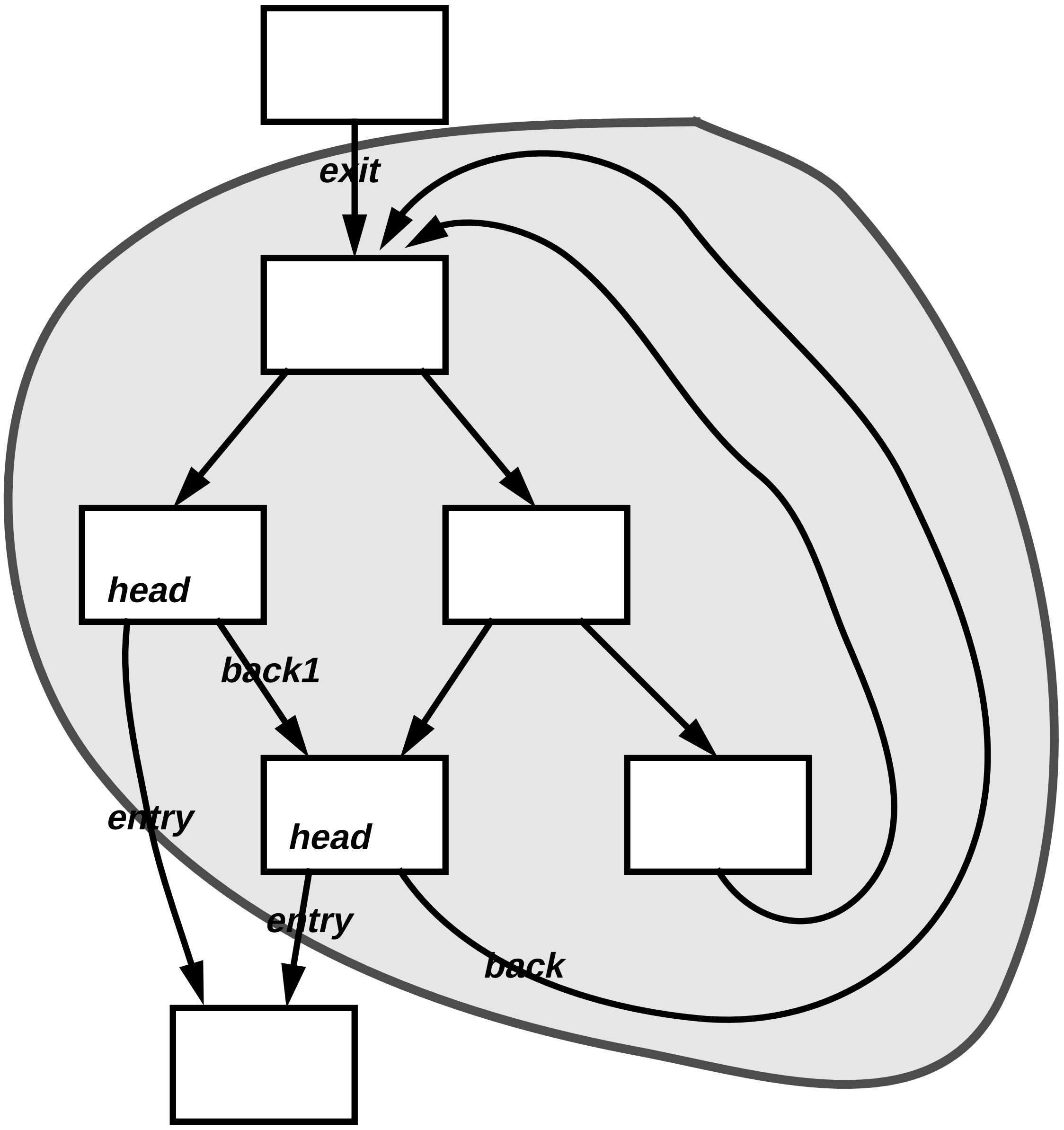
Name: ***Solutions***

100 points total.

1) a) [5 pts] For the following control flow graph, circle the loop(s) and label all the occurrences of each loop's loop heads, loop entries, loop exits, and backward branches (indicating to which loop(s) each label relates), for the purposes of a **reverse** iterative data flow problem such as live variables.



b) [3 pts] Consider the following flow function for a simple assignment statement, implemented as part of an analysis computing the set of live variables at each statement:

$$F_{x := y \text{ op } z}(in) = in \cup \{y,z\} - \{x\}$$

What is wrong with this function? Give an example where this flow function computes the wrong result. What is the correct specification?

The function throws out the definition after adding in the uses. This is backwards. This will fail for e.g. increment statements:

```
x := x + 1;
```

Before this increment, x is live, but the flow function will compute that it is dead.

The correct flow function is:

$$F_{x := y \text{ op } z}(in) = in - \{x\} \cup \{y,z\}$$

2) a) [3 pts] What is the primary property of a program in SSA form?

Each use is reached by a single definition.

b) [5 pts] Aside from presenting SSA form and value-graph-based calculations of available expressions, what was the novel contribution of the work in Alpern, Wegman, and Zadeck's paper "Detecting Equality of Variables in Programs"?

They showed how to incorporate structured control flow dependencies into the value graph framework. For instance, after

```
if i < j then k := 1 else k := 2;
if i < j then l := 1 else l := 2;
```

the analysis would learn that k and l had the same value.

c) [10 pts] Outline an iterative data flow analysis algorithm for converting a CFG into SSA form. Describe the information propagated through the graph, the direction of propagation, the initial conditions, the merge functions, and the interesting flow functions. You may assume that modifications to the graph only take effect when the iteration reaches a fixpoint (as in the Vortex infrastructure). Be sure to handle the case when the l.h.s. of a ϕ function is one of its own arguments. You may assume that all variables are defined before use.

First, make a pre-pass over the graph, giving each l.h.s. variable a unique subscript, and giving each merge node (or basic block) a unique number different from any variable's subscript.

Then, propagate sets of ($var \rightarrow var_{subscript}$) bindings (with at most one binding for each var) forward through the graph, starting with the empty set.

At each statement of the form $x_{subscript}$:= y op z, replace y with $y_{subscript\text{-}y}$ in the table (and similarly for all other operands on the r.h.s.). Add ($x \rightarrow x_{subscript}$) to the table, replacing any earlier binding for x if there was one.

The merge function is tricky. Consider each variable var defined in the incoming sets. (It will be defined in all or none, assuming that defs always precede uses.) In each incoming set i, var maps to some $var_{subscript\text{-}i}$. If all the $var_{subscript\text{-}i}$ are the same, then include ($var \rightarrow var_{subscript\text{-}i}$) in the merged set. If there are any differences, then create a statement $var_{merge\text{-}id} := \phi(var_{subscript\text{-}1}, ..., var_{subscript\text{-}N})$, where merge-id is the number of the merge node, and add the statement right after the merge. Using the merge node's id to generate the subscript of the target of the ϕ function assures that the merge function is idempotent, and hence that iteration will reach a fixed point.

3) Lattices are often used to describe the information computed at each program point during an data flow analysis. This enables formal proofs about the correctness and termination of the analysis.

a) [3 pts] What properties are required of a partial order for it to be a lattice?

GLB's (meets) and LUB's (joins) of all pairs of elements must exist. (Note that top and bottom elements are not required, to allow infinite-height lattices like the integers ordered by $\leq$.)

b) [6 pts] What is the lattice for the reaching definitions problem? (A lattice is defined by a set of elements and a $\leq$ ordering operation over those elements.) What are the top and bottom elements of the lattice, if they exist? What is the meet function? (Use the standard convention for choosing the ordering function so that the lattice meet operation corresponds to the merge function.)

Lattice domain: powerset of set of all statements.

$a \leq b \equiv a \supseteq b$

Top: empty set

Bottom: set of all statements

Meet: union

c) [3 pts] For the computed approximate solution to be sound w.r.t. the meet-over-all-paths ideal solution, the flow functions must be monotonic and idempotent. For the lattice of the set of integers ordered by ≤, give an example of a non-monotonic function and a non-idempotent function.

not monotonic: F(in) = - in

not idempotent: F(in) = counter++;

d) [3 pts] What advantage is gained if the flow functions are distributive, not just monotonic?

The approximate solution is the same as the ideal meet-over-all-paths solution; there is no loss of precision by not considering each path through the CFG separately.

e) [3 pts] What properties are required of the lattice and/or the flow functions for the analysis to terminate? For each required property, give an example where it is not preserved.

The flow functions must be monotonic. Also, the lattice must be of finite height. Ideally, it should be short, too. (Or there must be some sort of “widening” lattice operator applied at loop heads where only a finite number of widenings can occur.)

Example: any infinite height lattice, e.g. the integers ordered by ≤.

f) [5 pts] 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. 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.

4) a) [3 pts] How do interprocedural summaries improve on the supergraph-based approach (i.e. linked CFG’s) to interprocedural analysis?

Summaries are much smaller to store.

They can be computed in isolation much more easily, leading to easier separate compilation and faster analysis.

b) [3 pts] How does procedure specialization improve on interprocedural analysis?

It allows multiple separate versions of a callee to be analyzed, each for a subset of “similar” callers.

c) [5 pts] Dean & Chambers's paper on making better inlining decisions described an inlining database that recorded the code space costs and execution speed benefits of previous attempts at inlining a routine, to avoid future inlining if it was not profitable. How could this mechanism be adapted to support effective and selective procedure specialization, in addition to its existing selective inlining support?

Those calls that have high benefit but high cost could be selected for specialization, since inlining is ruled out by the high code space cost. The part of each entry describing the statically-available information at the call site might naturally guide the specialization process.

5) Binding time analysis is exploited by off-line partial evaluators to annotate a procedure's expressions with either `static` or `dynamic`, given an initial annotation of the procedure's formal arguments.

a) [10 pts] Devise an iterative data flow analysis algorithm for performing binding time analysis. Describe in lattice-theoretic terms the information computed at each program point (including the top and bottom elements of the lattice and the lattice meet function), the direction of analysis, the initial lattice element, and the flow function for the ubiquitous `x := y op z` statement.

First, convert to SSA form.

For a single variable, lattice is a simple two-point lattice, with top = static and bottom = dynamic, dynamic ≤ static, meet(t1, t2) = min(t1, t2) (i.e. dynamic if either t1 or t2 or both is dynamic, static otherwise).

For a group of variables, maintain a set of 2-point lattices, one per defined variable. I.e., lattice is the powerset of set of (var → bt) bindings, each var bound at most once, bt ∈ {static,dynamic}; top = all vars are static, bottom = all vars are dynamic; meet = pointwise for each variable; d1 ≤ d2 iff individual vars are ≤ pointwise.

Analysis is forward, starting with domain where all variables are bound to static except for those formal arguments annotated as dynamic.

The flow function for `x := y op z` is

```
F(in) =
    if in(y) = static and in(z) = static
    then in[x→static]
    else in[x→dynamic]
```

The flow function for x_n := $\phi(x_1, ..., x_k)$ is

```
F(in) = in[xn→dynamic]
```

i.e., the result of merging multiple variables, even if they're static, is dynamic.

b) [5 pts] On-line partial evaluators do not use a prepass BTA to annotate what expressions will be computed at partial-evaluation-time. Instead, they directly evaluate expressions whose arguments are constants as they perform partial evaluation. In some cases, on-line PE's will identify more static computations than does an off-line BTA analysis, and consequently on-line PE's can eliminate more computations than an equivalent off-line PE. Give an example illustrating how this can happen. Hint: think about a conditional test that is labeled static by BTA.

If a test is static, but one of the branches is static while the other is dynamic, a BTA analysis must be conservative and assume that either branch might be selected at specialization time, leading to a dynamic result of the if expression. Alternatively, an on-line PE will evaluate the test at PE time, selecting only one of the branches to be evaluated, leading to a static result of the if if the static branch is selected.

E.g. assuming that x is static and specialized to the value zero, after

```
if x = 0 then y := 0 else y := input() end;
```

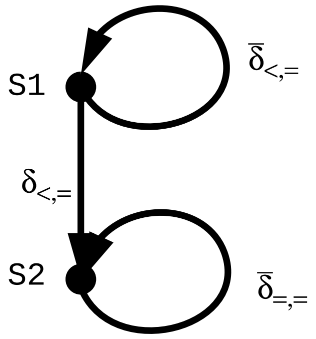
an off-line PE must conservatively label y as dynamic (it cannot predict the outcome of the `x = 0` test), while an on-line PE, running with the actual bindings for x etc., will determine that only the `then` branch will be executed and hence that y is static (with value 0). Downstream uses of y will be optimized in the on-line PE but not the off-line PE.

6) a) [5 pts] How does a compiler test when loop interchanging is legal?

When the dependences among the statements in the loop remain lexicographically non-negative, after reordering dependence subscripts according to the reordered loops.

b) [7 pts] Compute the data dependencies between the statements in the following loop nest, using direction vectors.

```
    for i = 1, N
      for j = 1, N
S1:         A[i,j] := A[i+2,j] * C[1,j];
S2:         B[i,j+1] := B[i,j+1] + A[i-1,j] + K[j];
      end
    end
```



c) [3 pts] What do the data dependencies inform the compiler about the possible parallelization of the loops?

The inner loop has no loop carried dependencies, so it can be parallelized. The outer loop cannot be parallelized very well.

d) [5 pts] What transformations can be applied legally to this loop to make it parallelize better? Why is the transformed loop better?

Loop interchanging the i and j loops is legal, and profitable because the outermost loop is the parallel one, and each iteration presumably has a lot of work to do.

e) [5 pts] Specifically, how could blocking improve the performance of this loop on a uniprocessor?

The C[1,j] and K[j] references read the same row of C and all of K over and over again for each i iteration. If the j loop is blocked (which is legal because i and j can be interchanged), then the same subset of the C columns and the same elements of K could be read over and over again while they were still in the cache. I.e.:

```
for j’ = 1, N, ChunkSize
  for i = 1, N
    for j = j’, j’+ChunkSize-1
S1:     A[i,j] := A[i+2,j] * C[1,j];
S2:     B[i,j+1] := B[i,j+1] + A[i-1,j] + K[j];
    end
  end
end
```

Note that just doing loop interchange would also accomplish the same effect, more easily, for this particular example.