CMSC 631, Fall 2010
Practice problems

1. Let S be a finite set, and let L be the lattice of subsets of S, with order $\subseteq$. Show that any function $f(x)$ constructed from union, intersection, and constant sets is monotonic. Here, I mean that $f(x) = e$ where e can be specified by the grammar

$$e ::= x \mid S' \mid e \cup e \mid e \cap e$$

   where S' is any subset of S. Your proof should be by induction on the structure of e.

2. Suppose that we extend the grammar for e from problem 1 to include the complement operator $!e$, where $!T = S - T$. Is f still guaranteed to be monotonic? If it is, justify your answer. If it's not, explain why a transfer function defined by $Out(stmt) = Gen(stmt) \cup (In(stmt) - Kill(stmt))$, which seems to include negation, is monotonic.

3. Let A be a lattice, with order $\leq$. Define $A \to A$ to be the set of all functions from A to A, and define $f \leq' g$ if $f(x) \leq g(x)$ for all $x \in A$.

   - Show that $A \to A$ with order $\leq'$ is also a lattice. That is, show that for all $f, g \in A \to A$, $f \sqcup g$ and $f \sqcap g$ always exist.
   - Suppose lattice A has height h and that A is finite with n elements. What is the height of the lattice $(A \to A, \leq')$? (When counting height, count "edges" rather than "nodes," e.g., if A were the lattice $\{a, b\}$ with $a < b$, then its height would be 1.)

4. (ASU, exercise 10.35) In class we talked about how an analysis is *conservative* if it models the behavior of the program in a way that is safe. As it turns out, "safe" is in the eye of the beholder.

   When performing dataflow analysis to estimate the following properties, determine whether too-large or too-small estimates are conservative. Explain your answer in terms of the intended use of the information. (Hint: This is a bit of a trick question.)

   (a) Available expressions

   (b) Variables changed by a procedure

   (c) Variables not changed by a procedure

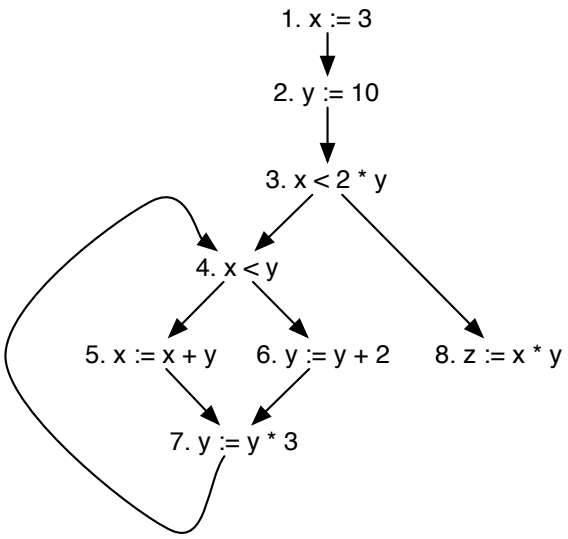
   (d) Copy statements reaching a given program point

5. For the following control flow graph

   (a) Draw the dominator tree

   (b) List the dominance frontiers of each node (assume nodes 7 and 8 go to exit)

   (c) Put the control-flow graph in SSA form (you can eyeball this instead of running the algorithm by hand)



6. Write down a sequence of reduction steps reducing each of the following terms to normal form. For this problem, reduction is allowed anywhere within a term, including under a λ.

   (a) $(\lambda x.x(xy))(\lambda u.u)$

   (b) $(\lambda xyz.zyx)aa(\lambda pq.q)$

   (c) $(\lambda xyz.xz(yz))(\lambda xy.x)(\lambda xy.x)$

   Note: $\lambda xy.e$ is short for $\lambda x.\lambda y.e$. Remember also that the scope of λ extends as far to the right as possible, and that application associates to the left.

7. (a) Give an encoding of lists in the lambda calculus. Your encoding should include combinators *nil*, *cons*, *head*, *tail*, and *isnil*, with the following requirements:

   - *head* (*cons* e_1 e_2) $= e_1$
   - *tail* (*cons* e_1 e_2) $= e_2$
   - *isnil nil* e_1 $e_2 = e_1$
   - *isnil* (*cons* e_1 e_2) e_3 $e_4 = e_4$

   Here $=$ is beta-equality. Argue that your encoding is correct by showing that your combinators adhere to the above rules. *Hint:* This is exercise 5.2.8 in Pierce, chapter 5, which suggests representing the list $[x, y, z]$ as $\lambda cn.cx(cy(czn))$.

   (b) Using Y and your combinators from part (a), write the *map* function, where (using OCaml notation) *map* f $[] = []$ and *map* f $(x :: xs) = (f\ x) :: (map\ f\ xs)$. (Please use the combinators as primitives and do *not* expand them to their definitions.)

8. (Barendregt exercise 6.8.14) Let

$$X = \lambda abcdefghijklmnopqstuvwxyzr.r(thisisafixedpointcombinator)$$
$$Z = XXXXXXXXXXXXXXXXXXXXXXXXXX \text{ (26 } X\text{'s)}$$

   Show that Z is a fixed point combinator (that is, show that $ZX = X(ZX)$ under beta equivalence).

9. For each type, construct a simply-typed lambda calculus term (variables, functions, and function application only) whose most general type is that type, or argue that no term has that type. (Hint: You can double-check your answers in OCaml.)

(a) $\alpha \to \beta \to \beta$

(b) $(\alpha \to \beta \to \gamma) \to \beta \to \alpha \to \gamma$

(c) $\alpha \to \beta$

(d) $\alpha \to \alpha \to \alpha$

10. Does the simply-typed lambda calculus with integers have a *subject expansion* property, meaning if $\Gamma \vdash e : \tau$ and $e' \to e$, does $\Gamma \vdash e' : \tau$? Here $\to$ is reduction under call-by-value semantics. Either prove that subject expansion holds, or give a counterexample showing that it does not hold.

11. Suppose we were to add booleans to the simply-typed lambda calculus:

$$e ::= x \mid n \mid \mathit{true} \mid \mathit{false} \mid \lambda x.e \mid e\ e \mid \mathit{if}\ e\ \mathit{then}\ e\ \mathit{else}\ e$$

(a) Write down small-step call-by-value semantic rules for the new forms *true*, *false*, and *if*. (Here *if* should behave as it does in O'Caml, evaluating to the result of either the true or false branch depending on the guard.)

(b) Extend the typing judgment $\Gamma \vdash e : \tau$ to the new forms *true*, *false*, and *if*.

(c) Prove progress and preservation for the extended language. (You don't need to reprove the cases for the old forms.)

12. Consider the following program, written in lambda calculus with tuples, integers, and strings:

let app2 $= \lambda f x y.(f\ x, f\ y)$ *in*
app2 $(\lambda x.x)$ 1 "foo"

Write down the type for *app2* in simply-typed lambda calculus with Hindley-Milner style polymorphism. Does this program exhibit any run-time errors (i.e., will its evaluation ever be stuck)? Does the program type check using Hindley-Milner style polymorphism? Explain what goes wrong. Can you give a type for *app2* that is polymorphic but not Hindley-Milner such that this program would type check? (Note: You will not be able to construct a most-general type for *app2* without using intersection types, which we have not discussed, but your type should work for this particular use of *app2*.)

13. Consider the lambda calculus with Hindley-Milner style polymorphism. Here $e ::= x \mid \lambda x.e \mid e\ e \mid$ *let* $x = e$ *in* e and types are given by $\sigma ::= \forall \alpha.\sigma \mid \tau$ and $\tau ::= \alpha \mid \mathit{int} \mid \tau \to \tau$, with the following type rules:

$$\textsc{Var}\ \frac{\Gamma(x) = \forall \vec{\alpha}.\tau \qquad \text{any } \vec{\tau}}{\Gamma \vdash x : \tau[\vec{\alpha} \mapsto \vec{\tau}]} \qquad \textsc{Lam}\ \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \to \tau'}$$

$$\textsc{App}\ \frac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2 : \tau'} \qquad \textsc{Let}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \vec{\alpha} = FV(\tau_1) - FV(\Gamma) \qquad \Gamma, x : \forall \vec{\alpha}.\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathit{let}\ x = e_1\ \mathit{in}\ e_2 : \tau_2}$$

Here $FV(\cdot)$ stands for the free variables of a type or type environment. Prove the *polymorphic substitution lemma*, which is used in proving subject reduction: If $\Gamma, x : \forall \vec{\alpha}.\tau_1 \vdash e_2 : \tau_2$ where $\vec{\alpha} = FV(\tau_1) - FV(\Gamma)$ and $\Gamma \vdash e_1 : \tau_1$, then $\Gamma \vdash e_2[x \mapsto e_1] : \tau_2$.