructions, such that the two instructions are independent. A load instruction in one instruction pair interlocks with an instruction in the following pair if the instruction in the following pair uses the result of the load. Explain how you'd modify the Gibbons & Muchnick algorithm for this target machine. Show the results of your algorithm when applied to the original unscheduled code sequence above, and identify the interlocks and the total number of cycles needed for execution.

When choosing from the available candidates, two instructions should be chosen (if only one candidate is available, then a nop candidate should be added). The same heuristics can be used to choose among the candidates, where the test for interlocking with the previous instruction checks both instructions of the previous pair.

```
Candidates:  Selection:                              Reason:
{s1,s3}      s1: ld  r1,0,r4; s3: add r2,r3,r5       no choice
{s2,s4}      s2: ld  r4,0,r4; s4: ld  r5,0,r5        no choice
{s5}         s5: shl r5,2,r5; nop                    no choice
{s6}         s6: add r4,r5,r4; nop                   no choice
```

```
s1: ld  r1,0,r4; s3: add r2,r3,r5
s2: ld  r4,0,r4; s4: ld  r5,0,r5  ** interlocks w/ prev instr
s5: shl r5,2,r5; nop              ** interlocks w/ prev instr
s6: add r4,r5,r4; nop
```

6 cycles

Better solutions were presented by several students, where only one non-nop instruction is selected for a pair if it doesn't interlock with the previously generated instruction pair while all other candidates do.

Candidates:	Selection:	Reason:
{s_1,s_3}	s_1: ld r1,0,r4; s_3: add r2,r3,r5	no choice
{s_2,s_4}	s_4: ld r5,0,r5; nop	doesn't i-lock
{s_2,s_5}	s_2: ld r4,0,r4; nop	doesn't i-lock
{s_5}	s_5: shl r5,2,r5; nop	no choice
{s_6}	s_6: add r4,r5,r4; nop	no choice

```
s1: ld  r1,0,r4; s3: add r2,r3,r5
s4: ld  r5,0,r5; nop
s2: ld  r4,0,r4; nop
s5: shl r5,2,r5; nop
s6: add r4,r5,r4; nop
```

No interlocks. 5 cycles.

8) [5 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.