Assignment 5: Tail Recursion — CS351—Fall 2008

Due 23:59 Sun 23-Nov-2008. Email *one text file* containing *all* your solutions to: `barak+cs351-hw5@cs.nuim.ie`. (This file should be loadable into Scheme, meaning essay question answers should be in comments.)

1. Mechanically (as in the example done in class) translate the following imperative program with assignment into tail-recursive Scheme code. Show the stages of the transformation.

```
procedure gcd(x, y)
  while (x != y) {
     if (x > y) {
       val temp;
       temp = x;
       x = y;
       y = temp;
     }
     y = modulo(y, x);
  }
  return x;
end
```

Solution: First add labels, annotated with live variables:

```
procedure gcd(x, y)
t0: (x y)
  while (x != y) {
t1: (x y)
     if (x > y) {
t2: (x y)
       val temp;
       temp = x;
t3: (x y temp)
       x = y;
t4: (x y temp)
       y = temp;
     }
t5: (x y)
     y = modulo(y, x);
  }
t6: (x)
  return x;
end
```

Then we can convert,

```
(define gcd
  (lambda (x y)

    ;; t0: (x y)
    ;;   while (x != y) {
    (define t0
      (lambda (x y)
        (if (not (= x y))
            (t1 x y)
            (t6 x))))

    ;; t1: (x y)
    ;;      if (x > y) {
    (define t1
      (lambda (x y)
        (if (> x y)
            (t2 x y)
            (t5 x y))))

    ;; t2: (x y)
    ;;        val temp;
    ;;        temp = x;
    (define t2
      (lambda (x y)
        (t3 x y x)))

    ;; t3: (x y temp)
    ;;        x = y;
    (define t3
      (lambda (x y temp)
        (t4 x x temp)))

    ;; t4: (x y temp)
    ;;        y = temp;
    ;;      }
    (define t4
      (lambda (x y temp)
        (t5 x temp)))

    ;; t5: (x y)
    ;;      y = modulo(y, x);
    ;;   }
    (define t5
      (lambda (x y)
        (t0 x (modulo y x))))

    ;; t6: (x)
    ;;   return x;
    (define t6
      (lambda (x)
        x))

    (t0 x y)))
```

2. Consider this definition:

```
(define fibb
  (lambda (n)
    (if (= n 0)
            1
            (if (= n 1)
                1
                (+ (fibb (- n 1))
                   (fibb (- n 2)))))))
```

(a) Is this definition tail recursive?

Solution: No: each recursive call to `fibb` is a non-tail-call, since the results must be added.

(b) Translate `fibb` into CPS.

Solution:

```
(define cfibb
  (lambda (k n)
    (c= (lambda (n=0)
          (if n=0
              (k 1)
              (c= (lambda (n=1)
                    (if n=1
                        (k 1)
                        (c- (lambda (nm1)
                              (cfibb (lambda (fnm1)
                                       (c- (lambda (nm2)
                                             (cfibb (lambda (fnm2)
                                                      (c+ fnm1 fnm2))
                                                    nm2))
                                           n 2))
                                     nm1))
                            n 1)))
                  n 1)))))
        n 0)))
```

3. Write a tail recursive definition of `my-reverse` which is functionally identical to the predefined function `reverse`.

Solution:

```
(define aux
  (lambda (x a)
    (if (null? x)
        a
        (aux (cdr x) (cons (car x) a)))))


(define my-reverse
  (lambda (x)
    (aux x ’())))
```

4. Define `calls-non-tr?` which takes two arguments: an s-expression representing a fragment of Scheme code and the name of a procedure, and returns true iff that fragment of code calls the given procedure in a non-tail-recursive fashion. The fragment of code is constrained to the following subset of Scheme, where s denotes an expression in this Scheme subset, p denotes a symbol representing a procedure name, v denotes a symbol representing a variable, and n denotes a number,

$$s ::= \texttt{(}\,p\;s\ldots\,\texttt{)} \mid n \mid \texttt{(if}\;s\;s\;s\,\texttt{)} \mid v$$

Sample expressions in this Scheme subset:

```
foo
+
(foo (+ 1 (car x y (car)) cons 32))
(if (a (b c) 3) d (e (f g h)))
```

Examples:

```
(calls-non-tr? 'f 12) ⇒ #f
(calls-non-tr? 'f '(f f)) ⇒ #f
(calls-non-tr? 'f '(a (f 12))) ⇒ #t
(calls-non-tr? 'f '(f (f 12))) ⇒ #t
(calls-non-tr? 'a '(a (f 12))) ⇒ #f
(calls-non-tr? 'f '(if (a) (b) (c))) ⇒ #f
(calls-non-tr? 'f '(if (f) (b) (c))) ⇒ #t
(calls-non-tr? 'f '(if (a) (f) (f))) ⇒ #f
(calls-non-tr? 'f '(if (a) f f)) ⇒ #f
(calls-non-tr? 'f '(if (a) (car f) f)) ⇒ #f
(calls-non-tr? 'f '(if (a) (car (f g)) c)) ⇒ #t
(calls-non-tr? 'f '(if (a) c (car (f g)))) ⇒ #t
```

Solution:

```
(define calls-non-tr?
  (lambda (f x)
    (and (pair? x)
         (if (eq? (car x) 'if)
             (or (calls? f (cadr x))
                 (calls-non-tr? f (caddr x))
                 (calls-non-tr? f (cadddr x)))
             (any? (lambda (x) (calls? f x))
                   (cdr x))))))

(define calls?
  (lambda (f x)
    (and (pair? x)
         (if (eq? (car x) 'if)
             (or (calls? f (cadr x))
                 (calls? f (caddr x))
                 (calls? f (cadddr x)))
             (any? (lambda (x) (calls? f x))
                   x)))))
```

```
(define any?
  (lambda (p? lis)
    (and (not (null? lis))
         (or (p? (car lis))
             (any? p? (cdr lis))))))
```

5. *(Optional)* If you encountered any problems with the assignment, or have any comments on it, or other comments or suggestions, I would appreciate hearing them. As practice for actual work, where weekly reports are not unusual, please embody these in a brief report.

Solution:

This is the best class ever. My only suggestion: longer harder assignments. And more of them!

Honor Code: You may discuss these with others, but please write your answers by yourself and without reference to communal notes. In other words, your answers should be *from your own head.*