Programming Languages [Fall 2014]
Test III

NAME: ____

Instructions:

1) This test is 8 pages in length.

2) You have 2 hours to complete and turn in this test.

3) This test is closed books, notes, papers, friends, neighbors, phones, etc.

4) Use the backs of pages in this test packet for scratch work. If you write more than a final answer in the area next to a question, circle your final answer.

5) Write and sign the following:
"I pledge my Honor that I have not cheated, and will not cheat, on this test."

____

____

Signed: ____

1. Essay (graded on accuracy, thoroughness, and readability) [20 points]
Type-safe PLs seem to have the momentum of a runaway freight train. Why are they so popular?

2. [15 points]

a) Encode a 3-value logic into the untyped lambda calculus. The three values are: T (true), F (false), and B (both). Besides for the values themselves, provide an encoding for the expression if(e_1)then(e_2)else(e_3). This expression works like a regular "if" expression when e_1 evaluates to T or F, but when e_1 evaluates to B the following occurs: e_2 is executed; if e_2 converges to a value then e_3 is also executed; then if e_3 converges to a value v then v is the final result. Your encoding must be lazy (e.g., if $e_1 \to^* T$ then e_3 doesn't get evaluated). Assume CBV evaluation.

b) Using the call-by-value strategy and your response to Part (a), trace the evaluation of if(B)then(if(F)then(F)else(F))else(T). Show each step and underline redexes.

3. [65 points]

Consider the following syntax for type-safe language L, which has types for integers and records.

types τ ::= int | $\{l_1{:}\tau_1 .. \; l_m{:}\tau_m\}$

exprs e ::= x | n | $\{l_1{=}e_1 .. \; l_m{=}e_m\}$ | e.l | let x = e_1 in e_2

Notes: (1) m is always a positive integer (2) Evaluation in L is *left to right* and *call by name* (which here means that *let* expressions are evaluated lazily); otherwise L is as discussed in class.

(a) Using the following SML definitions for L:

```
datatype exp = V of string | N of int | R of (string*exp) list
             | S of exp*string | L of string*exp*exp;
exception captured;
```

implement an SML function `sub:exp->string->exp->exp` such that `sub e x e′` returns `[e/x]e′` unless a variable gets captured, in which case *captured* gets raised.

Reminder: $e ::= x \mid n \mid \{l_1=e_1 \,..\, l_m=e_m\} \mid e.l \mid \text{let } x = e_1 \text{ in } e_2$

(b) Define *alpha*-equivalence for L (in a deductive system, not in code).

(c) Define L's static semantics.

Reminder: $e ::= x \mid n \mid \{l_1{=}e_1 \,..\, l_m{=}e_m\} \mid e.l \mid \text{let } x = e_1 \text{ in } e_2$

(d) Define L's SOS-style dynamic semantics ($e \rightarrow e'$), without using evaluation contexts.

(e) Redefine the small-step semantics for L, this time using evaluation contexts.

(f) Define L's big-step operational semantics ($e \Downarrow v$).

(g) Using your rules for L, as well as the following:

$$\boxed{e \rightarrow^* e'} \qquad \frac{}{e \rightarrow^* e}\ \text{Re} \qquad \frac{e \rightarrow e' \quad e' \rightarrow^* e''}{e \rightarrow^* e''}\ \text{Tr}$$

prove that $\forall e,v\colon\ (e \rightarrow^* v) \Rightarrow (e \Downarrow v)$.

[The following problem is for grad students; undergrads may do it for +5 extra credit]

4. [16 points]
Let D be diML with constructs added—as defined in class—for *binary* product and sum, unit, and recursive types. (Recall that D uses *let* expressions to eliminate product types and *case* expressions to eliminate sum types.) Implement in D a function `rev` that takes an arbitrary list L of integers and returns L reversed. In your response, please feel free to define and use abbreviations for types, but avoid non-D syntax (e.g., refrain from using syntactic sugar).