TAIL RECURSION AND INTERPRETERS

COMPUTER SCIENCE MENTORS 61A

April 9 to April 11, 2018

1 Tail Recursion

1. What is a tail context? What is a tail call? What is a tail recursive function?

> **Solution:** A tail call is a call expression in a tail context. A tail context is usually the final action of a procedure/function.
> A tail recursive function is where all the recursive calls of the function are in tail contexts.
>
> An ordinary recursive function is like building up a long chain of domino pieces, then knocking down the last one. A tai l recursive function is like putting a domino piece up, knocking it down, putting a domino piece up again, knocking it d own again, and so on. This metaphor helps explain why tail calls can be done in constant space, whereas ordinary recursi ve calls need space linear to the number of frames (in the metaphor, domino pieces are equivalent to frames).

2. Why are tail calls useful for recursive functions?

> **Solution:** When a function is tail recursive, it can effectively discard all the past recursive frames and only keep the current fr ame in memory. This means we can use a constant amount of memory with recursion, and that we can deal with an unbounded number of tail calls with our Scheme interpreter.

3. Consider the following function:

```
(define (count-instance lst x)
  (cond ((null? lst) 0)
        ((equal? (car lst) x) (+ 1 (count-instance
                                    (cdr lst) x)))
        (else (count-instance (cdr lst) x))))
```

Why is count-instance not a tail call? Optional: draw out the environment diagram of this sum-list with lst (1 2 1) with x = 1.

Solution: In the second case in the cond special form, evaluating the recursive call to count-instance is not the last thing we do (we have to add 1 afterwards).

4. Rewrite count-instance in a tail recursive context.

```
(define (count-tail lst x)




)
```

Solution:

```
(define (count-tail lst x)
    (define (count-helper lst x instances)
                    (cond ((null? lst) instances)
              ((equal? (car lst) x) (count-helper (cdr
                 lst) x (+ instances 1)))
              (else (count-helper (cdr lst) x instances))))
    (count-helper lst x 0))
```

5. Implement `filter`, which takes in a one-argument function `f` and a list `lst`, and returns a new list containing only the elements in `lst` for which `f` returns true. Your function must be tail recursive.

   You may wish to use the built-in append function, which takes in two lists and returns a new list containing the elements of the first list followed by the elements of the second.

```
(define (filter f lst)




)
```

Solution:

```
(define (filter f lst)
  (define (filter-tail f lst so-far)
     (cond ((null? lst) so-far)
           ((f (car lst)) (filter-tail f (cdr lst)
                          (append so-far (list (car
                            lst)))))
           (else (filter-tail f (cdr lst) so-far))))
  (filter-tail f lst nil))
```

2 Interpreters

1. Circle the number of calls to `scheme_eval` and `scheme_apply` for the code below.

```
(+ 1 2)
```

`scheme_eval`	1	3	4	6
`scheme_apply`	1	2	3	4

Solution: 4 `scheme_eval`, 1 `scheme_apply`.

2. Circle the number of calls to `scheme_eval` and `scheme_apply` for the code below.

```
(if 1 (+ 2 3) (/ 1 0))
```

`scheme_eval`	1	3	4	6
`scheme_apply`	1	2	3	4

Solution: 6 `scheme_eval`, 1 `scheme_apply`.

```
(or #f (and (+ 1 2) 'apple) (- 5 2))
```

`scheme_eval`	6	8	9	10
`scheme_apply`	1	2	3	4

Solution: 8 `scheme_eval`, 1 `scheme_apply`.

```
(define (square x) (* x x))

(+ (square 3) (- 3 2))
```

`scheme_eval`	2	5	14	24
`scheme_apply`	1	2	3	4

Solution: 14 `scheme_eval`, 4 `scheme_apply`.

```
(define (add x y) (+ x y))

(add (- 5 3) (or 0 2))
```

`scheme_eval`	12	13	14	15
`scheme_apply`	1	2	3	4

Solution: 13 `scheme_eval`, 3 `scheme_apply`.