Compiler Fall 2011
PRACTICE Final Exam

This is a full length practice final exam. If you want to take it at exam pace, give yourself 90 minutes to take the entire test. Just like the real exam, each question has a point value. There are 90 points in the exam, so that you can pace yourself to average 1 point per minute (some parts will be faster, some slower).

Questions:

1. Type Inference (10 points)
2. Frame Layout (15 points)
3. Translation to IR (10 points)
4. Register Allocation (15 points)
5. Dataflow Analysis/Optimization (15 points)
6. Domination (10 points)
7. Garbage Collection (15 points)

This is the solution set to the practice exam. The solutions appear in blue boxes.

Question 1: Types [10 pts]

1. Infer the type of the following ML function (show your work):

```
fun f (w,x) = case x of
                [] => []
              | y::z => w(y)::f(w,z)
```

Answer:
First, assign type variables where appropriate:
fun f (w:'a,x:'b):'c = case x of
[] => []
| (y:'d)::(z:'e) => w(y)::f(w,z)
Then start unifying:

Must Match		So unify		Resulting in
x	[]	'b	'f list	'b $\mapsto$ 'f list
(y,z)	cons's arg	('d * 'e)	('h * 'h list)	'd $\mapsto$ 'h 'e $\mapsto$ 'h list
x	cons's rslt	'f list	'h list	'f $\mapsto$ 'h
w	fn taking y	'a	'h $\rightarrow$ 'i	'a $\mapsto$ 'h $\rightarrow$ 'i
(w,z)	f's arg	('h $\rightarrow$ 'i) * 'h list	('h $\rightarrow$ 'i) * 'h list	
(w(x),f(w,z))	cons's arg	'i * 'c	'j * 'j list	'i $\mapsto$ 'j 'c $\mapsto$ 'j list
1^{st} case rslt	2^{nd} case rslt	'k list	'j list	'k $\mapsto$ 'j
fn body	fn rslt ty	'j list	'j list	

Replace types:
fun f (w:'h $\rightarrow$ 'j ,x:'h list):'j list = case x of
[] => []
| (y:'h)::(z:'h list) => w(y)::f(w,z)

2. What goes wrong if you attempt type inference on `fun f(x)= f`?

Answer:
When you attempt to run type-inference on fun f(x)= f, you try to unify 'a with ('a $\rightarrow$ 'b), which fails the *occurs* check.

Question 2: Frame Layout [15 pts]

- Explain the concept of a static link. Why is it needed?

  Answer:
  The static link is the frame pointer of the statically enclosing function's stack frame. This frame may dynamically be one or many frames out. The static link is required to allow access to variables declared in outer-functions, which reside in that function's frame.

- Explain the difference between the stack pointer and the frame pointer. One of them can be omitted in certain circumstances. Identify which one, and explain when it is not needed, and what benefits are obtained from omitting it.

  Answer:
  The FP points at the "start" of the current functions stack frame. Variables are always a fixed offset from the FP. The SP, on the other hand points at the end of the stack. In the prescense of dynamic stack allocation (alloca, variable sized arrays, etc), the offset of a variable from the SP may change depending on the size of the dynamically allocated structures. The FP could be omitted when no dynamic allocation is performed. Omitting the FP saves a few instructions at function entry and exit, improving the speed of the program.

- Given the following Tiger code:

```
let function odd(a: int) : int =
    if (x == 0)
      then 0
      else if (x == 1)
        then 1
        else if(x < 0)
          then odd(-x)
          else odd(x-2)
    function f (x: int) : int =
     let var b := x + 2
        function g (y: int) : int =
           if odd(y) then  b + g(y-1)
                       else  h(y / 2)
        function h(y : int) : int =
           if y <= 42 then 1
                      else  f(y - 7)
     in
       g(x*2-3)
     end
  in
    f(42)
  end
```

For each of the following function calls, state what is passed as the static link. Assume that the SL is stored in the frame slot pointed to by the frame pointer for each frame.

1. `f` calls `g` FP
2. `g` calls `h` Its own SL (or MEM(FP))
3. `g` calls itself recursively. Its own SL
4. `h` calls `f` MEM(MEM(FP))
5. `g` calls `odd` MEM(MEM(FP))

Question 3: Translation to IR [10 pts]

Translate the following bits of Tiger into IR (you can either draw the IR tree or write it out as SML constructors). For each case you should assume the following variable locations (all InFrame variables are in your own frame. You can refer to the frame pointer as simply FP). Note: you do not need to include bounds checks for array accesses:

Variable	Location
x	InReg t1
y	InReg t2
z	InFrame -4
a	InReg t3

- y := 1 + x

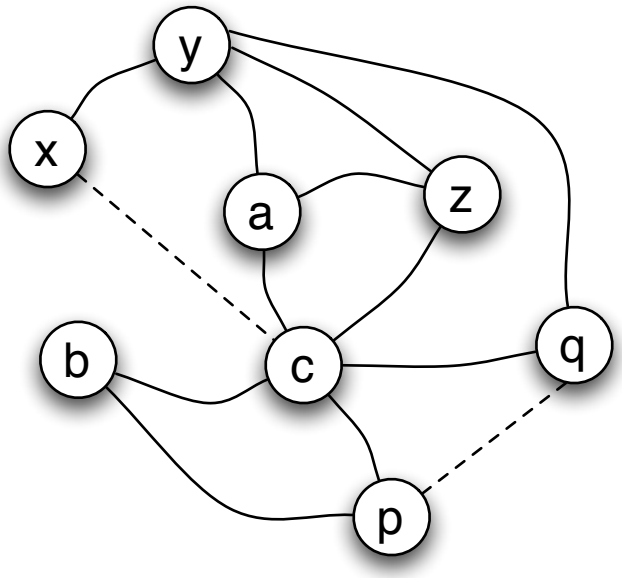
  Answer:
  MOVE(TEMP t2 , BINOP(PLUS,CONST 1 ,TEMP t1))

- a[y] := f(z)

  Answer:
  MOVE(MEM(BINOP(PLUS,
  TEMP t3,
  BINOP(TIMES, TEMP t2, CONST 4))),
  CALL(LABEL("f"),[MEM(BINOP(PLUS,FP,CONST -4))]))

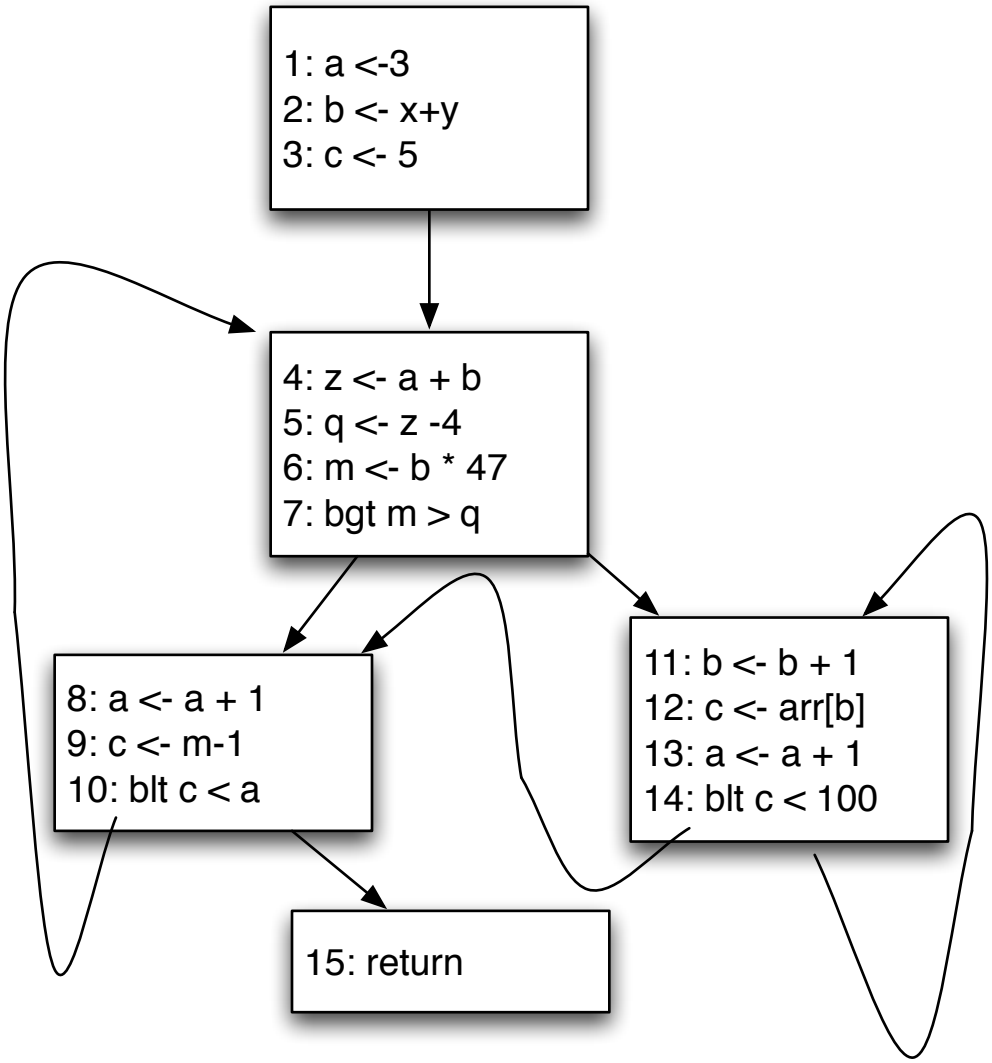
Question 4: Register Allocation [15 pts]

Perform register allocation for a 3 register machine on the following interference graph (dashed lines indicate move relationships, solid lines indicate interference). You should coallesce moves whenever it is safe to do so according to either heuristic we learned. Show your work (you do not need to redraw the graph for each step, but you should list the order in which you simplify/coallesce/freeze nodes)



Answer:
Simplify b
Coallesce p and q
Simplify pq
Freeze x and c
Simplify x, y, a, z, c
Registers: c: r1, z: r2, a: r3, y: r1, x: r2, pq: r2, b r3

Question 5: Dataflow Analysis/Optimization [15 pts]



1. Using the above program fragment, which definitions reach the following uses (you can identify them by their instruction number):
   - The use of `a` in instruction 4. 1,8
   - The use of `a` in instruction 8. 1,8,13

- The use of `b` in instruction 6. 2,11

2. For the same program fragment, indicate whether each of the following expressions is "very busy" (write Y or N) after each block (after the last instruction in that block, numbered down the left side of the table):

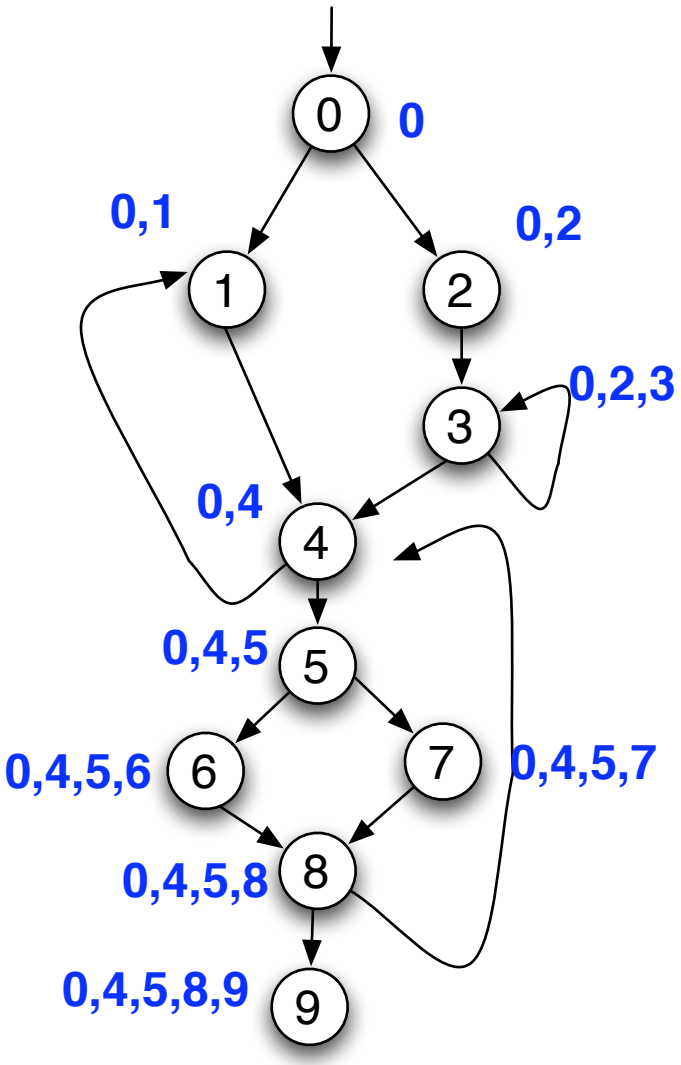
	a + 1	m - 1	a + b	b * 47	x + y	b+1	arr[b]
3	Y	N	Y	Y	N	N	N
7	Y	Y	N	N	N	N	N
10	N	N	N	N	N	N	N
14	Y	Y	N	N	N	N	N
15	N	N	N	N	N	N	N

3. For each of the following, indicate the appropriate data flow analysis (you can just write one of RD, LV, VBE, or AE on each line):

   - Common sub-expression elimination AE
   - Forward flow/union RD
   - Def/Use Web Formation RD
   - Backwards flow/intersection VBE

Question 6: Domination [10 pts]

Consider the following control flow graph:



- Label each node in the graph with its dominator set. Done in place above
- Identify the loops in the graph by their backedges.

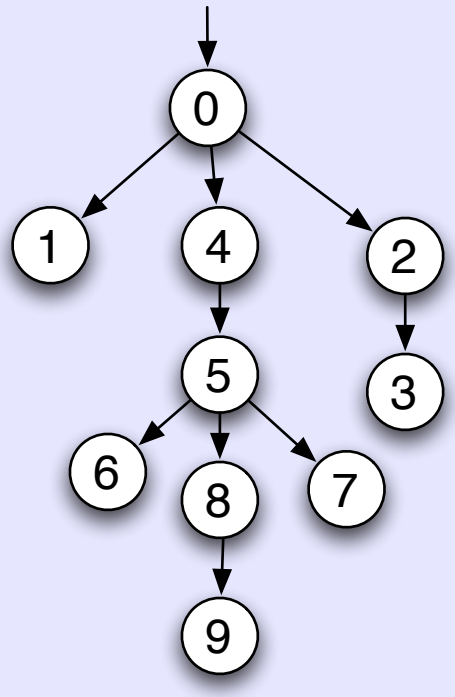
Answer:
$3 \rightarrow 3$ is one loop. The other is $8 \rightarrow 4$. Note that $4 \rightarrow 1$ is NOT a loop (see next question).

- This control flow graph has a part that looks like a loop to a naive definition, but is not a loop for the definition required for many optimizations. Identify this false loop and briefly explain why it is not a loop.

Answer:
4 makes a backwards edge to 1, which meets a naive definition of a loop (e.g., a cycle). However, 1 does not dominate 4 (0→2 →3 → 4 is a possible path), so this is not a true loop.

- Draw the immediate dominator tree for this graph.

Answer:



Question 7: Garbage Collection [15 pts]

- List three advantages of garbage collection. You may list advantages particular to one specific GC algorithm, but please specify which algorithm if you do so.

  Answer:
  Any 3 of these (plus possibly others): fast allocation (S&C), improved locality (S&C), avoid double frees, avoid free-then-use errors, avoid memory leaks, reduce development/debugging time/cost, ...

- Briefly describe the performance/space tradeoffs between the three algorithms we discussed: reference counting, mark and sweep, and stop and copy.

  Answer:
  Reference counting has very high performance overhead: it requires stores every time pointers are manipulted, typically resulting in 20–30word per object to hold the count. Mark and sweep requires a DFS (time proportional to the live objects) and examination of all objects (time proportional to the total heap size). When the heap size is large relative to the number of live objects, this amortized cost is acceptable. The direct space overhead is one word per object (to hold the marking state). Stop and copy requires time proportional to the number of live objects. It has large space overhead—half the heap must be unused to copy into. There may be an additional overhead of one word per object to hold forwarding pointers, depending on the object layout.

- The garbage collector needs to know which fields in an object are pointers, and which are not. Briefly describe two ways it might do this.

  Answer:
  Any two of these three: The compiler could pass down type information to tell it. It could be conservative: anything that looks like a pointer might be (but then no moving objects). It could use tagging: steal one bit from each word to indicate "pointer or not".