Let’s write a function that filters a list for positive numbers:

```
(define (fil-pos l)
  (cond
   [(empty? l) empty]
   [(cons? l)
    (cond
     [(positive? (first l))
      (cons (first l) (fil-pos (rest l)))]
     [else
      (fil-pos (rest l))])]
)
```

Now we write the standard CPS version:

```
(define (fil-pos/k l k)
  (cond
   [(empty? l) (k empty)]
   [(cons? l)
    (cond
     [(positive? (first l))
      (fil-pos/k (rest l)
        (lambda (v)
          (k (cons (first l) v)))]
     [else
      (fil-pos/k (rest l) k)])])
)
```

```
(define (fil-pos l)
  (fil-pos/k l (lambda (v) v))
)
```

Notice in this code how \( k \) is treated on the recursive call. If the first element in the list is positive, an augmented continuation is passed as \( k \) in the recursive call. If the number is non-positive, then the same \( k \) is simply passed back.

The latter type of recursive call is called a tail call. In other classes, you might have loosely defined a tail call as a function call which doesn’t have to remember anything
about the state at the time of the function. It turns out that CPS gives us a precise way
to define tail call. A tail call is a function call in which the continuation in the CPS
transformation of the program is not augmented.

Finally, we make the stack explicit and use a list of structures to represent it:

```scheme
(define-struct stk/mt ()
(define-struct stk/cons (fst))

(define (fil-pos/stk l stk)
  (cond
   [(empty? l) (Pop stk empty)]
   [(cons? l)
    (cond
     [(positive? (first l))
      (fil-pos/stk (rest l)
                   (Push stk (make-stk/cons (first l)))]
     [else
      (fil-pos/stk (rest l)
                   stk)])])

(define (Push stk frame)
  (cons frame stk))

(define (Pop stk v)
  (let ((1st (first stk)))
    (cond
     [(stk/mt? 1st) v]
     [(stk/cons? 1st)
      (Pop (rest stk)
           (cons (stk/cons-fst 1st) v))])

(define (fil-pos l)
  (fil-pos/stk l (list (make-stk/mt))))
```

We have now done something very interesting with our code. We have removed the
non-tail call in `fil-pos/stk`. Tail calls are nice, because in assembly code, they are just
jumps; we do not have to remember about the calling state when we do a jump.

Actually, the tail calls above aren’t exactly jumps—they still have arguments. But
arguments are easy to get rid of. We will define several registers which will represent
arguments and the stack:

```scheme
(define =r1= 'dummy)
(define =stk= 'dummy)
```

We now rewrite the above code to use the registers. When we make a tail call, we’ll
mutate the registers instead of passing arguments explicitly:
(define (fil-pos/reg)
  (cond
   [(empty? =r1=) (Pop =stk= empty)]
   [(cons? =r1=)
    (cond
     [(positive? (first =r1=))
      (begin
       (set! =stk= (Push =stk= (make-stk/cons (first =r1=))))
       (set! =r1= (rest =r1=))
       (fil-pos/reg))]
     [else
      (begin
       (set! =r1= (rest =r1=))
       (fil-pos/reg)))]
   )]
)

(define (Push stk frame)
  (cons frame stk))

(define (Pop stk v)
  (let ((1st (first stk)))
    (cond
     [(stk/mt? 1st) v]
     [(stk/cons? 1st)
      (Pop (rest stk)
        (cons (stk/cons-fst 1st) v))]])
)

(define (fil-pos l)
  (begin
   (set! =r1= l)
   (set! =stk= (list (make-stk/mt)))
   (fil-pos/reg)))