Name:

Midterm 2 — Sample solution

CMSC 430

Introduction to Compilers

Fall 2012

November 28, 2012

Instructions

This exam contains 9 pages, including this one. Make sure you have all the pages. Write your name on the top of this page before starting the exam.

Write your answers on the exam sheets. If you finish at least 15 minutes early, bring your exam to the front when you are finished; otherwise, wait until the end of the exam to turn it in. Please be as quiet as possible.

If you have a question, raise your hand. If you feel an exam question assumes something that is not written, write it down on your exam sheet. Barring some unforeseen error on the exam, however, you shouldn't need to do this at all, so be careful when making assumptions.

Question	Score	Max
1		20
2		14
3		30
4		12
5		24
Total		100

Question 1. Short Answer (20 points).

a. (6 points) Briefly explain what a *basic block* is.

> **Answer:** A basic block is a sequence of statements such that (a) there are no jumps from the basic block except after its last statement and (b) there are no jumps into the basic block except to its initial statement.

b. (7 points) Briefly explain what *state transformation* is in dynamic software updating and why it may be needed.

> **Answer:** State transformation is the process of modifying the application state to be compatible with a new program version. State transformation is needed because often code updates code with data representation and type changes.

c. (7 points) Briefly explain what a *time travel debugger* is.

Answer: Typical interactive debuggers allow the developer to set breakpoints and then, whenever execution is stopped, inspect the program state at that point. A time-travel debugger additionally allows the developer to inspect *past* program states. (A canonical operation in a time-travel debugger is "step backwards," which runs execution one step in reverse.)

Question 2. Subtyping (14 points). Suppose that *int* is a subtype of *float*. To save writing, let's write i for *int* and f for *float*.

a. (7 points) Write down every type t such that $t \leq (i \to i \to f)$, following standard subtyping rules. *Hint: It may be easiest to write down everything that could possibly be a subtype and then cross out the ones that aren't subtypes.*

Answer:

Recall that $i \to i \to f$ is parsed as $i \to (i \to f)$. So, if you work through the rule for function subtyping, you will see that every combination of types with the right shape is possible:

$i \to i \to i$
$i \to i \to f$
$i \to f \to i$
$i \to f \to f$
$f \to i \to i$
$f \to i \to f$
$f \to f \to i$
$f \to f \to f$

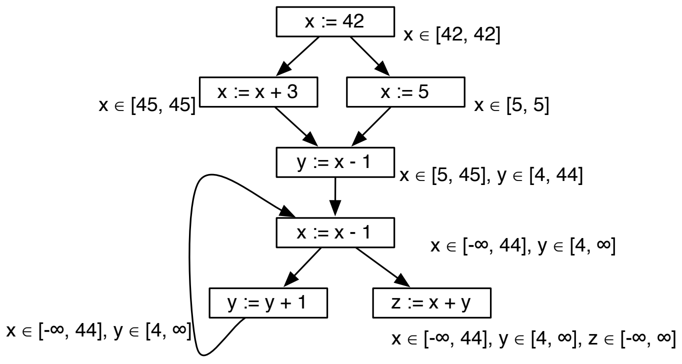
b. (7 points) In class, we argued it would be unsound to allow *int ref* $\leq$ *float ref*. Demonstrate the issue by writing down, in OCaml notation, a program that would type check under this subtyping rule but that would "go wrong" with a type error at run-time. Explain your answer very briefly.

Answer:

```
let (x:int ref) = ref 0 in
let (y:float ref) = x in
  y := 3.0;
  x + 1  (* uh oh! integer arithmetic on a float *)
```

Note: several people wrote answers showing how an *int* could reach an argument of `+.` (floating point addition). However, the premise of this whole question is that *int* is a subtype of *float*, and it's unreasonable to assume that would be the case without also adjusting `+.` to handle integers.

Question 3. Interval Analysis (30 points). In this question, we will develop an *interval analysis*, which, for each variable x at each program point, determines a closed interval $[a, b]$ such that the run-time value of x is guaranteed to be in the interval $[a, b]$. We also allow a and b to be $-\infty$ and ∞, respectively, in case we cannot bound the interval on one or both sides. Use $\emptyset$ for the empty inverval. For example, here is a CFG annotated with the intervals determined *after* each statement (empty intervals omitted):



Note that we have left out any conditional tests; as is usual in dataflow analysis, your analysis should always assume all branches could be taken.

a. (5 points) Should the analysis be forward or backward?

Answer: Forward.

b. (5 points) What should the initial facts be at the entry or exit of the program? (You can explain in words.)

Answer: Every variable should be mapped to the empty interval.

c. (5 points) What should $\top$ be in the lattice? (You can explain in words.)

Answer: Every variable should be mapped to the empty interval.

Many students answered that $[-\infty, \infty]$ should be $\top$, but that doesn't work for two reasons. First, mathematically, it shouldb be that $x \sqcap \top = x$. But with the $\sqcap$ operation defined below, $x \sqcap \top$ would be $\top$. Second, dataflow analysis should start with the most optimistic assumption ($\top$) and monotonically decrease to the actual answer. In this case, it's most optimistic to assume at a program point that each variable x maps to the empty interval, and then as we discover more information about it, *expand* the interval to contain its actual value.

d. (5 points) Suppose that on one incoming edge to a join point, $x \in [a, b]$, and on another incoming edge to the same point, $x \in [c, d]$. What should $(x \in [a, b]) \sqcap (x \in [c, d])$ be defined as?

Answer: $x \in [\min(a, c), \max(b, d)]$.

e. (5 points) Suppose $x \in [a, b]$ just prior to each of the following statements. Write down the new dataflow fact $x \in [c, d]$ after the statements:

i. `x := 42`

Answer: $x \in [42, 42]$

ii. `x := x + 1`

Answer: $x \in [a + 1, b + 1]$

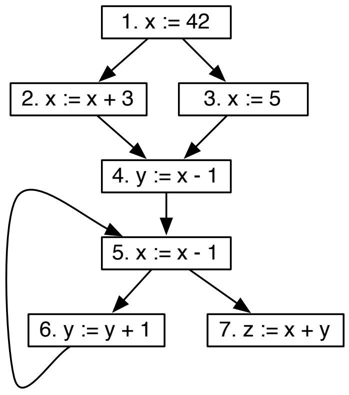
iii. `x := 4 - x`

Answer: $x \in [4 - b, 4 - a]$

f. (5 points) If we implement the usual dataflow analysis algorithm, is the algorithm guaranteed to terminate? Why or why not?

Answer: No, it's not guaranteed to terminate because this lattice does not have finite height.

Question 4. Data flow analysis (12 points). Here is the control-flow graph from the last problem again, this time with numbers for each statement:



a. (6 points) Write down the sets of live variables at the *beginning* of each statement. Write ∅ for the empty set, if necessary.

Stmt	Live variables at beginning of stmt
1	∅
2	x
3	∅
4	x
5	x,y
6	x,y
7	x,y

b. (6 points) Write down the sets of reaching definitions at the *end* of each statement. Write ∅ for the empty set, if necessary.

Stmt	Reaching definitions at end of stmt
1	1
2	2
3	3
4	2,3,4
5	4,5,6
6	5,6
7	4,5,6,7

Question 5. Code generation and register allocation (24 points). Below is a snippet of `08-codegen-2.ml` from class, showing the input expression language, the "bytecode" instruction language, and compilation. We've made two small changes: We renamed `‘L_Register` to `‘L_Reg` to save some writing; and we removed reads and writes through pointers and identifiers.

```
type expr =
  | EInt of int
  | EAdd of expr * expr
  | ESub of expr * expr
  | EMul of expr * expr
  | EIfZero of expr * expr * expr

type reg = [ ‘L_Reg of int ]
type src = [ ‘L_Int of int ]

type instr =
  | ILoad of reg * src        (* dst, src *)
  | IAdd of reg * reg * reg   (* dst, src1, src2 *)
  | IMul of reg * reg * reg   (* dst, src1, src2 *)
  | IIfZero of reg * int      (* guard, target *)
  | IJmp of int               (* target *)
  | IMov of reg * reg         (* dst, src *)
```

```
let rec comp_expr (st:(string*int) list) = function
 | EInt n →
     let r = next_reg () in
       (r, [ILoad (‘L_Reg r, ‘L_Int n)])
 | EIfZero (e1, e2, e3) →
     let (r1, p1) = comp_expr st e1 in
     let (r2, p2) = comp_expr st e2 in
     let (r3, p3) = comp_expr st e3 in
     let r = next_reg () in
       (r, p1 @
          [IIfZero (‘L_Reg r1, (2+(List.length p3)))] @
          p3 @
          [IMov (‘L_Reg r, ‘L_Reg r3);
           IJmp (1+(List.length p2))] @
          p2 @
          [IMov (‘L_Reg r, ‘L_Reg r2)]
       )
```

a. (14 points) Suppose we extend the source language with *short-circuiting disjunction* `EOr(e1, e2)` that does the following: First, it evaluates expression `e1` to produce a value v. If v is non-zero, then v is returned as the value of the disjunction. Otherwise, expression `e2` is evaluated and its value is returned. For example, `EOr (EInt 1, EInt 2)` evaluates to 1, and `EOr (EInt 0, EInt 2)` evaluates to 2. (Notice that `e2` is not evaluated if `e1` is non-zero.)

Write a case of comp_expr that compiles `EOr`.

```
let rec comp_expr (st:(string*int) list) = function
  ...
 | EOr (e1, e2) →
```

Answer:

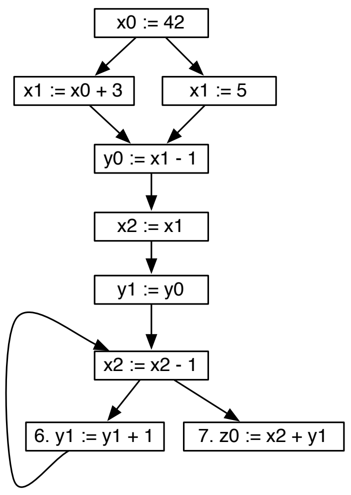
```
let (r1, p1) = comp_expr st e1 in
let (r2, p2) = comp_expr st e2 in
let r = next_reg () in
  (r, p1 @
     [IIfZero (‘L_Reg r1, 2);
      IMov (‘L_Reg r, ‘L_Reg r1);
      IJmp (1 + (List.length p2))] @
     p2 @
     [IMov (‘L_Reg r, ‘L_Reg r2)])
```

Several students also discovered a much simpler answer:

```
comp_expr (EIfZero(e1, e2, e1))
```

(Note that this is not an ideal solution, though, because it may evaluate e1 twice, which is probably something we care about if we're implementing short-circuiting disjunction.)

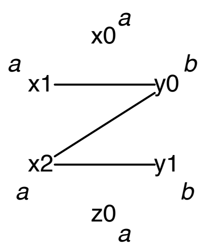
b. (10 points) Finally, consider the following slight modification of the CFG from the earlier problems:



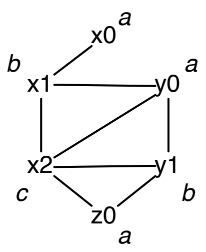
Draw the interference graph for the variables referred to in the above CFG. After you have drawn the graph, "color" it by labeling nodes with colors a, b, c, d, etc, using the minimal number of colors possible.

Answer:

The answer we were expecting to get was the following:



This answer assumes that in a statement such as `x:=y`, there is no interference between `x` and `y`—which there's not, since the read of `y` occurs before the write of `x`. However, several people assumed that `x` and `y` would interfere in that statement. Since we weren't completely clear in the lecture notes how to handle this situation, we gave almost full credit for an answer with that assumption, which is the following:


x0 a
x1 b
y0 a
x2 c
y1 b
z0 a