COMPUTER SCIENCE 320
CONCEPTS OF PROGRAMMING LANGUAGES

Solutions for Mid-Term Examination

FRIDAY, MARCH 3, 2006

Problem 1. [25 pts.] A *special form* is an expression that is not evaluated according to the usual rules in Scheme. *Syntactic sugar* is an expression that is not part of the language, but is transformed before evaluation into something that is a legal expression. For each of the following, state whether it is a special form, or syntatic sugar, or neither. If it is a special form, state why it must be a special form. If it is sytatic sugar, state what it is transformed to before evaluation.

```
   (define foo 17)
   Special Form, because ‘‘foo’’ is not evaluated.
   (car ’(1 b 3))
   Neither.
   (lambda (x) (+ 3 x))
   Special Form, because ‘‘(x)’’ is not evaluated.
   (let ((y 10)) (* 5 y))
   Syntactic Sugar.  Becomes:  ((lambda (y) (* 5 y)) 10)
   (if (= a b) (/ 1 0) (+ 1 0))
   Special Form, because (/ 1 0) is not evaluated (in this case).
   (letrec ((a 1) (b 2)) (* a b))
   Special Form, because the definitions of a and b are in the enclosing en-
vironment.
   (mapcar ’car ’((1 3) (4 5)))
   Neither.
   (define (sum a b) (+ a b))
   Syntactic Sugar.  Becomes:  (define sum (lambda (a b) (+ a b)))
   (+ a 12)
   Neither
   (cond ((= a 1) (foo)) ((= a 0) 20) (bar))
   Special Form, because some expressions may not be evaluated.
```

Problem 2. **(a)** [11 points.] An *accumulator* is a procedure that is called repeatedly with a single numeric argument and accumulates its arguments into a sum. Each time it is called, it returns the currently accumulated sum. Write a procedure make-accumulator that generates accumulators, each maintaining an independent sum. The input to make-accumulator should specify the initial value of the sum. For example:

```
(define A (make-accumulator 5))

(A 10)
15

(A 10)
25

 (define (make-accumulator initial-value)
        (let ((sum initial-value))
              (set! sum (+ sum addend))
               sum))
```

or equivalent de-sugared expressions. **(b)** [11 points.] Now modify make-accumulator so that if the accumulator is called with the special value 'reset, it resets the sum to the original value. For example:

```
(define A (make-accumulator 5))

(A 10)
15

(A 10)
25

(A 'reset)
5


 (define (make-accumulator initial-value)
        (let ((sum initial-value)
              (iv initial-value))
        (lambda (addend)
                (if (equal? addend 'reset)
                   (set! sum iv)
                   (set! sum (+ sum addend)))
                 sum)))
```

or equivalent de-sugared expressions. **(c)** [3 points] Explain why the example given above of how A works tells you that A cannot be a functional program.

Problem 3. A *ring* is a list in which the tail pointer of the last element points to the first element of the list. For example:

myring
a
friend
of

(a) [4 points] Assume `myring` is defined to be the above data structure. What happens if you type ”myring” to the scheme interpreter?

prints out a friend of a friend of a friend of a friend of ... forever

(b) [7 points] Write a procedure `make-ring!` that takes a list and makes it a ring (without using cons or append). For example,

```
(define myring (make-ring! ’(a friend of)))
```

yields the ring above.

```
(define (helper-fn list head)
        (if (= 1 (length list))
            (set-cdr! list head)
            (helper-fn (cdr list) head)))

(define (make-ring! list)
        (helper-fn list list)
        list)
```

(c) [7 points] Write a procedure that prints out each element of the ring only once (using display) in forward order. For example:

```
(display-ring myring)
a
friend
of
```

(hint: `eq?` tests if two pointers are the same.)

```
(define (display-ring ring)

  (define (helper-fn2 head ring)
   (begin
      (display (car ring))
      (display " ")
      (if (not (eq? head (cdr ring)))
          (helper-fn2 head (cdr ring))
          ’())))

   (helper-fn2 ring ring))
```

(d) [7 points] Write a procedure that deletes the first occurrence of an item from the ring, leaving it a ring, using list mutator functions such as `set-car!` and `set-cdr!`. You can assume that at least one instance of the item is in fact in the ring. For example,

```
(set! myring (delete-ring! ’a myring))

(display-ring myring)
friend of
```

```
(define (delete-ring! item ring)
      (if (equal? item (car (cdr ring)))
          (begin (set-cdr! ring (cdr (cdr ring))) (cdr ring))
          (delete-ring! item (cdr ring))))
```

Problem 4. We have studied the higher order function `accumulate`. For example

```
(accumulate + 0 list)
```

returns the sum of the items in `list` and

```
(accumulate append ’() ’((1 2) (3 4) (5 6))
```

returns `(1 2 3 4 5 6)`

(a) [15 points] Write accumulate as a recursive function. It doesn’t matter what order you use to perform the accumulation.

```
 (define (accumulate fn initial-value inputlist)
        (if (null? inputlist)
            initial-value
            (fn (car inputlist) (accumulate fn initial-value (cdr inputlist)))))
```

(b) [10 points] Write accumulate as an iterative (imperative) function. It doesn’t matter what order you use to perform the accumulation.

```
 (define (accumulate fn initial-value inputlist)
        (let ((list inputlist)
              (result initial-value))
         (while ((not (null? list)))
                (begin (set! result (fn result (car lst))
                       (set! list (cdr list)))))
         result))
```

Extra Credit [5 points] Yesterday the ACM announced that Peter Naur is the Winner of this year's Turing Award, the Nobel Prize of Computer Science (no kidding!). What is Peter Naur most famous for? (hint: along with John Backus).

Algol-60 and/or Backus-Naur Form (BNF)