cs173: Programming Languages
Midterm Exam
Fall 2000

The questions on this exam are worth 100 points.
Exam rules:

- Your responses to this exam are due before the start of class on 2000-11-08.
- If you believe a question is underspecified, make a reasonable assumption, and document your claim and your work-around.
- The exam is open book with respect to the course text (Krishnamurthi and Felleisen) and course lecture notes (Tucker and Krishnamurthi, with a guest appearance by Blaheta). You may cite these by lecture date. You may not refer to any other sources related to the course between the time you start and finish the exam.
- You have three hours and 15 minutes to take this exam. Time begins when you start reading the exam questions. The first three hours must be a single, contiguous block. After you have completed this period, you must close your responses. At some later point, at your discretion (and before the responses are due), you may open your responses for one 15 minute period to edit or append to them.
- All code responses must use the dialect of Scheme employed in this course. You may not use side-effects or continuations unless expressly permitted to do so in the problem statement.
- Staple your solutions together. Put your name or initials on every page.
- Solutions may be hand-written, but the burden of legibility rests on you.
- You may not evaluate any of programs related to this exam on a computer.
- You must neither give assistance to any other course participant, nor receive assistance from anyone other than the course staff, on material under examination.

Brown University has an Academic Code that governs this exam and all our other transactions. I expect you, as students and scholars, to abide by it faithfully and fully.

Problem 1 [15 points]

Continuing our obsession with the Greek alphabet, let us consider the ι-calculus (IC). The IC is an extension to the (eager) λ-calculus. It extends the language with a new keyword, **iota**. **iota** is always bound to the time elapsed since the program began execution (as in, "an iota of time"), so the value of **iota** tells the programmer how long the program has been running. Time is discrete. It begins at 0, and increases by 1 at each function application. The IC can be used to profile program execution or to implement pre-emptive multi-tasking.

We've extended the abstract syntax for you. Extend or modify the interpreter to implement the ι-calculus. You may use mutation in your interpreter.

```
(define-datatype IC IC?
  [numE (n number?)]
  [varE (v symbol?)]
  [addE (lhs IC?) (rhs IC?)]
  [funE (param symbol?) (body IC?)]
  [appE (func IC?) (arg IC?)]
  [iotaE])                        ;; new syntax clause

;; assume the implementation of D-Sub

(define (interp a d)
  (cases IC a
         [numE (n) n]
         [varE (v) (get-sub v d)]
         [addE (le re) (+ (interp le d) (interp re d))]
         [funE (param body)
               (λ (argval)
                 (interp body (new-sub param argval d)))]
         [appE (fe ae) ((interp fe d) (interp ae d))]))
```

Problem 2 [15 points]

Your slightly daffy boss, having once taken a survey of programming languages course, fancies himself as a language designer. It's your lot in life to satisfy his megalomania. Fascinated by tinkering with features, he asks you to extend your in-house implementation with the following construct, **i-letrec** (short for "irreflexive letrec"), which is syntactically similar to but semantically a cross between **let** and **letrec**.

Syntax:

(**i-letrec** ([*var expr*] ...) *expr*)

Binding:

The expression

(**i-letrec** ([var_1 $expr_1$] ... [var_n $expr_n$]) *body*)

extends its inherited environment as follows:

- $\{var_1, ..., var_n\} - \{var_k\}$ are bound in $expr_k$
  (In contrast, **letrec** binds all the var_i in each $expr_k$, and **let** binds none of them)
- $\{var_1, ..., var_n\}$ are bound in *body*
  (All of these variables are bound in *body* in **letrec** and **let**, too)

N.B.: $-$ denotes set difference.

Semantics:

i-letrec evaluates all the $expr_i$ in parallel, assigns each value to the corresponding var_i, then evaluates and returns the value of *body*. Any use of a var_i in an $expr_j$ must be in the body of a function. (This is also expected of **letrec** terms.) Thus, (**i-letrec** ([x 3] [y x]) y) is illegal, but (**i-letrec** ([x 3] [y (λ () x)]) (y)) is not.

Problem:

Write three side-effect-free Scheme expressions that are syntactically identical in all respects except where one uses **let**, another uses **letrec**, and the third **i-letrec**. No two expressions may evaluate to the same result. None of them should produce an error. At most one of the expressions may fail to terminate. Specify the resulting value for each expression. The solution expression may use **let** and **letrec**. (You may of course write the expression just once, and indicate the locus of the change.)

Problem 3 [25 points]

Consider the language

$$
\begin{array}{rcll}
L & ::= & N & \text{where } N \in \{1, 2, 3, \ldots\} \\
  & | & (L + L) & \\
  & | & id & \\
  & | & (\textbf{fun}\ (id)\ L) & \\
  & | & (L\ (L)) & \\
  & | & (\textbf{set}\ id\ L) &
\end{array}
$$

Provide an operational semantics for this language using the style of semantics in the lecture on 2000-10-06. The semantics of **set** is the same as that of Scheme's **set!**. This is the same exercise as homework Program 3 because there is no mutation operation in mathematics, either. The difference lies not in the solution but in how you express it: you must use mathematics, not Scheme code. You may use standard mathematical values such as sets, tuples, and so forth.

Problem 4

[25 points]

Consider the language

$$
\begin{array}{rcll}
L & ::= & N & \text{where } N \in \{1, 2, 3, \ldots\} \\
 & | & x & \\
 & | & (\textbf{fun}\ (x)\ L) & \\
 & | & (L\ L) &
\end{array}
$$

You are given the following rules:

$$
\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{}{\Gamma \vdash N : num}
$$

$$
\frac{\Gamma[x : \tau'] \vdash L : \tau}{\Gamma \vdash (\textbf{fun}\ (x)\ L) : (\tau' \rightarrow \tau)} \qquad \frac{\Gamma \vdash L_f : (\tau' \rightarrow \tau) \quad \Gamma \vdash L_a : \tau'}{\Gamma \vdash (L_f\ L_a) : \tau}
$$

The rule for **fun** is "magical": it uses variables to guess types, expecting the user to eventually resolve the guesses.
Provide a type for the term

$$(((\textbf{fun}\ (x)\ x)\ (\textbf{fun}\ (x)\ x))\ 3)$$

by generating and solving one or more constraints. The final type must not contain any variables: it must be in terms of num and $\rightarrow$ only.
Suggestion: Turn the page by 90° before you start writing.

Problem 5 [20 points]

Consider the following features we have studied in class:

(1) parametric polymorphism

(2) polymorphic datatypes

Many languages that offer one of these features offer both, e.g., ML and Haskell. Does it make sense to include them individually?

- Consider a language with (1) but not (2). Characterize the programs you can write that exploit (1). [10 points]
- Consider a language with (2) but not (1). What kinds of programs can you write that exploit (2)? [10 points]