cs173: Programming Languages

Midterm Exam

Fall 2001

This exam is worth 90 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 by 2200hrs on 2001-11-06. We will post turn-in instructions on the course newsgroup.
- 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 notes for the year 2001. You may not refer to any other sources related to the course between the time you start and finish the exam. The exam may refer to specific lectures, so it would be useful to have them handy before you begin to respond.
- 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 dialect employed in this course. You may not use assignments or continuations unless expressly permitted to do so in the problem statement.
- Staple your solutions together. *Put your name 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 [20 points]

For this assignment, you don't need to CPS primitives such as *empty?*, *first*, *cons* and $<$.
CPS the following Scheme function:

```
(define (filter f l)
  (cond
   [(empty? l) empty]
   [else (cond
          [(f (first l)) (cons (first l)
                               (filter f (rest l)))]
          [else (filter f (rest l))])]))
```

(Please write the converted code in the space below, instead of modifying the code above.) You may assume that the function argument to *filter* is in CPS.

Now CPS the following expression to use the converted *filter*:

```
(filter (lambda (x) (< x 3))
        (cons 1 (cons 4 empty)))
```

(Likewise, please write the solution below.)

Problem 2 [20 points]

Consider the Rip language with numbers, variables, addition, procedures, application **and assignment**. (Assume static scope and eager evaluation.) We've seen an interpreter for this language. Now write an operational semantics for it. Be sure to clearly specify the contracts for *all* mappings. (In a language without assignment, for instance, the environment maps names to values.) You may use sets, tuples and other mathematical notation, as necessary, but don't invent any of your own!

Problem 3 [15 points]

Consider the language with assignment from the previous problem, augmented with explicit type annotations. Write down type judgments representing a reasonable type system for this language. The type judgments you saw in class were for a language without assignment. You will need to

- think about whether adding assignment changes the form of the judgments, and
- add at least one judgment, for the assignment construct.

If the existing judgments don't change, please indicate this explicitly, otherwise write them in their new form.

Problem 4 [10 points]

Consider the type judgments discussed in the lecture notes for 2001-10-22 and 2001-10-24. These rules are for an *eager* version of Rip. Suppose Rip were *lazy* instead (i.e., it evaluates arguments only on demand). Indicate which of the judgments would change, and how, by writing the modified judgments below. You don't need to write judgments that don't change, but if you believe *none* change, indicate why.

Problem 5 [25 points]

In the Rip language we've seen so far, it's impossible to write a generic library procedure that swaps the values of two variables. For instance, consider the following Rip code (we'll assume **let** is transformed automatically into procedure application):

```
{let {swapper {proc {x}
                    {proc {y}
                      {let {hold x}
                        {let {dummy1 {:= x y}}
                          {let {dummy2 {:= y hold}}
                            0}}}}}}}
  {let {x 4}
    {let {y 5}
      {let {dummy {{swapper x} y}}
        y}}}})
```

This program will evaluate to 5, not 4.
To address this shortcoming, we define a new kind of procedure, called a **refproc**. The rest of **refproc** has the same syntax as **proc**. The argument to a **refproc** must *syntactically* be a variable (e.g., if f is bound to a **refproc**, the program $\{f\ 3\}$ is illegal). Invoking a **refproc** gives it the *address* of the argument variable; the **refproc** associates its locally-bound variable with this same address. As a result, all changes to the value of this variable inside the **refproc** reflect as changes in the caller. This makes it possible to write a generic swapper. For instance, the following program evaluates to 4:

```
{let {swapper {refproc {x}
                    {refproc {y}
                      {let {hold x}
                        {let {dummy1 {:= x y}}
                          {let {dummy2 {:= y hold}}
                            0}}}}}}}
  {let {x 4}
    {let {y 5}
      {let {dummy {{swapper x} y}}
        y}}}})
```

Starting with the interpreter on the next pages, clearly indicate all changes necessary to implement this version of Rip. (You may assume that all input programs are legal.) You can ignore the parser's code, to avoid that tedium.
Hint: Add a new type of closure value. In the main interpreter clause that handles applications, first check which kind of procedure you're applying, and handle **refproc**s appropriately. You may want to write a suitably altered version of *do-app* for **refproc**s.
FYI: This form of parameter passing is called *call-by-reference*; what we've seen in class thusfar is *call-by-value*.

```
(define-struct numE (num))
;; num : number

(define-struct addE (lhs rhs))
;; lhs : RP, rhs : RP

(define-struct varE (name))
;; name : symbol

(define-struct procE (arg-name body))
;; arg-name : symbol, body : RP

(define-struct appE (proc arg))
;; proc : RP, arg : RP

(define-struct setE (var-name val))
;; var-name : symbol, val : RP

(define-struct closure (proc dsub))
;; proc : procE, dsub : dsub

(define-struct binding (var-name val))
;; var-name : symbol, val : RP

(define-struct storage (locn value))
;; locn: number, value : RP

;; interp: RP × dsub × store → Value × store
(define (interp expr dsub store)
  (cond
    [(numE? expr) (values expr store)]
    [(addE? expr) (let-values ([(l-val l-store)
                                (interp (addE-lhs expr) dsub store)])
                    (let-values ([(r-val r-store)
                                  (interp (addE-rhs expr) dsub l-store)])
                      (values (numE+ l-val r-val)
                              r-store)))]
    [(varE? expr) (values (lookup-store
                           (lookup-dsub (varE-name expr) dsub) store)
                          store)]
    [(procE? expr) (values (make-closure expr dsub) store)]
    [(appE? expr) (let-values ([(p-val p-store)
                                (interp (appE-proc expr) dsub store)])
                    (let-values ([(a-val a-store)
                                  (interp (appE-arg expr) dsub p-store)])
                      (do-app p-val a-val a-store)))]
    [(setE? expr) (let-values ([(v-val v-store)
                                (interp (setE-val expr) dsub store)])
                    (values v-val
                            (update-store v-store
                                          (lookup-dsub (setE-var-name expr) dsub)
                                          v-val)))]
    [else (error 'interp "not a valid expression" expr)]))
```

[continued]

```
;; do-app: closure × RP × store → RP × store
(define (do-app clos argV store)
  (let-values ([(ext-env ext-store)
                (extend-env-and-store (closure-dsub clos)
                                      store
                                      (procE-arg-name (closure-proc clos))
                                      argV)])
    (interp (procE-body (closure-proc clos))
            ext-env
            ext-store)))

;; extend-env-and-store: dsub × store × symbol × RP → dsub × store
(define (extend-env-and-store env store var-name val)
  (let ([new-locn (next-store-locn)])
    (values
     (extend-dsub env var-name new-locn)
     (extend-store store new-locn val))))

;; next-store-locn: () → number
(define next-store-locn
  (let ([last -1])
    (lambda ()
      (begin
        (set! last (add1 last))
        last))))
```

Eliding code for:

- numE+: numE × numE → numE
- empty-dsub: dsub
- extend-dsub: dsub × symbol × RP → dsub
- lookup-dsub: symbol × dsub → RP
- empty-store: store
- extend-store: store × number × RP → store
- lookup-store: number × store → RP
- update-store: store × number × RP → store