Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.035 Spring 2013

Test II Solutions

Mean 80 Median 83 Std. dev 16

I Generating Assembly for References

In this question, you'll generate code to implement C++'s pass-by-reference features in Decaf. In C++, if a function declaration contains a parameter with a declaration `int &x`, then `x` is passed by *reference* meaning that 1) the function receives the address of `x` and 2) an assignment to `x`, such as `x = 1`, automatically dereferences the address for `x` and stores `1` in that location.

1. [16 points]: Generate x64 assembly code for `swap`.

```
void foo()
{
  int a[2];
  a[0] = 0;
  a[1] = 1;
  swap(a[0], a[1]);
}
```

```
void swap(int &r1, int &r2)
{
  int t = r1;
  r1 = r2;
  r2 = t;
}
```

Write your assembly in AT&T syntax (src then dest). You should only use the instructions described in the table below. Remember that the first argument is passed in register `rdi` and the second in `rsi`. Finally, remember the simple x86_64 addressing modes: `%rax` references register rax, `(%rax)` references memory at the address in rax, and `100(%rax)` references memory at 100 bytes + the address in rax. **Only one dereference** (e.g. `(%rax)`) **is allowed per instruction.**

x86_64 instructions to use

`enter $n, $0`	Adjust stack for *n* bytes of local storage
`mov a, b`	Move value of a into destination b
`add a, b`	Add value of a to value in b; store in b
`call sym`	Call function `sym`
`leave`	Undo effects of `enter`
`ret`	Return from function call

```
foo:
 enter $16, $0
 mov $0, -8(%rbp)
 mov $1, -16(%rbp)
 mov $-8  %rdi
 add %rbp %rdi
 mov $-16 %rsi
 add %rbp %rsi
 call swap
 leave
 ret
```

Solution:

```
swap:
  enter $0, $0
  mov %(rdi) %r10
  mov %(rsi) %r11
  mov %r11 %(rdi)
  mov %r10 %(rsi)
  leave
  ret
```

II What Makes a Lattice a Lattice?

2. [2 points]: Given a partial order $\leq$ over a set P, for an element $a \in P$ to be an upper bound of a set $Q \subseteq P$, what must be true of a?

Solution: $\forall x \in Q \,.\, x \leq a$

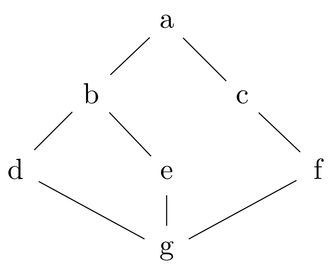
3. [3 points]: Given a partial order $\leq$ over a set P, for an element $a \in P$ to be a least upper bound of a set $Q \subseteq P$, what must be true of a?

Solution: Let X be the set of upper bounds of Q. Then $a \in X$ and $\forall x \in X \,.\, a \leq x$

4. [3 points]: True/False. All infinite lattices are incomplete. If true, give a proof. If false, give a counterexample – i.e., provide an infinite lattice that is complete.

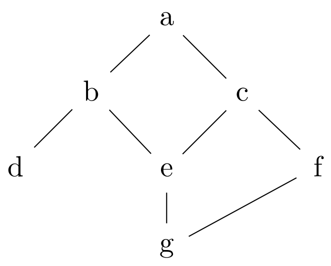
Solution: False. We can take $\mathbb{Z} \cup \{\infty, -\infty\}$ as the set of elements and use less-than-or-equal as the partial order.

5. [12 points]: Each of the following Hasse diagrams describe a different partial order $\leq_i$ for the set $P = \{a, b, c, d, e, f, g\}$. For each diagram, describe why $(P, \leq_i)$ is or is not a lattice.



A.

Solution: This is a lattice. There is both a least upper bound and a greatest lower bound for each pair of elements in P.



B.

Solution: This is not a lattice. For this to be a lattice there must be a greatest lower bound for each pair of elements in P. However, there is no greatest lower bound for d and g.

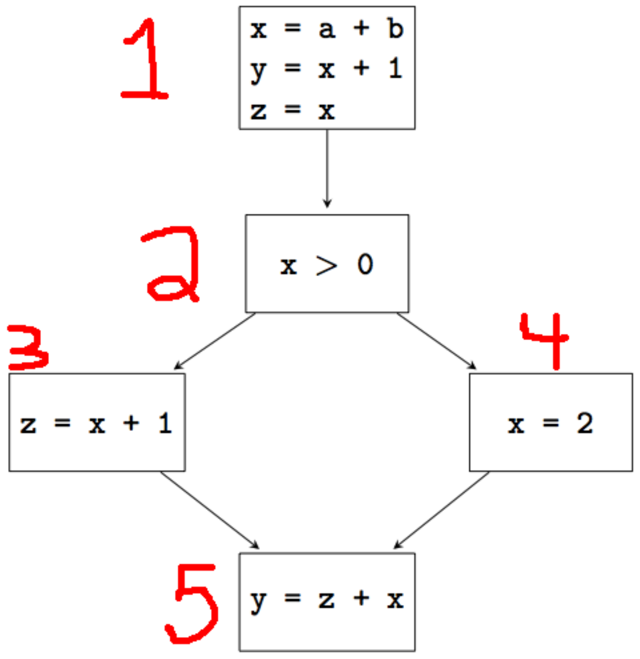
III Liveness Analysis

In this problem, you will perform liveness analysis on the following piece of code using a bit-vector formalization where the order of the variables in the vector is xyz:

```
x = a + b
y = x + 1
z = x
if (x > 0) {
    z = x + 1
} else {
    x = 2
}
y = z + x
```

6. [5 points]: Draw the control flow graph for this program. Label each basic block with a number n.

Solution:



Man, sometimes LaTeX just isn't worth it. -Cam

7. [8 points]: Compute GEN[n] and KILL[n] for each basic block n.

Solution:

GEN[1] = 000	KILL[1] = 111
GEN[2] = 100	KILL[2] = 000
GEN[3] = 100	KILL[3] = 001
GEN[4] = 000	KILL[4] = 100
GEN[5] = 101	KILL[5] = 010

8. [8 points]: Compute the least solution of the data flow equations, e.g. IN[n] = ... and OUT[n] = ... for each basic block. Assume that all variables are live after the end of the last basic block.

Solution:

IN[1] = 000	OUT[1] = 101
IN[2] = 101	OUT[2] = 101
IN[3] = 100	OUT[3] = 101
IN[4] = 001	OUT[4] = 101
IN[5] = 101	OUT[5] = 111

9. [6 points]: Do the results of liveness analysis on this code enable any optimization opportunities? If so, describe the optimization. If not, describe an optimization that uses liveness analysis and explain why it's not applicable.

Solution: Yes; y is not live at any point before its redefinition in block 5, and therefore its prior definition can be removed (freeing any register allocated to it).

IV Home on The Range

Ben Bittdiddle heard that instructions like `movb` (move a single byte) can be faster than instructions like `movq` (move an entire quadword). Because of this, he'd like to build an analysis that computes the range of values that an unsigned 64-bit integer variable may have so that his compiler knows when it's safe to use these other instructions on that variable. As so often happens on tests at MIT, for some reason, you have to help him with this.

Ben knows that to analyze the range of a variable in the program, he needs to define a lattice that defines the data-flow facts that the analysis will track for the variable.

Ben chooses to define the base elements of his lattice to be from the set $P = \{\ [l, u] \mid l \leq u \text{ and } 0 \leq l \text{ and } u \leq 2^{64}\}$. This is the set of all ranges $[l, u]$ of bounded 64-bit unsigned integers, where l is the lower end of the range (inclusive) and u is the upper end (inclusive).

Ben then chooses the following partial order for two ranges $[l_1, u_1] \in P$ and $[l_2, u_2] \in P$:

$$[l_1, u_1] \leq [l_2, u_2] \text{ if and only if } l_2 \leq l_1 \text{ and } u_1 \leq u_2.$$

10. [2 points]: Describe the relationship between $[l_1, u_1]$ and $[l_2, u_2]$ when $[l_1, u_1] \leq [l_2, u_2]$. Define their relationship in terms of their overlap/intersection, containment, or order.

Solution: $[l_1, u_1] \leq [l_2, u_2]$ implies that $[l_1, u_1]$ is contained within $[l_2, u_2]$.

11. [8 points]: Define the join operator, $[l_1, u_1] \vee [l_2, u_2]$, that is consistent with Ben's partial order.

Solution: $[l_1, u_1] \vee [l_2, u_2] = [\min(l_1, l_2), max(u_1, u_2)]$

12. [10 points]: Under Ben's partial order, is P a lattice? Why or why not? If not, explain how you would extend P to be a lattice.

Solution: P is not a lattice under Ben's partial order because there isn't a greatest lower bound for each pair of elements in P. For example, there isn't a greatest lower bound for the elements $[0, 0]$ and $[1, 1]$. We can however extend P with a bottom element $\perp$, such that $\forall p \in P\,.\perp \leq p$.

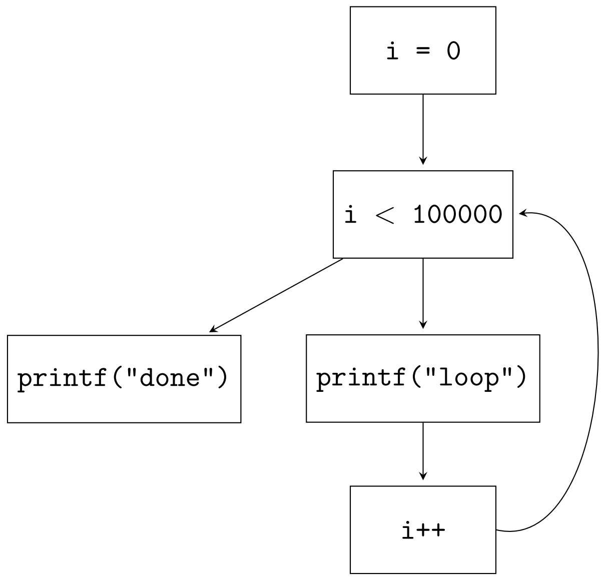
13. [7 points]: When Ben's compiler runs the analysis on a program, it will maintain a single lattice element for each of the program's variables. Assume that at the program point before a statement `a = b + c`, the lattice element for `b` is $[l_b, u_b]$ and the lattice element for `c` is $[l_c, u_c]$. What value will the transfer function compute for the lattice element $[l_a, u_a]$ for `a` at the program point after the statement? Assume that u_b and u_c are less than 2^{15}.

Solution: $\text{plus}(a, b) = \{(a_l + b_l), (a_u + b_u)\}$

14. [5 points]: Assuming that all the transfer functions in Ben's analysis are monotonic, does his analysis terminate? Why or why not?

Solution: Yes, the lattice is finite and therefore has the ascending chain property.

15. [5 points]: Ben implemented the data-flow framework you helped him with but is running into problems. In certain cases, it seems to take an inordinately long time, even for small code segments. So, he printed the control flow graph for one of the offending segments:



Why is this code slowing down the analysis framework? How can you fix this problem?

Solution: A vanilla implementation of this analysis framework (one that does not take into account conditionals) will naively increment i's interval on each iteration of the loop. The interval for i on the loop's backedge will therefore take the sequence of ascending values $[0, 0], [0, 1], [0, 2] .... [0, 2^{64}]$, at which point the algorithm finally terminates. We can instead modify this analysis by using a widening operator to replace the interval with the top element $([0, 2^{64}])$ directly after we observe an excessive number of ascending values.