Tail Recursion, Scheme, Regular Expressions Solutions

CS 61A

August 4, 2021

1 Tail Recursion

Tail Recursion Overview

Often, when we write recursive functions, they can take up a lot of space by opening a bunch of frames. Think about `factorial(6)`. In order to solve it, we will have to open 6 frames. Now what if we tried `factorial(1000000)`? To avoid opening 1,000,000 frames, we can use a method called **tail recursion**. In tail call optimized languages, like Scheme (but not Python), tail recursive functions only use a **constant** amount of space. The key to defining a tail recursive function is to make sure no further calculations are done after the recursive call, so that none of the values in the current frame have to be saved. If we don't have to save any values in the current frame, we can close it as we make the next recursive call, ensuring that we only have one frame open.

In order to identify whether a function is tail recursive, first find the recursive call in your function. Then, check whether you return the exact result of your recursive call, or if you do work on the result. If you simply return the result of your recursive call, then your function is tail recursive! However, if you do additional work to the result of your recursive call, then it is not tail recursive. Additional work could be adding one to the result of your recursive call and returning the new value, or appending it to a list and returning the resulting list.

The general way we convert a recursive function to a tail recursive one is to move the calculation outside the recursive call into one of the recursive call arguments to accumulate the results. However, this is not always possible if our function doesn't have an argument that accumulates the results, so we may have to create a helper function with an accumulating argument and have the helper be a tail recursive function.

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

   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 a function where all its recursive calls are in tail contexts.

2. Why are tail calls useful for recursive functions?

   When a function is tail recursive, it can effectively discard all the past recursive frames and only keep the current frame 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))))
```

What is the purpose of `count-instance`? Is it tail recursive? Why or why not? Optional: draw out the environment diagram of this `count-instance` with `lst = (1 2 1)` and `x = 1`.

`count-instance` returns the number of times `x` appears in `lst`.
It is not tail recursive. The call to `count-instance` is an arguments to a function call, so it will not be the final thing we do in every frame (we will have to apply + after evaluating it.)

4. Rewrite count-instance to be tail recursive. (Hint: helper functions are often useful in implementing Tail Recursion.)

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









)
```

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

```
;Doctests
scm> (filter (lambda (x) (> x 2)) '(1 2 3 4 5))
(3 4 5)

(define (filter f lst)









)
```

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

2 Scheme

1. Suppose Isabelle bought turnips from the Stalk Market and has stored them in random amounts among an ordered sequence of boxes. By the magic of time travel, Isabelle's friend Tom Nook can fast-forward one week into the future and determine exactly how many of Isabelle's turnips will rot over the week and have to be discarded.

   Assuming that boxes of turnips will rot in order, i.e. all of box 1's turnips will rot before any of box 2's turnips, help Isabelle determine which turnips will still be fresh by week's end. Specifically, fill in `decay`, which takes in a list of positive integers `boxes`, which represents how many turnips are in each box, and a positive integer `rotten` representing the number of turnips that will rot, and returns a list of non-negative integers that represents how many fresh turnips will remain in each box.

```
; doctests
scm> (define a '(1 6 3 4))
a
scm> (decay a 1)
(0 6 3 4)
scm> (decay a 5)
(0 2 3 4)
scm> (decay a 9)
(0 0 1 4)
scm> (decay a 1000)
(0 0 0 0)

(define (decay boxes rotten)









)
```

```
(define (decay boxes rotten)
    (cond
        ((null? boxes) nil)
        ((< rotten (car boxes)) (cons (- (car boxes) rotten)
           (cdr boxes)))
        (else (cons 0 (decay (cdr boxes) (- rotten (car
           boxes)))))
    )
)
```

2. (a) Define `append`, which takes in two lists and returns a new list with all the elements of the first list followed by all the elements of the second. Do not use the built-in `append` function.

```
> (append '(1 2 3) '(4 5 6))
(1 2 3 4 5 6)

(define (append lst1 lst2)









)
```

```
(define (append lst1 lst2)
    (if (null? lst1) lst2
        (cons (car lst1) (append (cdr lst1) lst2)))))
```

(b) Define `reverse`. Hint: use `append`.

```
> (reverse '(1 2 3))
(3 2 1)

(define (reverse lst)









)
```

```
(define (reverse lst)
    (if (null? lst) lst
        (append (reverse (cdr lst)) (list (car lst)))))
```

(c) Define `reverse` tail-recursively. Hint: use a helper function and `cons`.

```
(define (reverse lst)









)
```

```
(define (reverse lst)
    (define (helper lst reversed)
        (if (null? lst) reversed
            (helper (cdr lst) (cons (car lst) reversed ))))
    (helper lst '()))
```

3 Scheme Challenge

1. Finish the functions `max` and `max-depth`. `max` takes in two numbers and returns the larger. Function `max-depth` takes in a list `lst` and returns the maximum depth of the list. In a nested scheme list, we define the depth as the number of scheme lists a sublist is nested within. A scheme list with no nested lists has a `max-depth` of 0.

```
;doctests
scm> (max 1 5)
5
scm> (max-depth '(1 2 3))
0
scm> (max-depth '(1 2 (3 (4) 5)))
2
scm> (max-depth '(0 (1 (2 (3 (4) 5) 6) 7))
4

(define (max x y) _____)

(define (max-depth lst)
    (define (helper lst curr)
        (cond
            ((_____) _____)
            ((_____) (max _____
                          _____))
            (else (helper _____))
        )
    )
    (_____)
)
```

```
(define (max x y) (if (> x y) x y))

(define (max-depth lst)
    (define (helper lst curr)
            (cond
              ((null? lst) curr)
              ((pair? (car lst)) (max (helper (car lst)
                                              (+ 1 curr))
                                      (helper (cdr lst) curr)))
              (else (helper (cdr lst) curr))
            )
      )
    (helper lst 0)
)
```

4 Regular Expressions (Optional)

Note: This problem is provided as extra practice, and most likely will not be covered in tutorial.

1. We are given a linear equation of the form $mx + b$, and we want to extract the m and b values. Remember that '.' and '+' are special meta-characters in Regex.

   This problem is written by Kunal Agarwal

```
import re
def linear_functions(eq_str):
    """
    Given the equation in the form of 'mx + b', returns a
       tuple of m and b values.
    >>> linear_functions("1x+0")
    [('1', '0')]
    >>> linear_functions("100y+44")
    [('100', '44')]
    >>> linear_functions("99.9z+23")
    [('99.9', '23')]
    >>> linear_functions("55t+0.4")
    [('55', '0.4')]
    """
    return re.findall(r"__________", eq_str)
```

```
r'(\d*\.?\d+)\w\+?(\d*\.?\d+)'
```