cs173: Programming Languages

Midterm Exam

Fall 2002

Please read this page of instructions
before you turn the page!

This exam is worth 181 points. We will assign partial credit to partial responses, provided we can understand the response and it reflects a correct understanding of course material.
Exam rules:

- Your responses to this exam are due at the beginning of class on 2002-10-28 (Monday).
- On turning in solutions:
  - Solutions may be hand-written, but the burden of legibility rests on you.
  - If possible, please answer the questions on the exam sheets, and within the space provided. If you do use separate sheets of paper, be sure to make clear where your work on a problem begins and ends.
  - Staple your solutions together. *Put your name on every page.*
- Do not assume that the length of an answer should be proportional to the number of points assigned. Even for weighty problems, the correct solution can take just a few words, especially if you use the right technical phrases! No problem on this exam asks for, or even really wants, a lengthy essay as a solution.
- If you believe a question is underspecified, make a reasonable assumption and document it.
- The exam is open book with respect to the course notes for the year 2002. The exam may refer to specific lectures, so it would be useful to have them handy before you begin to respond. 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. At some point, either immediately or later (but before the responses are due), you may use an additional 15 contiguous minutes to change or augment your answers.
- All Scheme code responses must use the dialects employed in this course.
- You may not evaluate any 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. Act honorably.

Problem 1 [10 points]

The program

```
{let {x 4}
  {let {f {fun {y}
            {+ x y}}}
    {let {x 5}
      {f 10}}}}
```

should evaluate to `14` by static scoping. Evaluating `x` in the environment at the point of invoking `f`, however, yields a value of `15` for the program. Ben Bitdiddle, a sharp if eccentric student, points out that we can still use the dynamic environment, so long as we take the *oldest* value of `x` in the environment rather than the *newest*—and for this example, he's right!

Is Ben right in general? If so, justify. If not, provide a counterexample program and explain why Ben's evaluation strategy would produce the wrong answer. (A bonus point for explaining why Ben's way of thinking about environments is conceptually wrong, irrespective of whether or not it produces the right answer.)

Problem 2 [25 points]

Consider the following program (the Scheme primitive *display* prints its argument to the console):

```
(define (tester k1)
  (begin
    (display "beginning ")
    (let/cc k2 (k1 k2))
    (display "middle ")
    (let/cc k3 (k1 k3))
    (display "end")))

(tester (let/cc k0 k0))
```

What output does this program print to the console? Write down each of the continuations (escapers) explicitly. (If you think it doesn't terminate, use some reasonable representation for the infinite stream of output and similarly for the continuations.) Suggestion: To avoid turning in a messy solution that we cannot grade, it may be better to solve the problem on a separate sheet and copy your final response to the sheet you turn in.

Problem 3 [15 points]

In class, we discussed that there are three fundamental ways of writing programs that distinguish between eager and lazy evaluation regimes in a language like Java. What are the three?

Of these, one is not a difference observable by a program. Which one, and why not?

Problem 4 [20 points]

In our lazy interpreter, we identified three points in the language where we need to force evaluation of expression closures (by invoking *strict*): the function position of an application, the test expression of a conditional, and arithmetic primitives. Doug Oord, a fairly sedentary student, is rather taken with the idea of laziness. He suggests that we can reduce the amount of code we need to write by replacing all invocations of *strict* with just one. In the interpreter from 2002-09-30, he removes all instances of *strict* and replaces

[*id* (*v*) (*lookup v env*)]

with

[*id* (*v*) (*strict* (*lookup v env*))]

Doug's reasoning is that the only time the interpreter returns an expression closure is on looking up an identifier in the environment. If we force its evaluation right away, we can be sure no other part of the interpreter will get an expression closure, so removing those other invocations of *strict* will do no harm. Being lazy himself, however, Doug fails to reason in the other direction, namely whether this will result in an overly eager interpreter.

Write a program that will produce different results under the original interpreter and Doug's. Write down the result under each interpreter, and clearly identify which interpreter will produce each result. You may assume that the interpreted language features arithmetic, first-class functions, `with`, `if0` and `rec` (even though these aren't in our in-class lazy interpreter).

Hint: Be sure to compare this behavior against that of the lazy interpreter of the sort we've written in class, not the behavior of Haskell!

Problem 5 [25 points]

No lazy language in history has also had state operations (such as mutating the values in boxes, or assigning values to variables). Why not? The best answer to this question would include two things: a short program (which we assume will evaluate in a lazy regime) that uses state, and a brief explanation of what problem the execution of this program illustrates. Please be sure to use the *non*-caching (ie, original) notion of laziness. If you present a sufficiently illustrative example (which needn't be very long!), your explanation can be quite short.

Problem 6 [25 points]

In the lecture notes on implementing recursion (dated 2002-09-23), we asked you to ponder lifting the restriction that the named expression must syntactically be a function. Given below is the core of the interpreter. Modify it in two ways:

- lift this restriction, so the named expression is an arbitrary expression that must be evaluated; and,
- if the program accesses the `rec`-bound identifier and it does not hold a function value, the interpreter halts with an error.

Clearly mark all lines that need to change and indicate their changes unambiguously. Likewise, if you add any new code, clearly indicate where it should be inserted. Hint: Unless your solution is very different from ours, you shouldn't need to modify any lines of code not shown below.

```
(define-datatype Env Env?
  [mtSub]
  [aSub (name symbol?)
        (value RCFWA-value?)
        (env Env?)]
  [aRecSub (name symbol?)
           (value (lambda (x)
                    (and (box? x)
                         (RCFWA-value? (unbox x)))))
           (env Env?)])

(define-datatype RCFWA-value RCFWA-value?
  [numV (n number?)]
  [closureV (param symbol?)
            (body RCFWAE?)
            (env Env?)])

(define (lookup name an-env)
  (cases Env an-env
    [mtSub () (error 'lookup (string-append "unbound: " (symbol→string name)))]
    [aSub (bound-name bound-value rest-env)
          (if (symbol=? bound-name name)
              bound-value
              (lookup name rest-env))]
    [aRecSub (bound-name bound-value-box rest-env)
             (if (symbol=? bound-name name)
                 (unbox bound-value-box)
                 (lookup name rest-env))]))

(define (cyclically-bind-and-interp bound-id named-expr env)
  (local ([define value-holder (box (numV 1729))]
          [define new-env (aRecSub bound-id value-holder env)]
          [define named-expr-val (interp named-expr new-env)])
    (begin
      (set-box! value-holder named-expr-val)
      new-env)))
```

(continued on next page)

```
(define (interp expr env)
  (cases RCFWAE expr
    [num (n) (numV n)]
    [add (l r) (numV+ (interp l env) (interp r env))]
    [sub (l r) (numV− (interp l env) (interp r env))]
    [mult (l r) (numV∗ (interp l env) (interp r env))]
    [with (bound-id named-expr bound-body)
          (interp bound-body
                  (aSub bound-id
                        (interp named-expr env)
                        env))]
    [rec (bound-id named-expr bound-body)
         (cases RCFWAE named-expr
                [fun (fun-param fun-body)
                     (interp bound-body
                             (cyclically-bind-and-interp bound-id
                                                         named-expr
                                                         env))]
                [else (error 'rec "expecting a syntactic function")])]
    [if0 (test then otherwise)
         (if (numV-zero? (interp test env))
             (interp then env)
             (interp otherwise env))]
    [id (v) (lookup v env)]
    [fun (param body)
         (closureV param body env)]
    [app (fun-expr arg-expr)
         (local ([define fun-val (interp fun-expr env)]
                 [define arg-val (interp arg-expr env)])
           (cases RCFWA-value fun-val
             [closureV (cl-param cl-body cl-env) (interp cl-body
                                                         (aSub cl-param
                                                               arg-val
                                                               cl-env))]
             [else (error 'interp "can only apply functions")]))])))
```

Problem 7 [15 points]

Define *next-location* (from the lecture on implementing state, dated 2002-10-04) in a purely functional manner, i.e., without using boxes or other effects. The ideal solution is a Scheme function that replaces the implementation given in the notes. If you are not comfortable enough programming in Scheme you may provide pseudocode instead, but the onus of legibility rests on you, so choose this option with care. In either case, you are responsible for the clarity of your code.

Problem 8 [45 points]

We studied mutation through boxes, which is related to but different from the notion of mutation found in most programming languages, such as C and Java. In those languages, one mutates a *variable* (we can now use this term in place of *identifier*) directly, rather than a box bound to an identifier: for instance, a programmer can write

```
x = 4;
```

To model this operation, we define the language MSFAE, which has functions, arithmetic, sequencing, and one one new operator, `:=`. Its BNF is as follows:

```
MSFAE ::= <var>
        | <number>
        | {+ <MSFAE> <MSFAE>}
        | {fun {<var>} <MSFAE>}
        | {<MSFAE> <MSFAE>}
        | {seqn <MSFAE> <MSFAE>}
        | {:= <var> <MSFAE>}
```

The semantics of `:=` is to first evaluate the sub-expression, then assign its value to the named variable, and finally return the sub-expression's value as that of the entire expression. We will assume that the parser converts instances of `with` into immediate function definition and application. Thus, for instance, the program

```
{with {x 3}
  {+ {:= x 4}
     x}}
```

evaluates to 8 (the assignment changes x to 4 and returns that value; addition operates left-to-right; looking up x yields its new value, 4, for a sum of 8), while

```
{with {x 2}
  {+ {with {x 3}
       {:= x 4}}
     x}}
```

evaluates to 6, since the mutation affects only the inner x.

Let's implement this model. As usual, we want to avoid writing an interpreter that uses too many Scheme features, so we'll use a purely functional interpreter. Given below is the skeleton of an interpreter written in store-passing style. You will need to complete the definition of *interp*, and provide implementations for any auxiliary procedures you introduce. On the next two pages, we have given you an interpreter based on that from 2002-10-04; you need not provide implementations of procedures already in the notes (such as *next-location*)—make sure you don't change any of them! You shouldn't need to modify any code we've given, but if you want to, you are welcome to do so.

Hint: You will want the environment to map names to locations, and the store to map locations to values. In the absence of `newbox`, what should allocate a new store location?

```
(define-datatype MSFAE MSFAE?
  [var (name symbol?)]
  [num (n number?)]
  [add (lhs MSFAE?) (rhs MSFAE?)]
  [fun (param symbol?) (body MSFAE?)]
  [app (fun-expr MSFAE?) (arg-expr MSFAE?)]
  [seqn (expr1 MSFAE?) (expr2 MSFAE?)]
  [asgn (var symbol?) (expr MSFAE?)])

(define-datatype Env Env?
  [mtSub]
  [aSub (name symbol?)
        (value number?)
        (env Env?)])

(define-datatype MSFA-value MSFA-value?
  [numV (n number?)]
  [closureV (param symbol?)
            (body MSFAE?)
            (env Env?)])

(define-datatype Store Store?
  [mtSto]
  [aSto (location number?)
        (value MSFA-value?)
        (store Store?)])

(define-datatype Value*Store Value*Store?
  [v-s (value MSFA-value?) (store Store?)])

;; env-lookup : symbol Env → location

(define (env-lookup name an-env)
  (cases Env an-env
    [mtSub () (error 'env-lookup
                     (string-append "unbound: " (symbol→string name)))]
    [aSub (bound-name bound-value rest-env)
          (if (symbol=? bound-name name)
              bound-value
              (env-lookup name rest-env))]))

;; store-lookup : location Store → MSFA-value

(define (store-lookup location a-store)
  (cases Store a-store
    [mtSto () (error 'store-lookup
                     (string-append "invalid store location: " (number→string location)))]
    [aSto (filled-location filled-value rest-store)
          (if (= location filled-location)
              filled-value
              (store-lookup location rest-store))]))
```

(continued on next page)

```
;; interp : MSFAE Env Store → Value*Store

(define (interp expr env store)
  (cases MSFAE expr
    [num (n) (v-s (numV n) store)]
    [add (l r)
         (cases Value*Store (interp l env store)
           [v-s (lhs-val lhs-sto)
                (cases Value*Store (interp r env lhs-sto)
                  [v-s (rhs-val rhs-sto)
                       (v-s (numV+ lhs-val rhs-val) rhs-sto)])])]
    [var (v) ...]
    [fun (param body)
         (v-s (closureV param body env)
              store)]
    [app (fun-expr arg-expr)
         (cases Value*Store (interp fun-expr env store)
           [v-s (fun-val fun-sto)
                (cases Value*Store (interp arg-expr env fun-sto)
                  [v-s (arg-val arg-sto)
                       (cases MSFA-value fun-val
                         [closureV (cl-param cl-body cl-env)
                                   ...]
                         [else (error 'interp
                                      "can only apply functions")])])])]
    [seqn (e1 e2)
          (cases Value*Store (interp e1 env store)
            [v-s (e1-val e1-sto)
                 (interp e2 env e1-sto)])]
    [asgn (var expr)
          ...]))
```

Problem 9 [1 points]

Name the author of *The Kon Tiki Expedition*.