CS 61A Structure and Interpretation of Computer Programs
Spring 2017

Discussion Quiz 9 Solutions

1. (3 points) Pin the Tail

Identify whether or not each of the following procedures uses a constant amount of space in a tail-recursive Scheme implementation (i.e. whether **every** recursive call is a tail call).

```
(define (copy lst result)
    (if (null? lst) result
        ((lambda (copy) copy) (copy (cdr lst)
                                    (append result (list (car lst)))))))
```

(Remember that `append` takes zero or more lists and constructs a new list with all of the lists' elements.)

`copy` is *not* tail-recursive. After the recursive call returns, we still have to apply a `lambda` procedure.

```
(define (broken lst) (broken (broken lst)))
```

`broken` is *not* tail-recursive. One of the recursive calls is not a tail call.

```
(define (is-ascending lst last-num)
    (if (null? lst) #t
        (and (is-ascending (cdr lst) (car lst)) (> (car lst) last-num))))
```

(Assume that this procedure is always called with a `last-num` that is less than all of the elements in the list.)

`is-ascending` is *not* tail-recursive. The recursive call isn't even in a tail context!

2. (4 points) Hail Recursion

Write a *tail-recursive* version of `hailstone`. This procedure accepts a positive integer `n` and returns a list that contains the hailstone sequence starting at `n`. For instance, `(hailstone 5)` would return `(5 16 8 4 2 1)`.

```
(define (hailstone n)
  (define (hs-helper n lst)
    (cond ((= n 1) (append lst (list 1)))
          ((even? n) (hs-helper (/ n 2) (append lst (list n))))
          (else (hs-helper (+ 1 (* 3 n)) (append lst (list n))))))
  (hs-helper n nil))
```

3. (3 points) Humans Need Not Apply

What does `eval` do, in the context of an interpreter? What does `apply` do?

`eval` parses expressions (all kinds of expressions; `eval` doesn't discriminate!), *evaluating* an expression to determine its value. When `eval` is passed a call expression, it evaluates the operator and operands and then hands them off to `apply`, which performs the body of the function on the evaluated operands.

`eval` and `apply` are mutually recursive. Whenever `eval` encounters a function call, it sends the expression to `apply` to do the actual calling. In turn, `apply` uses `eval` while processing function bodies.

For reference, this is only an intermediate stage (specifically the evaluation bit) of the interpreter's read-eval-print loop (REPL). The preceding stage, "read", takes input code and tokenizes it before converting it into the data structures (e.g. `Pairs` or `Exprs`) used during "eval". The following stage, "print", displays the expression's value for the user to see.