CSE526: Principles of Programming Languages (Spring 2003)

Scott Stoller

Exam 2 (version 24apr2003)

Each problem is worth 20 points. Justify your answers.

Problem 1

Consider adding a new command CAS ("compare and swap") to the language of chapter 8. The syntax of CAS is:

$$\langle \text{comm} \rangle ::= \textbf{CAS}(\langle \text{var} \rangle, \langle \text{var} \rangle, \langle \text{intexp} \rangle, \langle \text{var} \rangle)$$

Informally, the semantics of **CAS**(x,old,new,out) is that it executes the following two atomic steps.

(1) evaluate the expression "new" to an integer k.

(2) if x=old then (x:=k; out:=1) else (old:=x; out:=0).

Each of these two steps executes atomically. Transitions of other threads may occur between the steps.

(a) Extend the transition semantics of section 8.1 with transition rule(s) for CAS. (If necessary, you may augment the variety of configurations by introducing additional kinds of commands.)

(b) An alternative way to give semantics for CAS is to treat it as syntactic sugar, by defining it in terms of critical regions (Section 8.2). Give such a definition of CAS. This semantics for CAS must be consistent with the above informal semantics, but it does not need to be exactly equivalent to the semantics in part (a).

Problem 2

Consider the lambda calculus expression $(\lambda f.f(fI))P_1$, where we used the following abbreviations: $I = (\lambda x.x)$, $P_1 = (\lambda x.\lambda y.x)$.

(a) Give a proof of the normal order evaluation of this expression using the inferences rules in Section 10.3. (You may write the proof as a sequence of steps, like the textbook does, or as a tree, like I occasionally did in class.)

(b) Give the result of eager evaluation of this expression using the inference rules in Section 10.4. (For part (b), you do not need to show the proof in detail, although you should still sketch the basic steps to justify your answer.)

Problem 3

Exercise 11.8.

Problem 4

Consider the expression

$$\textbf{letrec } v_0 \equiv \lambda u_0.e_0, \ldots, v_{n-1} \equiv \lambda u_{n-1}.e_{n-1} \textbf{ in } e \tag{1}$$

When $n > 1$, this expression defines a collection of mutually recursive functions. Your friend proposes to simplify the programming language by restricting **letrec** to define a single function; in other words, the new syntax of **letrec** is

$$\textbf{letrec } v \equiv \lambda u.e' \textbf{ in } e'' \tag{2}$$

Is it possible to treat expressions like (1) as syntactic sugar by defining them in terms of expressions like (2)? If so, give such a definition. If not, explain informally why it is impossible.

Problem 5

In the Iswim-like language of chapter 13, define a function newIter that, when applied to a list, returns a pair of functions $\langle hasNext, next \rangle$ that work like an iterator in Java. For example, newIter can be used as follows to define a function "sum" that, when applied to a list of integers, returns the sum of the elements of the list

let newIter $\equiv \cdots$ **in**
let sum $\equiv \lambda$ x. **let** $\langle$hasNext, next$\rangle$ = newIter x **in**
newvar s:=0 in (**while** hasNext() **do** s:=(**val** s)+next()); **val** s

Note: This definition of "sum" uses the syntactic sugar of Section 13.4.