Writing Tail-Recursive Functions

Tail-recursive functions directly return the value of their recursive call. This worksheet will present a way of writing and re-writing recursive functions so that they are tail-recursive. Examples will be given in both Scheme and Python in order to aid understanding, however you should keep in mind that only Scheme has tail-recursion capabilities.

Difficulties in Writing Tail-Recursive Functions

In a tail-recursive function, we cannot modify the value of the recursive call before returning it. We must return it directly.

```scheme
(define (sum lst)
  ; Sums a list of numbers.
  (if (null? lst)
      0
      (+
       (car lst)
       (sum (cdr lst)))
  )
)
```

This function is not tail-recursive because after getting the value of the recursive call, it adds the first element of the list to it.

```python
def sum(lst):
    """Sums a list of numbers."""
    if len(lst) == 0:
        return 0
    return lst[0] + sum(lst[1:])
```

This is what the function would look like if it were tail-recursive. But how to write it in this format?

Pass-Up and Pass-Down Recursion

In order to see how to write sum in a tail-recursive way, we will take a detour to look at two general styles of writing recursive functions, “pass-up” and “pass-down” style. Pass-up style is often more natural and obvious, but it is pass-down style that enables tail-recursion.
(define (sum lst)
    ; Sums a list of numbers.
    (if (null? lst)
        0
        (+
            (car lst)
            (sum (cdr lst))
        )
    )
)

The version of sum we saw above was written in pass-up style. In pass-up recursion, partial solutions flow up the recursive call chain, through the return values. No useful computation occurs on the way down the recursive call chain. sum([1, 2, 3]) immediately calls sum([2, 3]) without doing anything; sum([2, 3]) then calls sum([3]), and so on. The useful computation occurs as we return back up the recursive call chain. sum([]) returns 0 to sum([3]). sum([3]) takes that 0 and returns 3 + 0 = 3 to sum([2, 3]). sum([2, 3]) takes that 3 and so on.
(define (sum lst sumsofar)
  ; Sums a list of numbers.
  (if (null? lst)
      sumsofar
      (sum
       (cdr lst)
       (+ (car lst) sumsofar)
      )
    )
  )

def sum(lst, sumsofar):
    """Sums a list of numbers."""
    if len(lst) == 0:
        return sumsofar
    newsum = lst[0] + sumsofar
    return sum(lst[1:], newsum)

This version of sum is written in pass-down style. In pass-down recursion, partial solutions flow down the recursive call chain, through the arguments. All the useful computation occurs on the way down the recursive call chain; the return values simply pass the final answer all the way back up. sum([1, 2, 3], 0) does 1+0=1 and returns sum([2, 3], 1). sum([2, 3], 1) does 2+1=3 and returns sum([3], 3).

Pass-Down Recursion Enables Tail-Recursiveness

In pass-down recursion, the value of the recursive call is returned directly, which makes it possible to write a tail-recursive function. The pass-down version of sum above is tail-recursive.

Pass-Down Recursion May Require Adding an Argument

To write a recursive function in the pass-down style, you may need to add an argument that represents the “result so far”. If writing a function that sums a list of
numbers, the “result so far” argument will represent the sum of the previous numbers in the list. If writing a function that finds the maximum of a list of numbers, the “result so far” argument will represent the maximum of the previous numbers in the list.

Pass-Down Recursion May Require Using a Helper Function

Our added “result so far” argument is a bit annoying for people who actually use our function; they always must pass an initial value for that additional argument. For example, notice how when we rewrote sum in pass-down style, it had to be called as (sum ‘(1 2 3) 0) instead of (sum ‘(1 2 3)). We can remove this annoying additional argument using a helper function. We will add the additional argument to the helper function instead of the original function. The original function’s job will simply be to call the helper function.

```
(define (sum lst)
  ; Sums a list of numbers.
  (define (helper lst sumsofar)
    (if (null? lst)
        sumsofar
        (sum
         (cdr lst)
         (+ (car lst) sumsofar)
        )
    )
  )
  (helper lst 0)
)

def sum(lst):
    """Sums a list of numbers."""
    def helper(lst, sumsofar):
        if len(lst) == 0:
            return sumsofar
        newsum = lst[0] + sumsofar
        return sum(lst[1:], newsum)
    return helper(lst, 0)
```
Practice Problems

Reverse List

Write a tail-recursive function that reverses `lst`.

```lisp
(define (reverse lst)
  (define (f l r)
    (if __________
        __________
        __________
      __________
    )
    __________
  )
  __________
)
```

Raise Number to Power

Write a tail-recursive function that raises `b` to the `n`-th power using multiplication.

```lisp
(define (power b n)
  (define (f p k)
    (if __________
        __________
        __________
      __________
    )
    __________
  )
)
```

Hailstone

Write a tail-recursive function that finds the length of the hailstone sequence that starts with `n`.

```lisp
(define (hailstone n)
  (define (f k i)
    (if __________
        __________
        __________
        __________
        __________
      __________
    )
    __________
  )
)
```
Solutions

Reverse List

(define (reverse lst)
  (define (f l r)
    (if (null? l)
        r
        (f (cdr l) (cons (car l) r)))
  )
  (f lst nil)
)

Raise Number to Power

(define (power b n)
  (define (f p k)
    (if (= p 0)
        k
        (f (- p 1) (* k b)))
  )
  (f n 1)
)

Hailstone

(define (hailstone n)
  (define (f k i)
    (if (= k 1)
        i
        (if (= (modulo k 2) 0)
            (f (/ k 2) (+ i 1))
            (f (+ (* k 3) 1) (+ i 1)))
    )
  )
  (f n 1)
)