Midterm II Solution

- Please read all instructions (including these) carefully.
- There are 5 questions on the exam, all with multiple parts. You have 80 minutes to work on the exam.
- The exam is closed book, but you may refer to your four sheets of prepared notes.
- Please write your answers in the space provided on the exam, and clearly mark your solutions. You may use the backs of the exam pages as scratch paper. Please do not use any additional scratch paper.
- Solutions will be graded on correctness and clarity. Each problem has a relatively simple and straightforward solution. You may get as few as 0 points for a question if your solution is far more complicated than necessary. Partial solutions will be graded for partial credit.

NAME: _____

SID or SS#: _____

Class account: _____

Problem	Max points	Points
1	15	
2	25	
3	25	
4	20	
5	15	
TOTAL	100	

1. **Parameter Passing** (15 points)

Consider the following program, written in a C-like syntax:

```
f(x) { return g(x = x * 2); }
g(x) { let y = 1 in { h(y); return x + y + x; }}
h(x) { x = x + 5; return 0; }
main() { print(f(7)); }
```

Note that each function is called in one place. Assume that each function call can be used with a different parameter passing mechanism: by value, by reference, or by name.

(a) For each of `f`, `g`, and `h`, state what calling convention should be used for the program to print 29.

`This problem turns out to have many correct answers. The simplest solution for this part is that all three are call-by-value.`

(b) For each of `f`, `g`, and `h`, state what calling convention should be used for the program to print 34.

`Any solution where h is call-by-reference and g is not call-by-name is correct.`

(c) For each of `f`, `g`, and `h`, state what calling convention should be used for the program to print 43.

`Any solution where h is not call-by-reference and g is call-by-name is correct.`

2. **Type Checking and Semantics** (25 points)

In this problem we consider adding a `for` loop to Cool. The syntax is:

```
for ( id:T <- e_init; e_test; e_update) e_body rof
```

A `for` expression first initializes the identifier `id` to the value of the expression `e_init`. It then evaluates the expression `e_test`. If the value of `e_test` is `false`, the loop terminates. If the value of `e_test` is `true`, then `e_body` is evaluated and then `e_update` is evaluated. Loop execution then continues by evaluating `e_test` again, and so on.

The type of a `for` expression is `Object`. The identifier `id` is visible in `e_test`, `e_update`, and `e_body`.

(a) Give a type checking rule for `for` expressions.

```
O, M, C |- e_init:T1
T1 <= T'
O[T'/id], M, C |- e_test:Bool
O[T'/id], M, C |- e_update:T2
O[T'/id], M, C |- e_body:T3
------------------------------
O, M, C |- for(id:T<-e_init; e_test; e_update) e_body rof: Object
```

(b) `for` expressions are equivalent to some combination of other Cool constructs. Show how to translate a `for` into Cool without `for`.

```
(let id:T<-e_init in
while e_test loop {
e_body;
e_update;
} pool
)
```

3. **Global Analysis** (25 points)

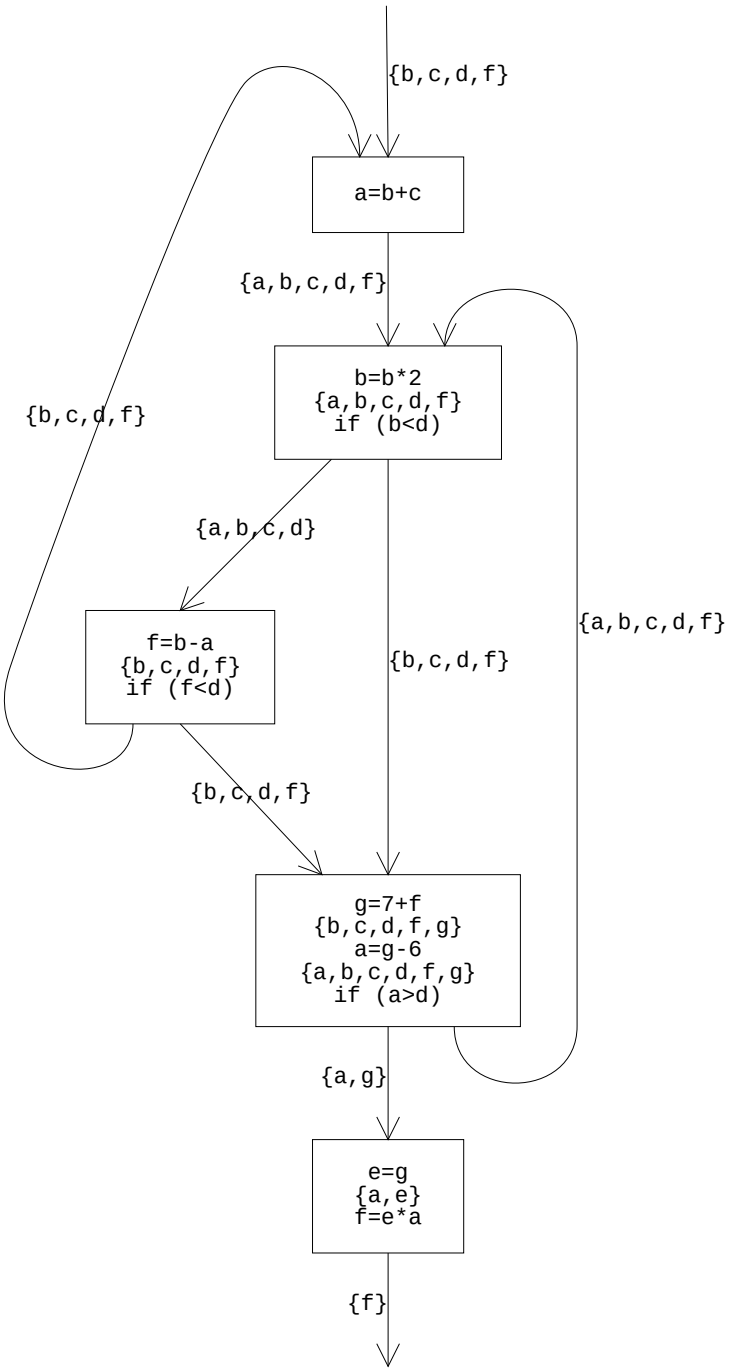
Consider the following three-address code fragment:

```
1.   L1:    a = b + c
2.   L2:    b = b * 2
3.           if (b < d) goto L3
4.           f = b - a
5.           if (f < d) goto L1
6.   L3:    g = 7 + f
7.           a = g - 6
8.           if (a > d) goto L2
9.           e = g
10.          f = e * a
```

(a) Identify the basic blocks in the above fragment. Assign each basic block a number, and show the set of lines contained within each block. For example, if you believed that the two multiplications and the two additions formed one basic block, you might write "basic block #1: { 1, 2, 6, 10 }". Please refer to the instruction numbers given above in your answer.

i. { 1 }
ii. { 2, 3 }
iii. { 4, 5 }
iv. { 6, 7, 8 }
v. { 9, 10 }

(b) Draw the control flow graph for the above fragment. Between each pair of instructions within a basic block, write the set of live variables at that program point. On each edge between basic blocks as well as at each entry and exit point of the control flow graph, write the set of live variables at that point. Assume that only `f` is live on exit from the entire code fragment.



4. **Register Allocation** (20 points)

Each part of this question describes a set of graphs. State whether the graph coloring heuristic used for register allocation will successfully color every graph in the set using k or fewer colors. If you believe the coloring heuristic can color the graphs, given an explanation why. If you believe the heuristic cannot color the graphs, give an example graph in the set that can't be colored.

(a) Graphs where every node has fewer than k neighbors.

As mentioned during the exam, the question is whether the heuristc can succeed without spilling. The graph coloring heuristic has two phases: in phase one, nodes are deleted from the graph and placed on a stack; in phase two, the nodes are removed from the stack and colored one at a time. If phase one succeeds in removing all of the nodes from the graph, then phase two always succeeds. Thus, the problem reduces to arguing whether or not phase one can remove all nodes from the graph.

For the first part, yes, the heuristic can color all such graphs. The graph coloring heuristic works by repeatedly selecting nodes with < k neighbors and removing them from the graph. Because removing a node cannot increase the number of neighbors of any other node, the heuristic will succeed in removing all nodes from the graph.

(b) Graphs where every node has at least k neighbors.

No. Consider the complete graph of $k + 1$ nodes (a complete graph has every possible edge). Every node has k neighbors, so the heuristic cannot remove any edges.

A common mistake was failing to give an example as instructed. Some people claimed that the optimistic coloring variant of the heuristic would work, but optimistic coloring does not work for a complete graph of size $k+1$.

(c) Graphs where only one node has k or more neighbors.

Yes. First remove all of the nodes except the one with k or more neighbors. At this point there is only one node left, so it must have zero neighbors and can be removed from the graph as well.

Some people correctly said the graph could be colored using a different procedure, but that was not an answer to the question asked.

(d) Graphs where at most $k-1$ nodes have k or more neighbors.

Yes. First remove all of the nodes that have $< k$ neighbors in the original graph. Now there are $k - 1$ nodes left, so each can have at most $k - 2$

neighors; i.e., each has less than k neighbors. Thus, the heuristic can remove all remaining nodes.

Again, some people correctly said the graph could be colored using a different procedure, but that was an answer to a different question.

5. **Code Generation** (15 points)

(a) Assume we disallow method override in Cool. How can we simplify code generation for method calls?

If method overriding is disallowed, dispatch pointers are no longer necessary. We can statically determine the location of the method to execute. In terms of code generation, dispatches consist of jump-and-link's to statically determined locations, rather than table lookups followed by register jumps.
Partial credit was awarded for stating that the static dispatch construct (@) can be eliminated, or that methods can be inlined.

(b) Succinctly, under what circumstances can we allocate an object on the stack rather than allocating it in the heap.

Objects that may remain live after the execution of a method must be allocated on the heap. Otherwise, they can be stack allocated.
Common mistakes-- Claiming that objects that are only accessed in one method are the only objects that can be stack allocated.

(c) As implemented by `coolc`, the header of a Cool object consists of three words: the class tag, a pointer to a dispatch table, and the size of the object. Outline a scheme for reducing the size of the object header to one word, without limiting the set of Cool programs we can compile. (For example will not accept a solution that allocates 4 bits to the class tag and assumes there are only 16 classes in a program!)

First, we note that in Cool, every instance of a class has the same size, dispatch pointer, and class tag. Reducing the amount of repeated information is our goal. The easiest way to achieve this goal is to augment dispatch tables with additional information. The tables become class descriptors; they now contain other fields like the object size and class tag in addition to the information normally contained in the dispatch table. In fact, since dispatch pointers are unique to each class, it is possible to use the dispatch pointer itself as a class tag. However, this complicates code generation for constructs like case. One complication with this approach is that it is not quite true that every instance of a Cool class has the same size. Cool Strings are the single exception. If strings are treated as special cases, this approach will still work.
Other strategies can be used, some of which might significantly slow down the execution of Cool programs (for instance, some proposed solutions add

```
another level of indirection to method dispatch).
Common mistakes:  Adding a pointer to a new record that contains the class
tag, object size, and dispatch table, without noting that this record is
shared among all instances of the same class.  This actually increases the
size of the Cool object header by one word.
```