Recursion

January 24–25, 2012
CS 60: Principles of Computer Science

"Just pretend that the function you're writing already exists."
Geoff Romer (CS 60)

"To understand recursion, you must first understand recursion…"
Anonymous (CS 60)

Assignment 1 (Recursion and Lists) due Monday 1/30

List operations

```
(define A '(w o o))
(define B '(h o o))

(list A B)   ;; ===>

(cons A B)   ;; ===>

(append A B) ;; ===>
```

In the world of Big-O

- ✓ *Constant* factors are ignored
- ✓ Inputs are arbitrarily large (so "small" sums are ignored)
- ✓ We are looking for upper bounds.

$O(1)$

- 6 steps
- 1 (big?) step
- no more than 4000 steps
- somewhere between 2 and 47 steps

$O(n)$

- $100n + 3$ steps
- $n - 1$ (big?) steps
- anywhere between 3 and 69 steps per item, for n items.

$O(n^2)$

- $2n^2 + 100n + 3$ steps
- $n^2 - n$ (big?) steps
- somewhere between 1 and 40 steps per item, for n^2 items
- anywhere between $\log n$ and $7n$ steps per item, for n items.

Computing Length

```
; Given a list L, return its length.
; Already built-in as the function "length"
 (define (myLength L)
   (cond
       [ (null? L)  0 ]
       [ else       (+ 1 (myLength (rest L))) ]
   ))
```

Running time?

Let n be the length of the initial list and let $T(n)$ be the (big-O) steps for an input of size n.

- ✓ Then $T(0) = 1$; otherwise $T(n) = 1 + T(n-1)$.
- ✓ So, $T(n) = 1 + T(n-1) = 2 + T(n-2) = \cdots = n + T(0) = O(n)$.

Alternatively:

- ✓ We look at n elements of the list
- ✓ We do $O(1)$ work per item (testing for empty, adding one, etc.)
- ✓ $O(n) \times O(1) = O(n \times 1) = O(n)$.

Checking Membership in a List

```
; Given a value e and a list L, check if e is in L
; Already built-in as the function "member"
 (define (myMember e L)
   (cond
       [ (null? L)              #f ]
       [ (equal? e (first L))   #t ]
       [ else                   (myMember e (rest L)) ]))
```

Worst-case asymptotic running time?

$O(n)$, *where* n *is the length of the list. (Basically the same argument as for* `length`.)

Removing an Item from a List

```
(define (myRemove e L)
  (cond
    [ (null? L) '() ]
    [ (equal? e (first L))  (rest L) ]
    [ else  (cons (first L) (remove e (rest L))) ]
  )
)
```

Worst-case asymptotic running time?

$O(n)$, *where* n *is the length of the list. (Basically the same argument as for* `length` *and* `member`.)

Optimization?

```
(define (myRemove e L)
  (cond
    [ (not (member e L))    L ]          ;; <--- new line
    [ (null? L)             '() ]
    [ (equal? e (first L))  (rest L) ]
    [ else  (cons (first L) (remove e (rest L))) ]
  )
)
```

Worst-case asymptotic running time?

Let n be the length of the initial list, and let $T(n)$ be the (big-O) steps for an input of size n.

- ✓ $T(0) = 1$; otherwise $T(n) = O(n) + T(n-1)$.
- ✓ So, $T(n) = n + T(n-1) = n + (n-1) + T(n-2) = \cdots =$ $n + (n-1) + \cdots + 3 + 2 + 1 + T(0) = O(n^2)$.

"Quiz" NAME:

1. Define `(append L M)`, by recursion on `L`.
   (How is `[1,2,3]+[4,5,6]` related to `[2,3]+[4,5,6]`?)

2. Define `(reverse L)`, by recursion on `L`.
   (Hint: use **append**).

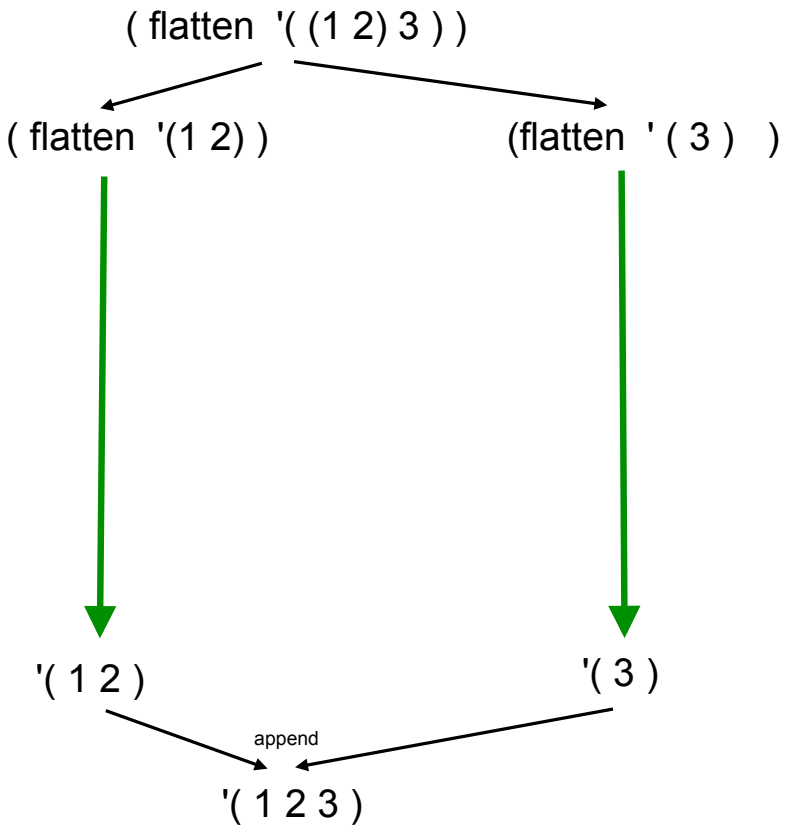
3. Define `(flatten X)`, which returns the list of items in `X` without any nesting, e.g.,

```
(flatten '(1 (2 3 (4)) (5 6))) ===> '(1 2 3 4 5 6)
(flatten 3)                    ===> '(3)
(flatten '())                  ===> '()
```

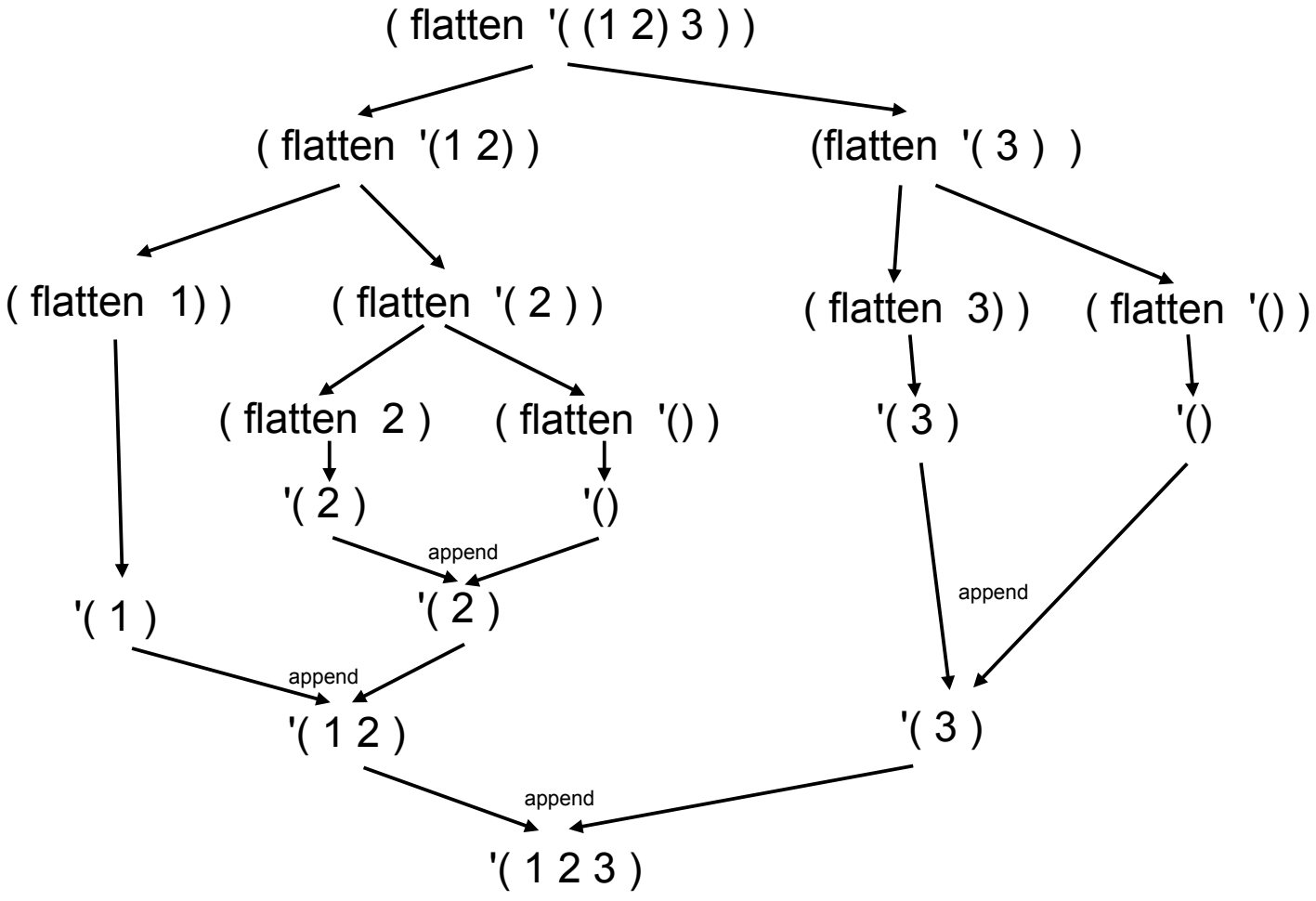
   (Hint: use `(list? X)` to check if `X` is a list.)

Extra Credit: What is the asymptotic running time of these functions?

Recursive Intuition



Recursive Reality



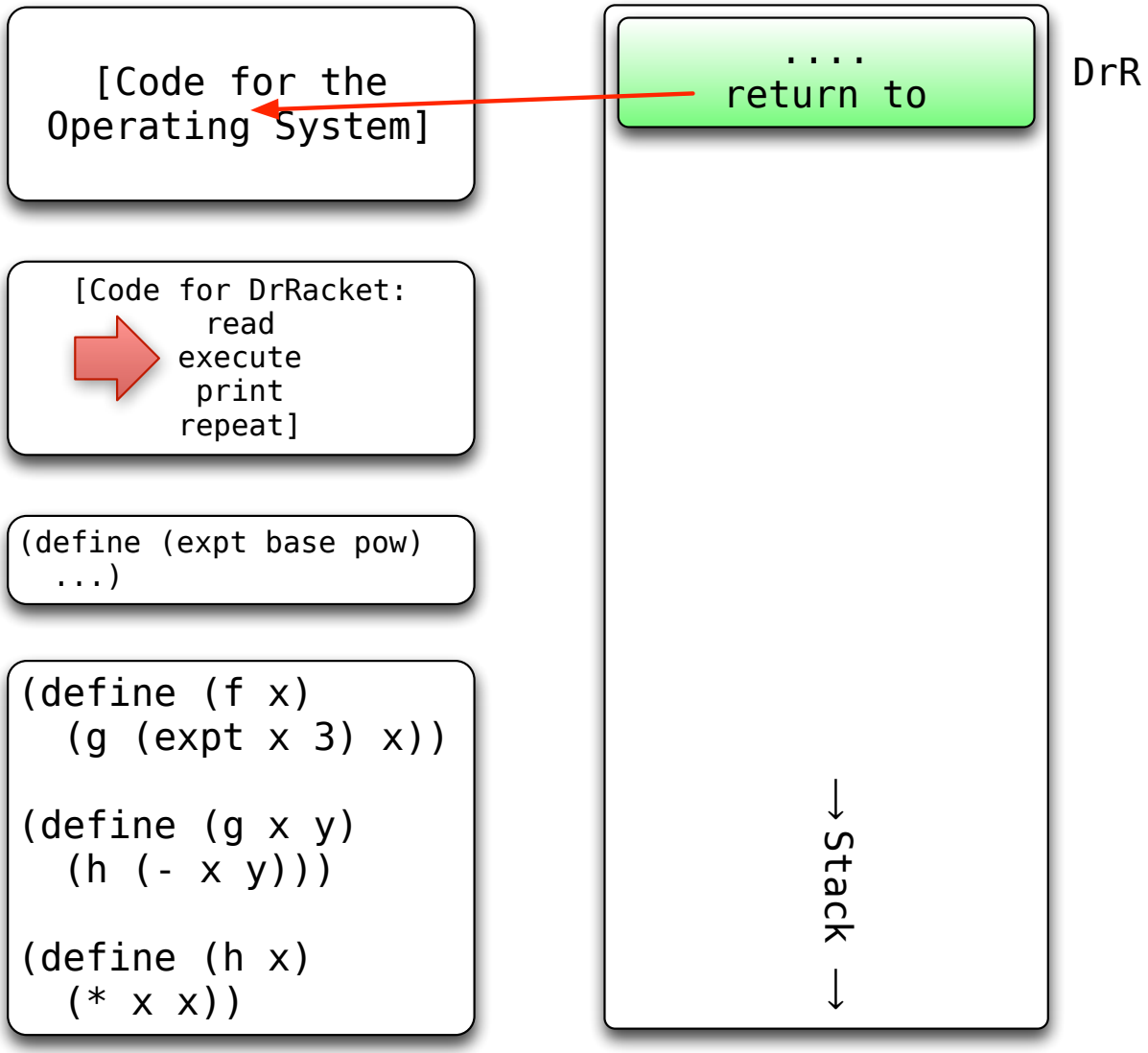
Functions on a Stack: `(f 4)`

```
(define (f x)
  (g (expt x 3) x))
```

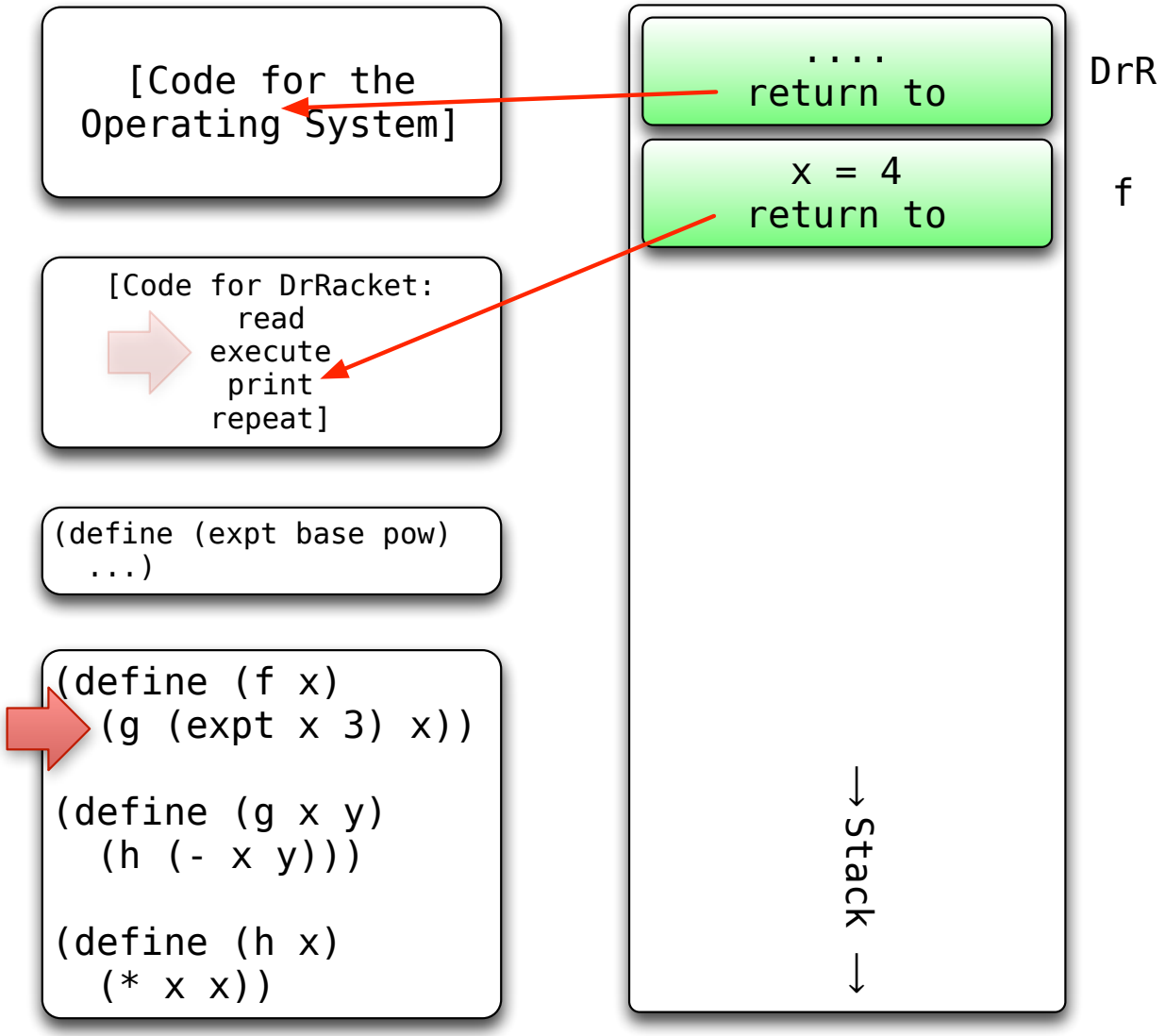
```
(define (g x y)
  (h (- x y)))
```

```
(define (h x)
  (* x x))
```

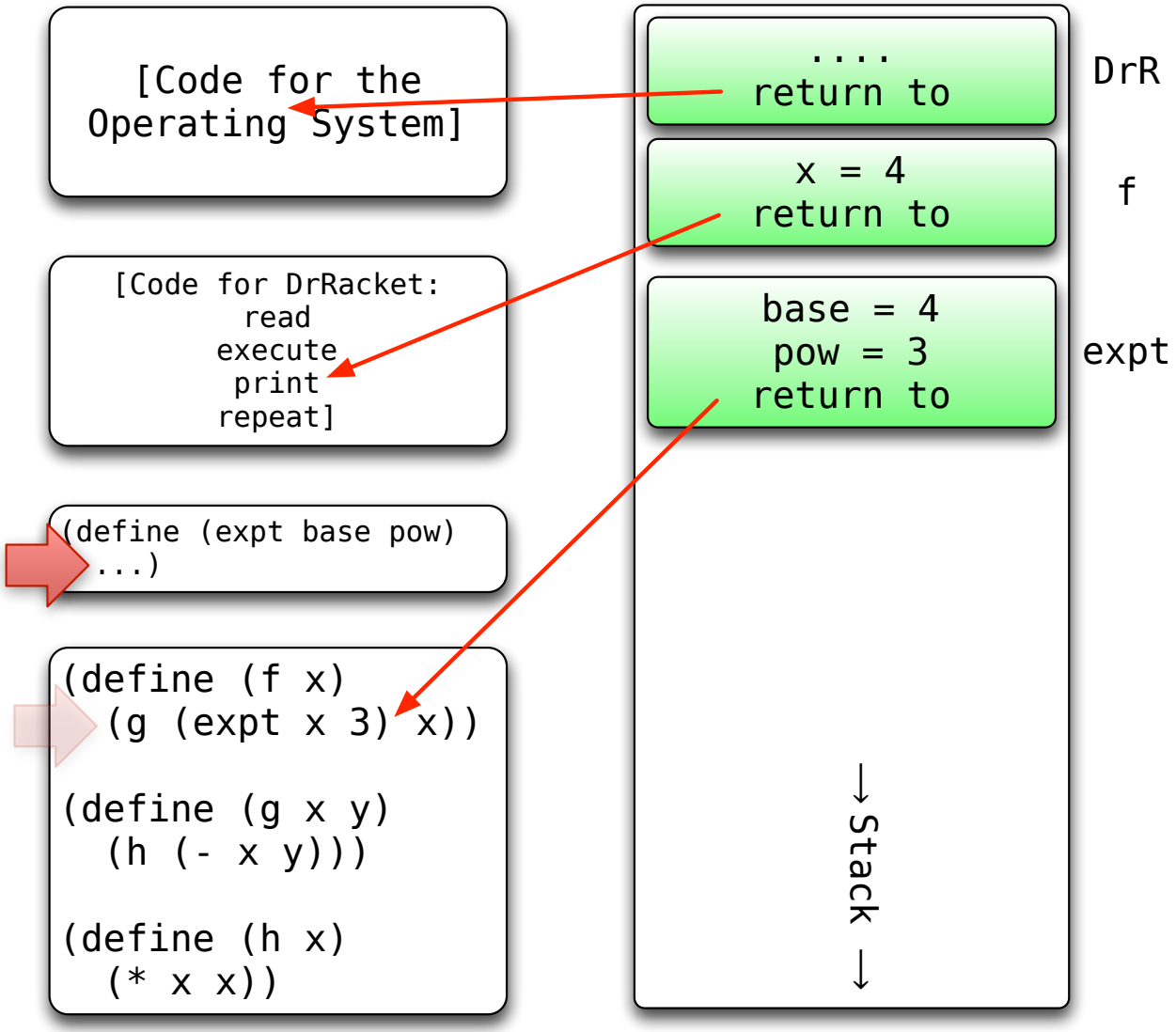
Executing Functions on a Stack: (f 4)



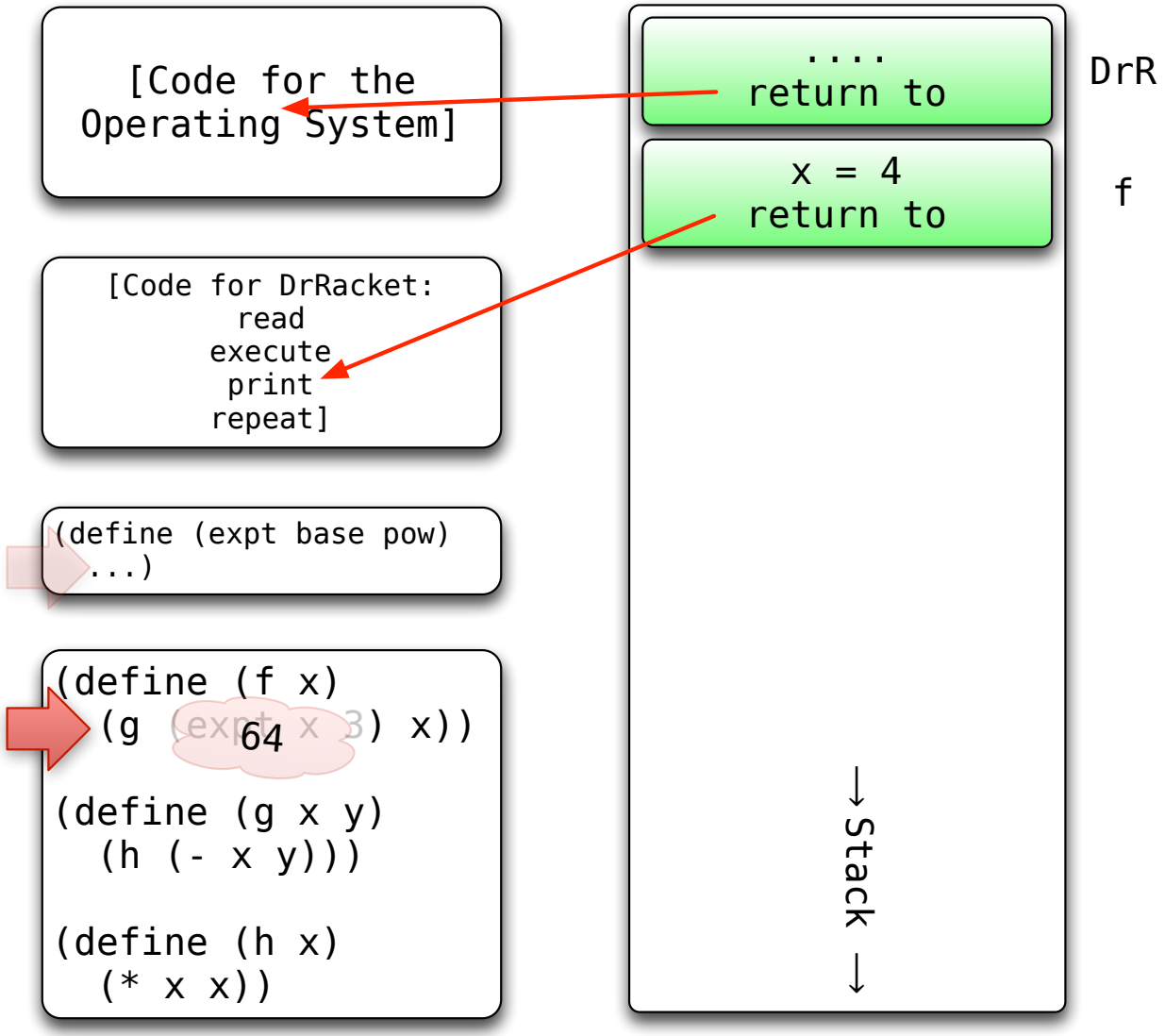
Executing Functions on a Stack: (f 4)



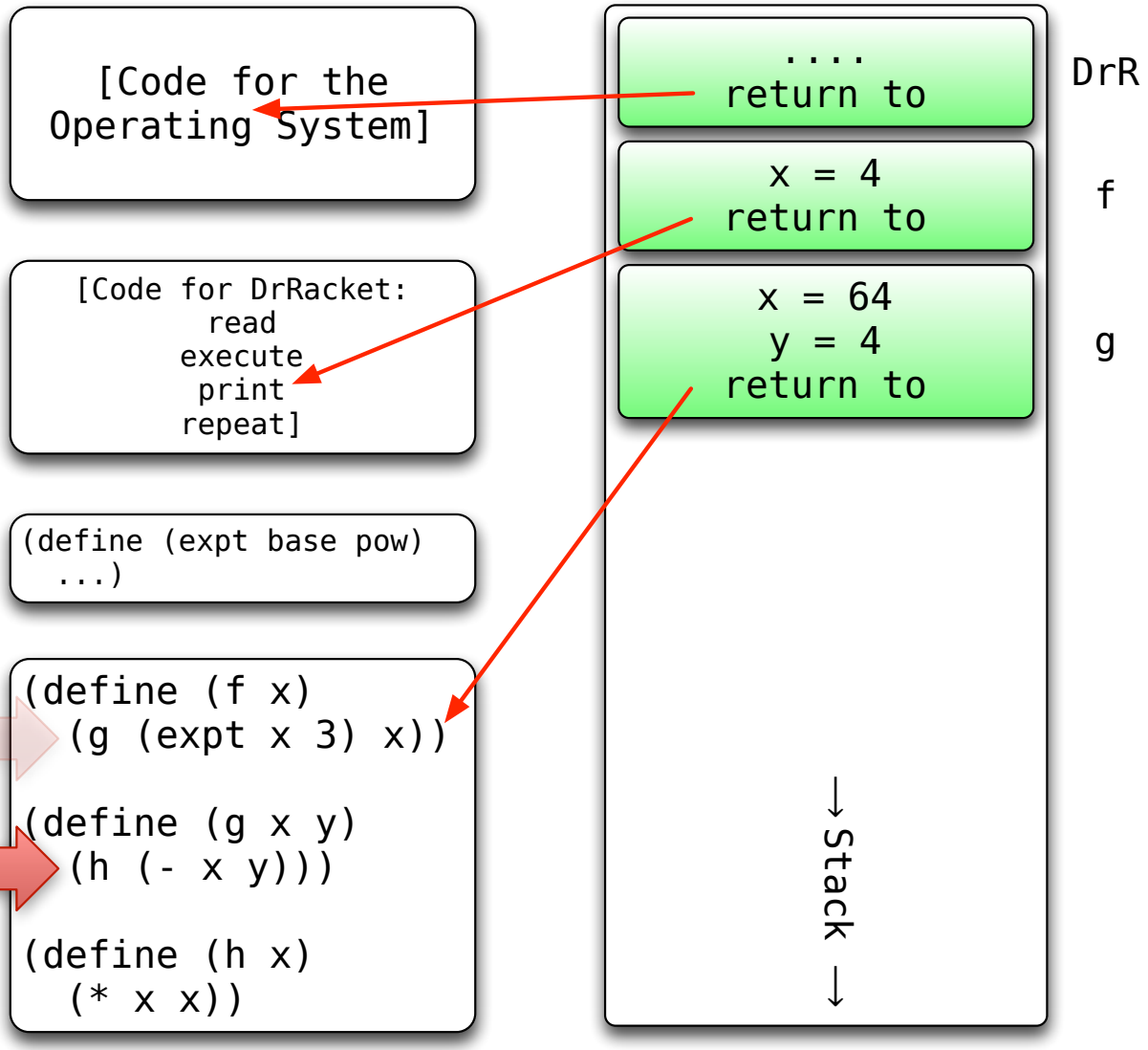
Executing Functions on a Stack: (f 4)



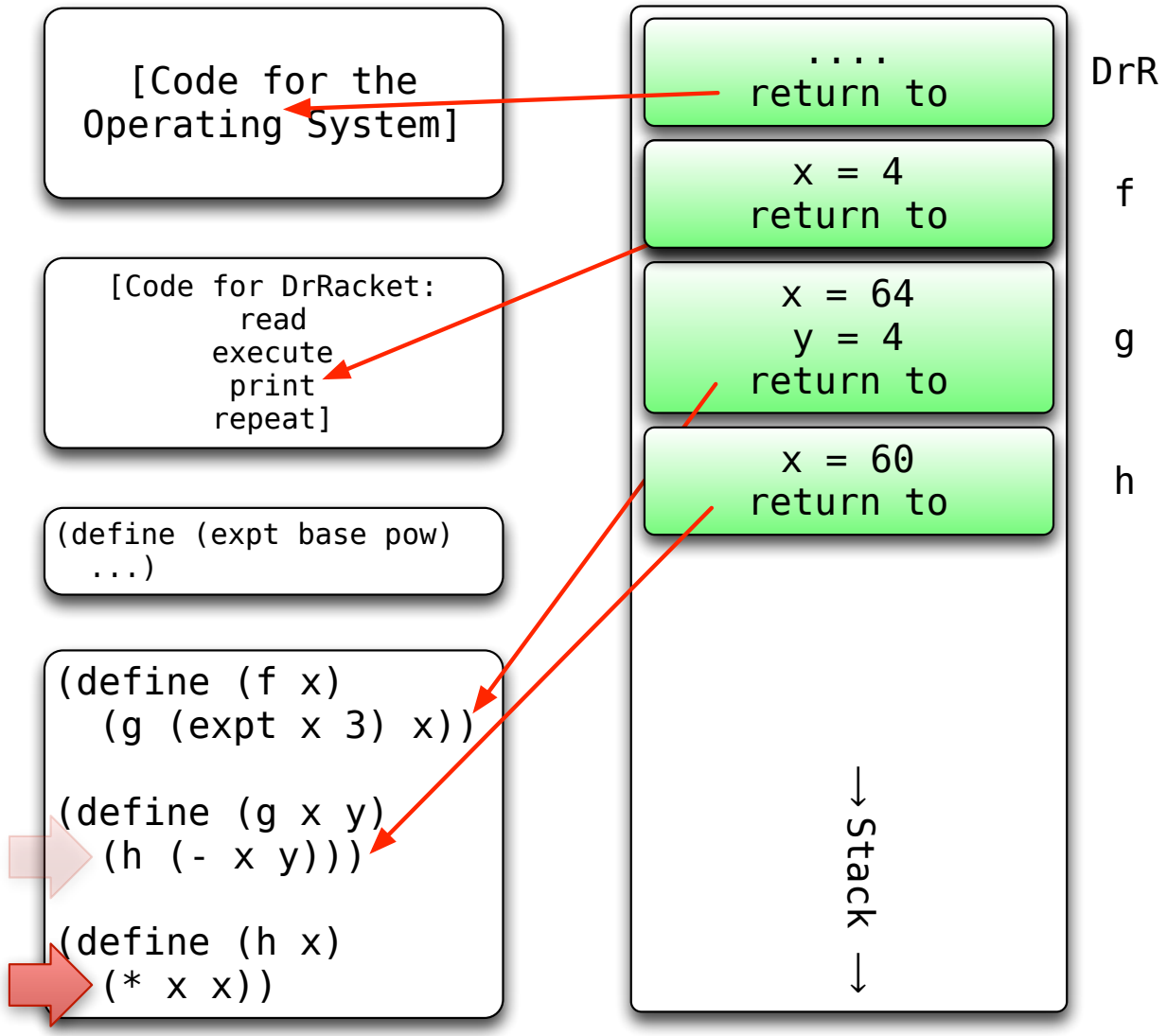
Executing Functions on a Stack: (f 4)



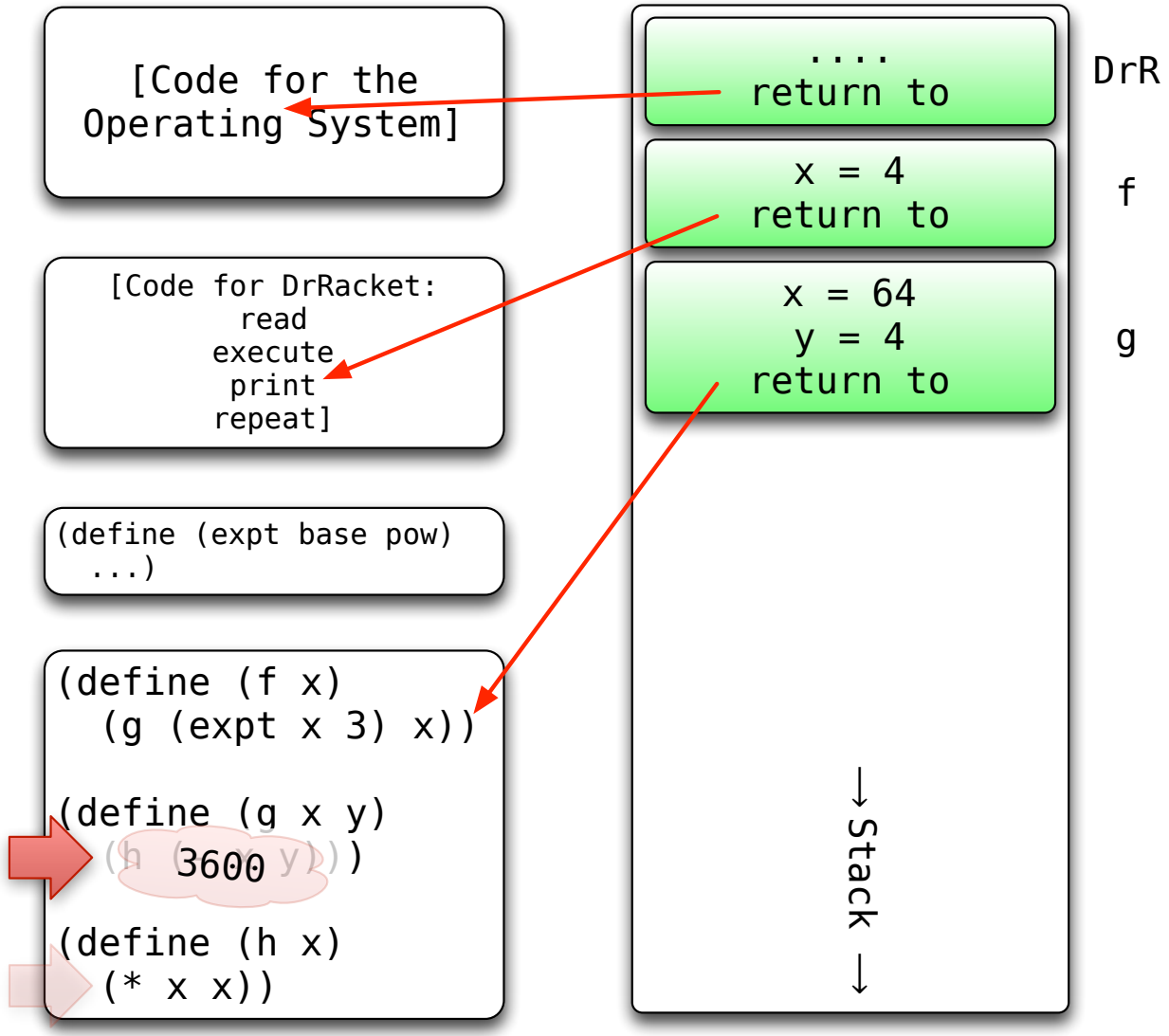
Executing Functions on a Stack: (f 4)



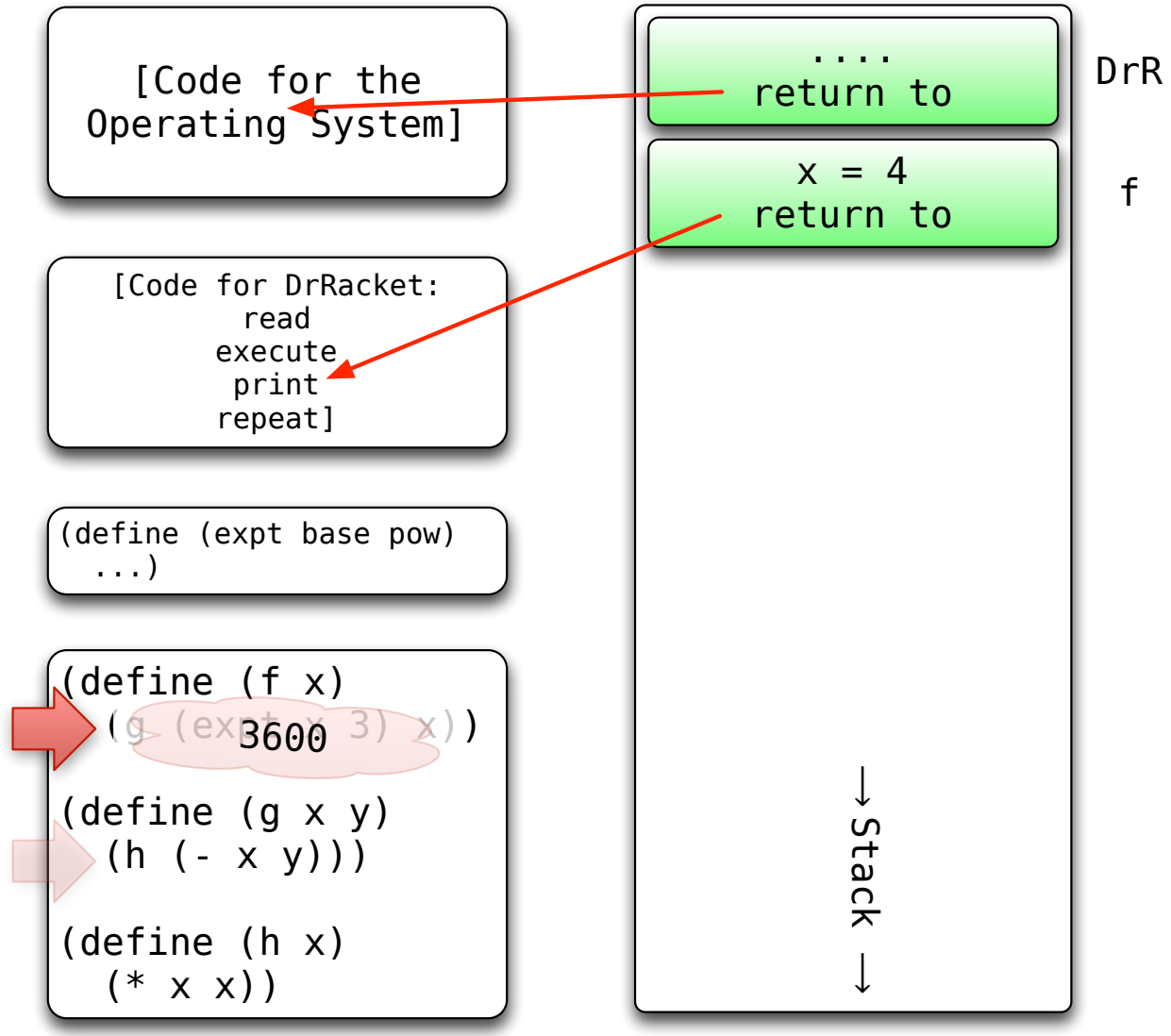
Executing Functions on a Stack: (f 4)



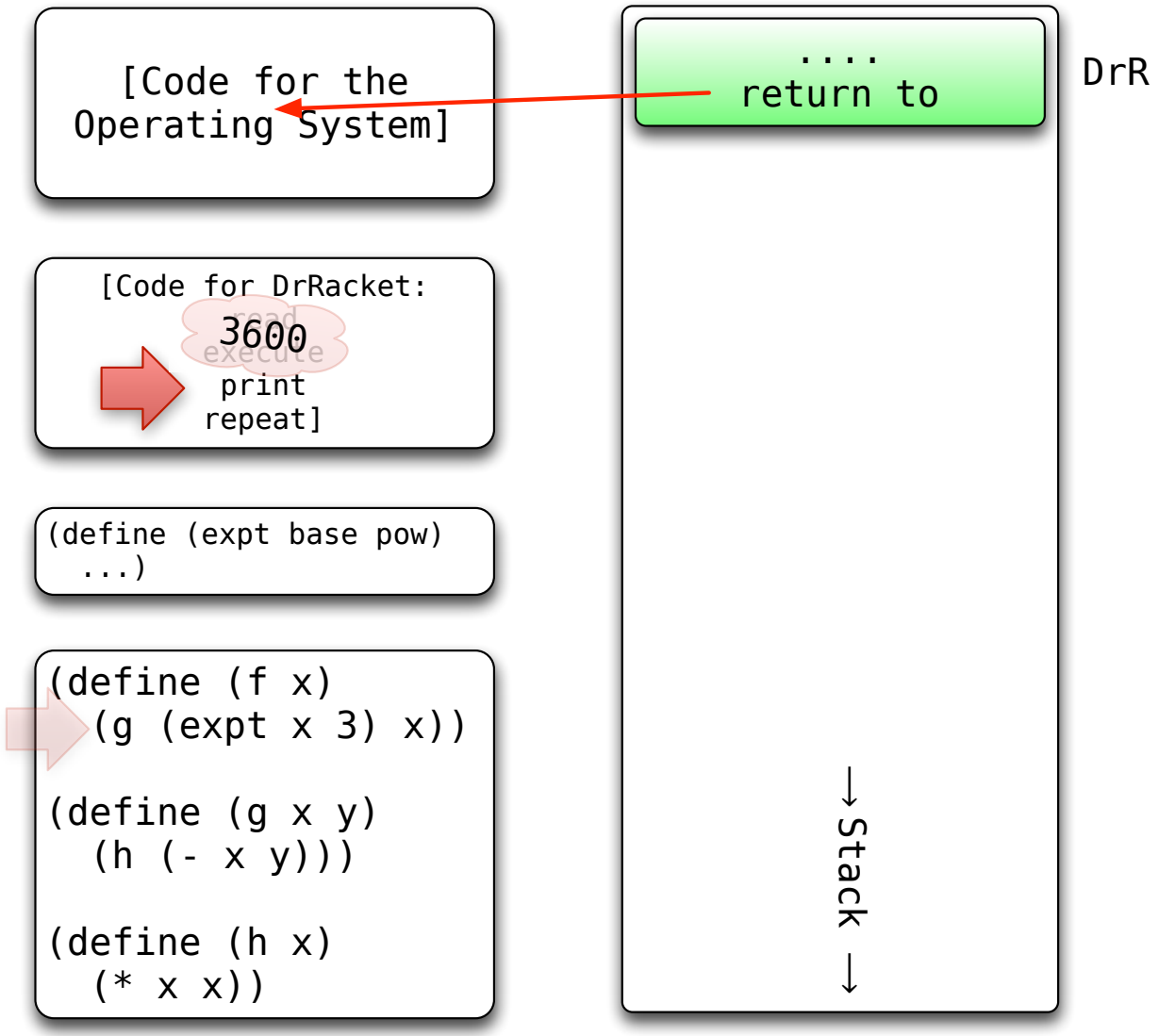
Executing Functions on a Stack: (f 4)



Executing Functions on a Stack: (f 4)



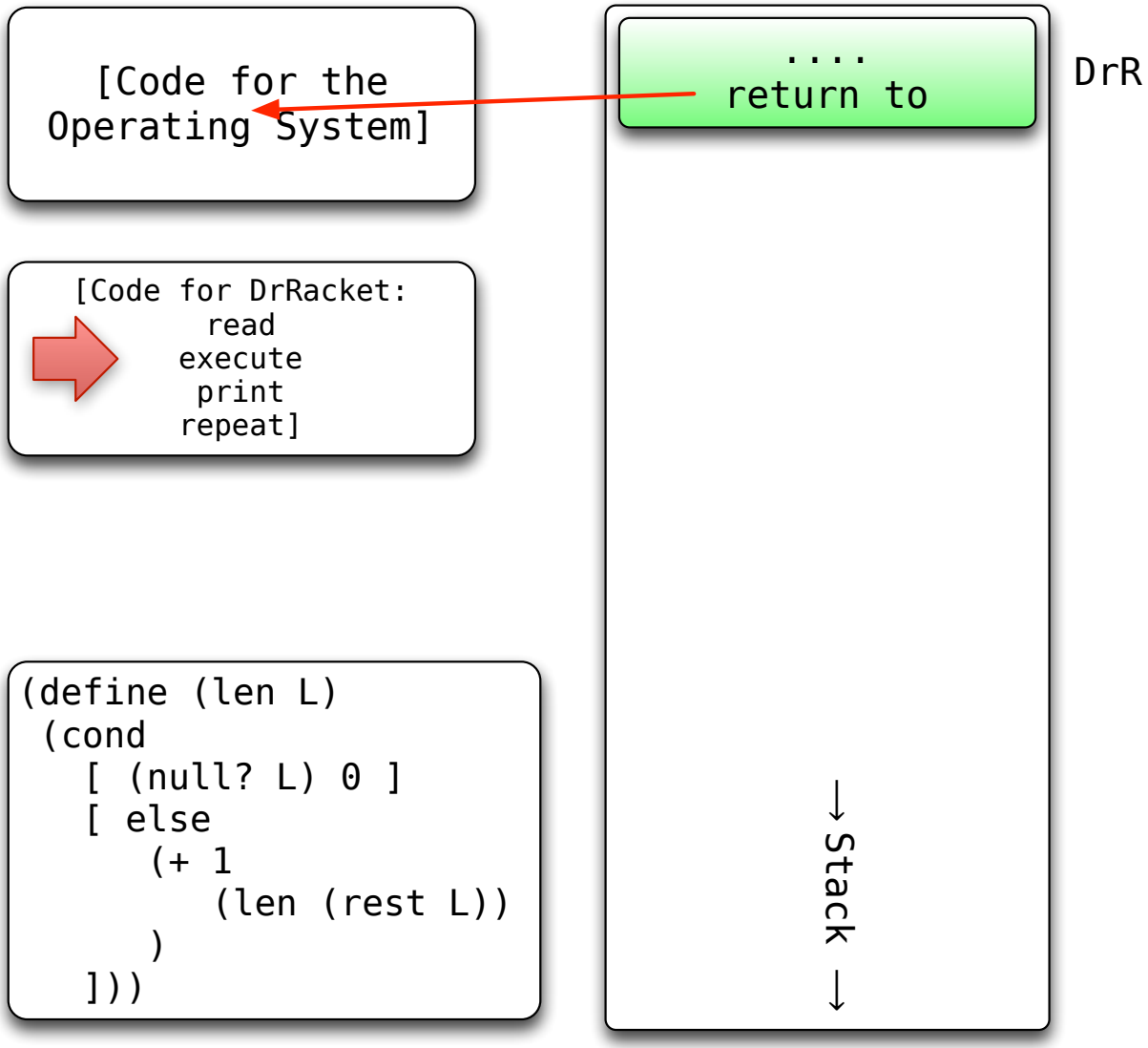
Executing Functions on a Stack: (f 4)



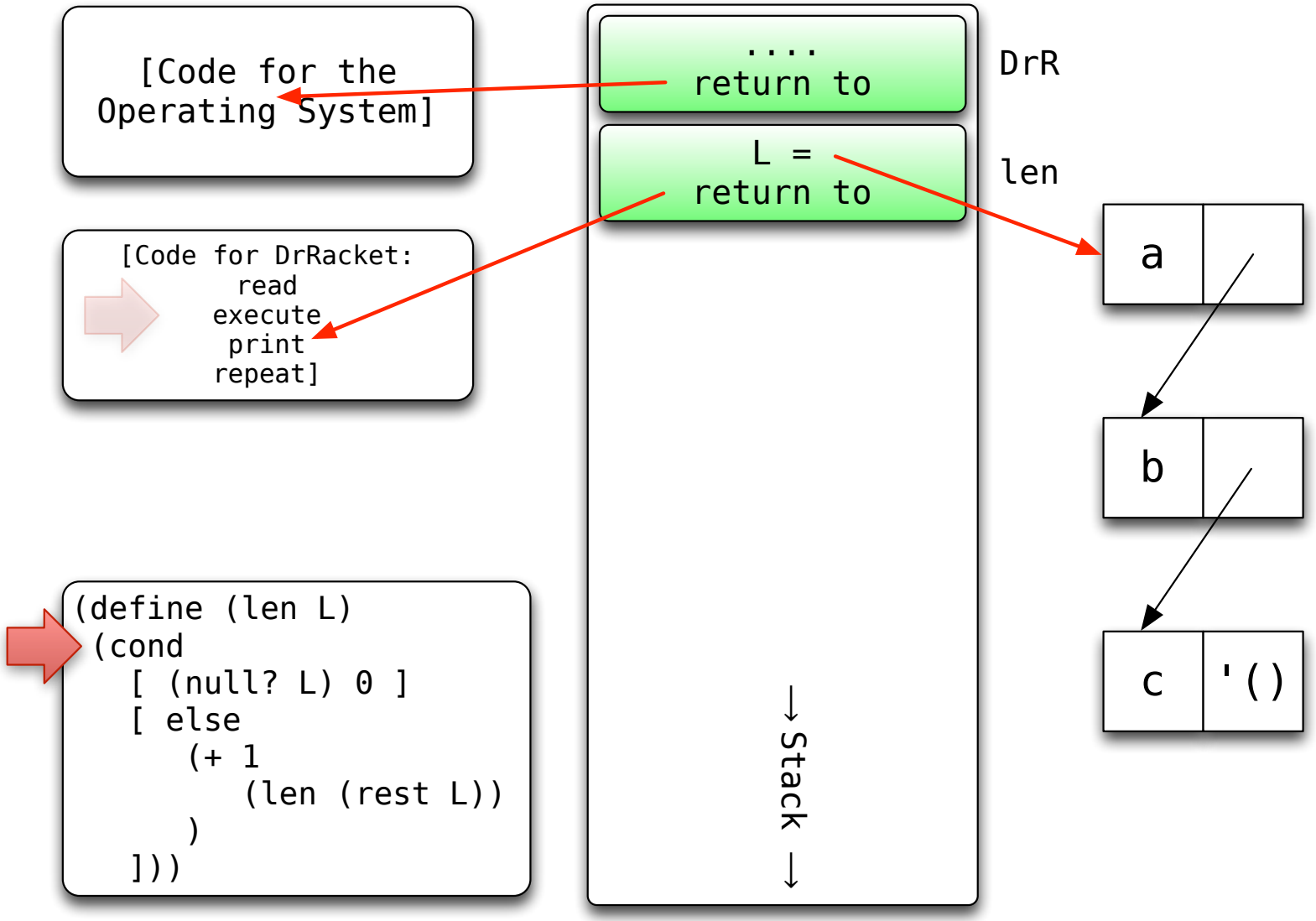
Recursion is Easy to Implement with a Stack

Things work *exactly* the same way!

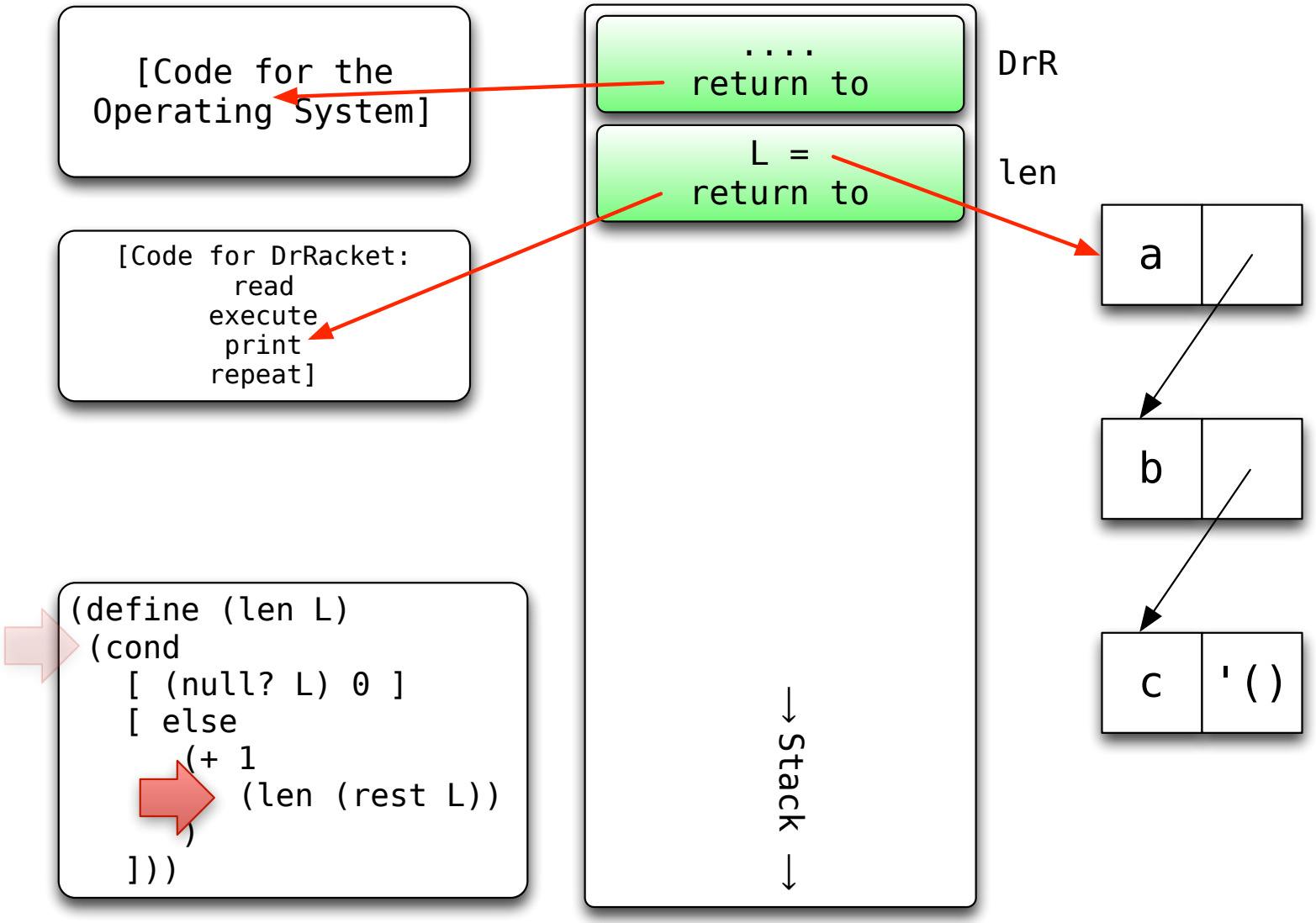
Recursion on a Stack: `(len '(a b c))`



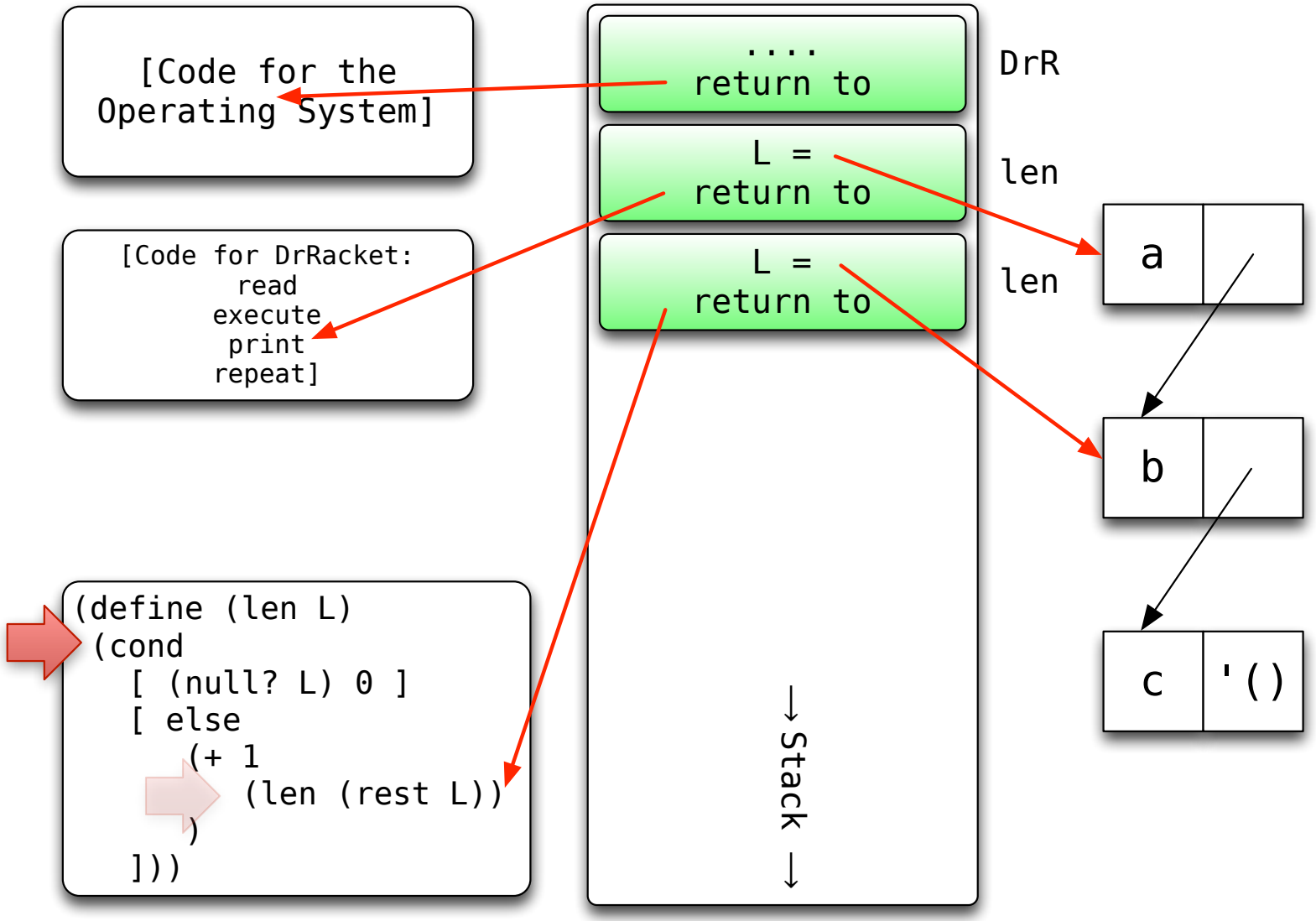
Recursion on a Stack: (len '(a b c))



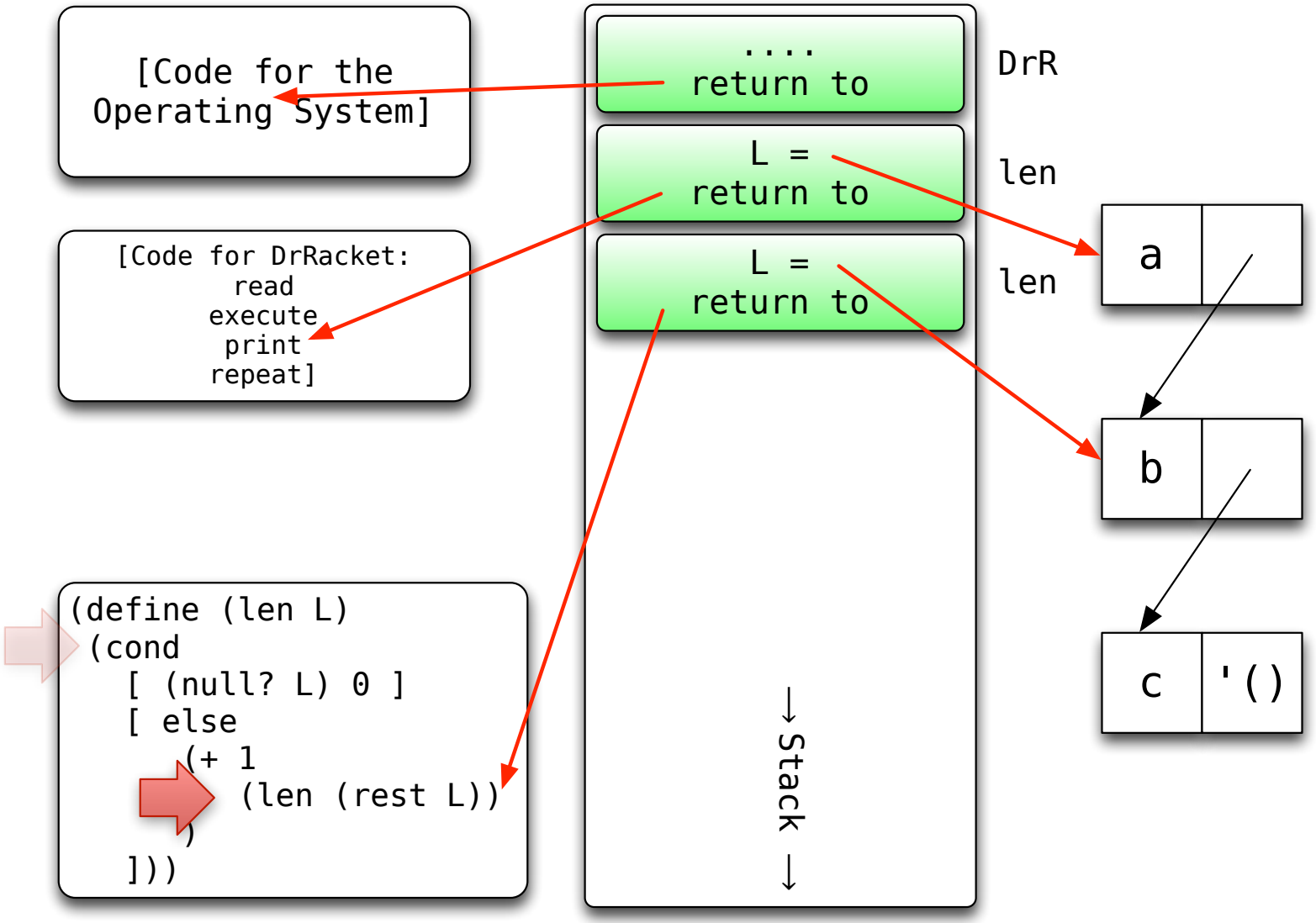
Recursion on a Stack: (len '(a b c))



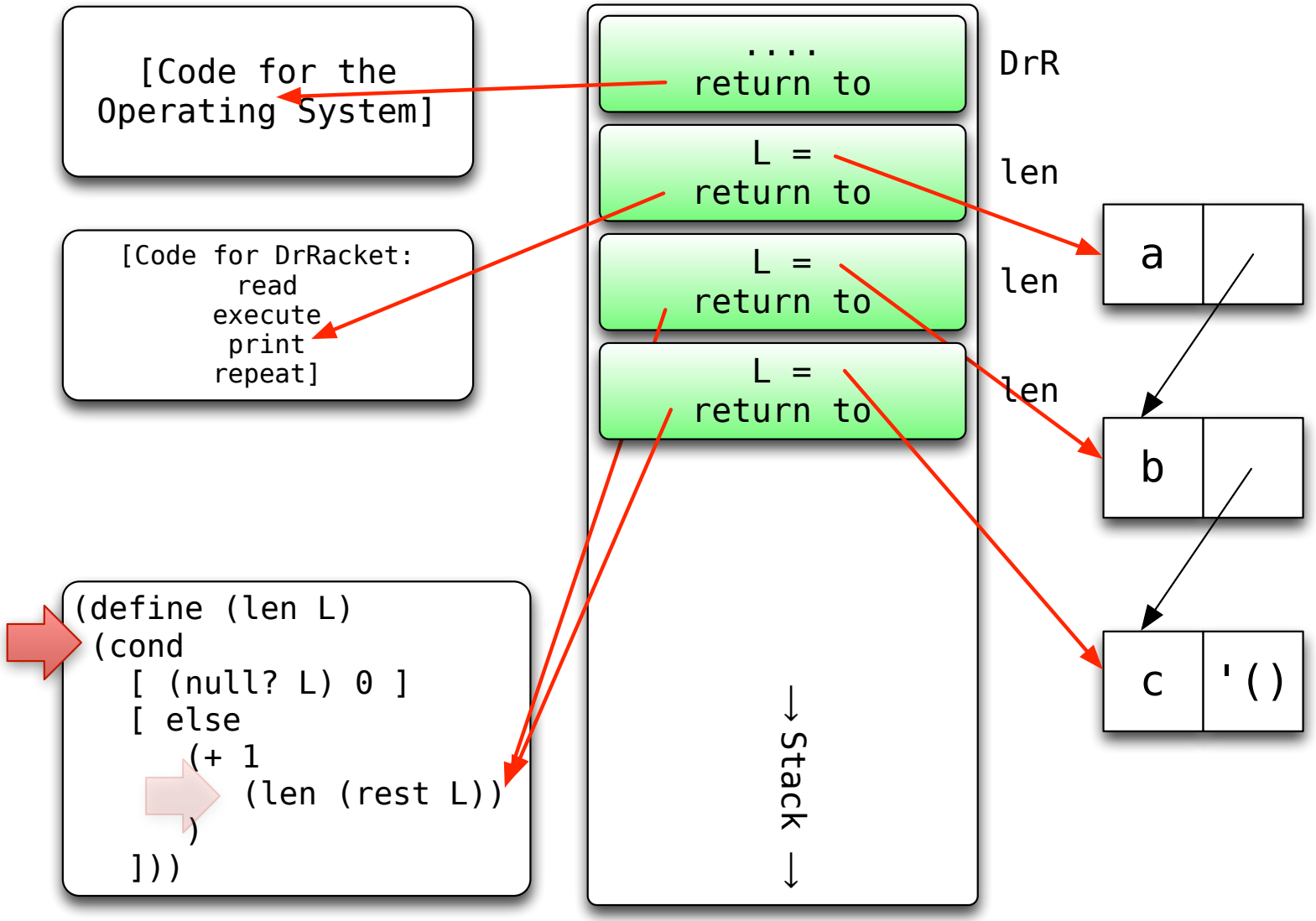
Recursion on a Stack: (len '(a b c))



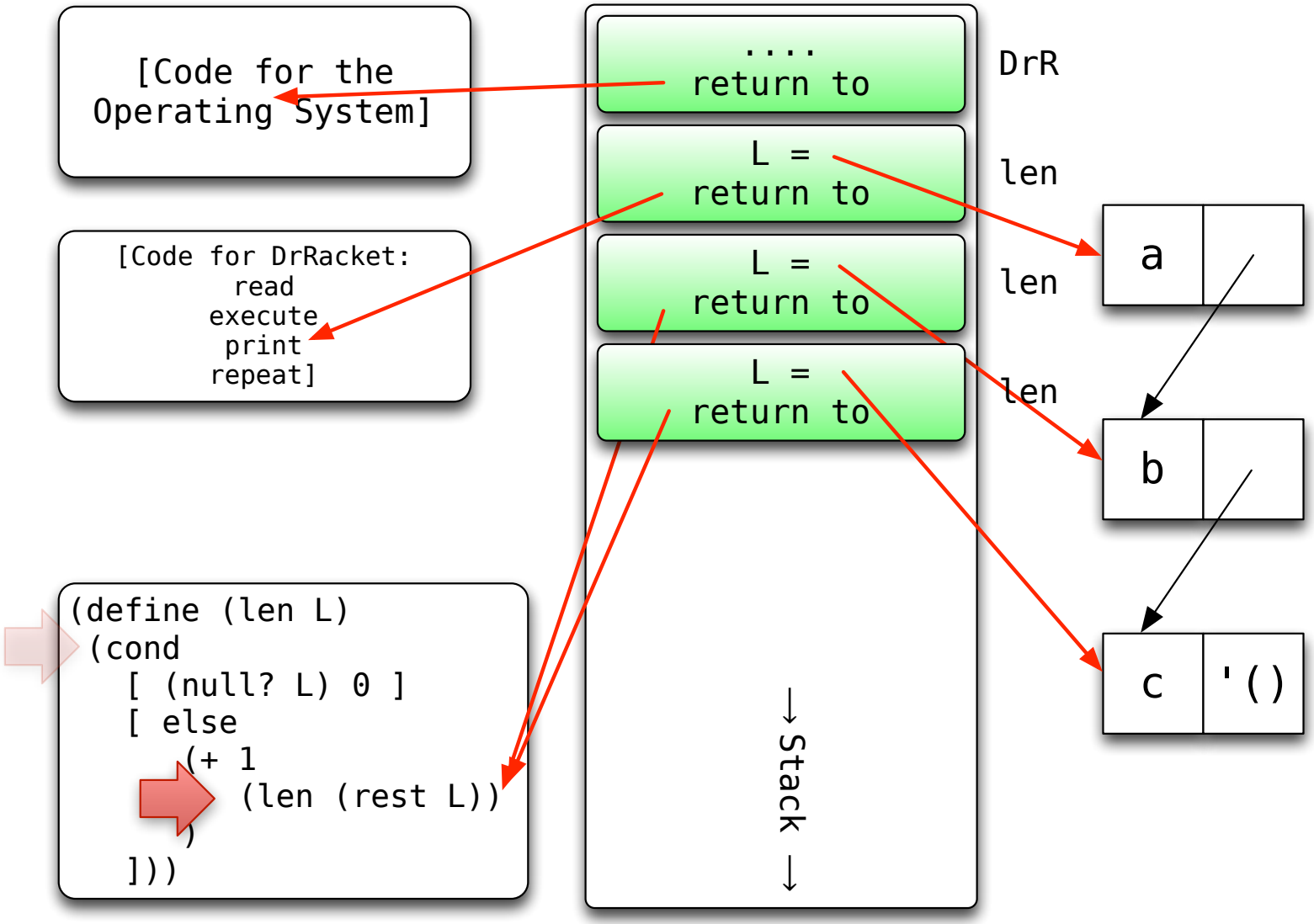
Recursion on a Stack: (len '(a b c))



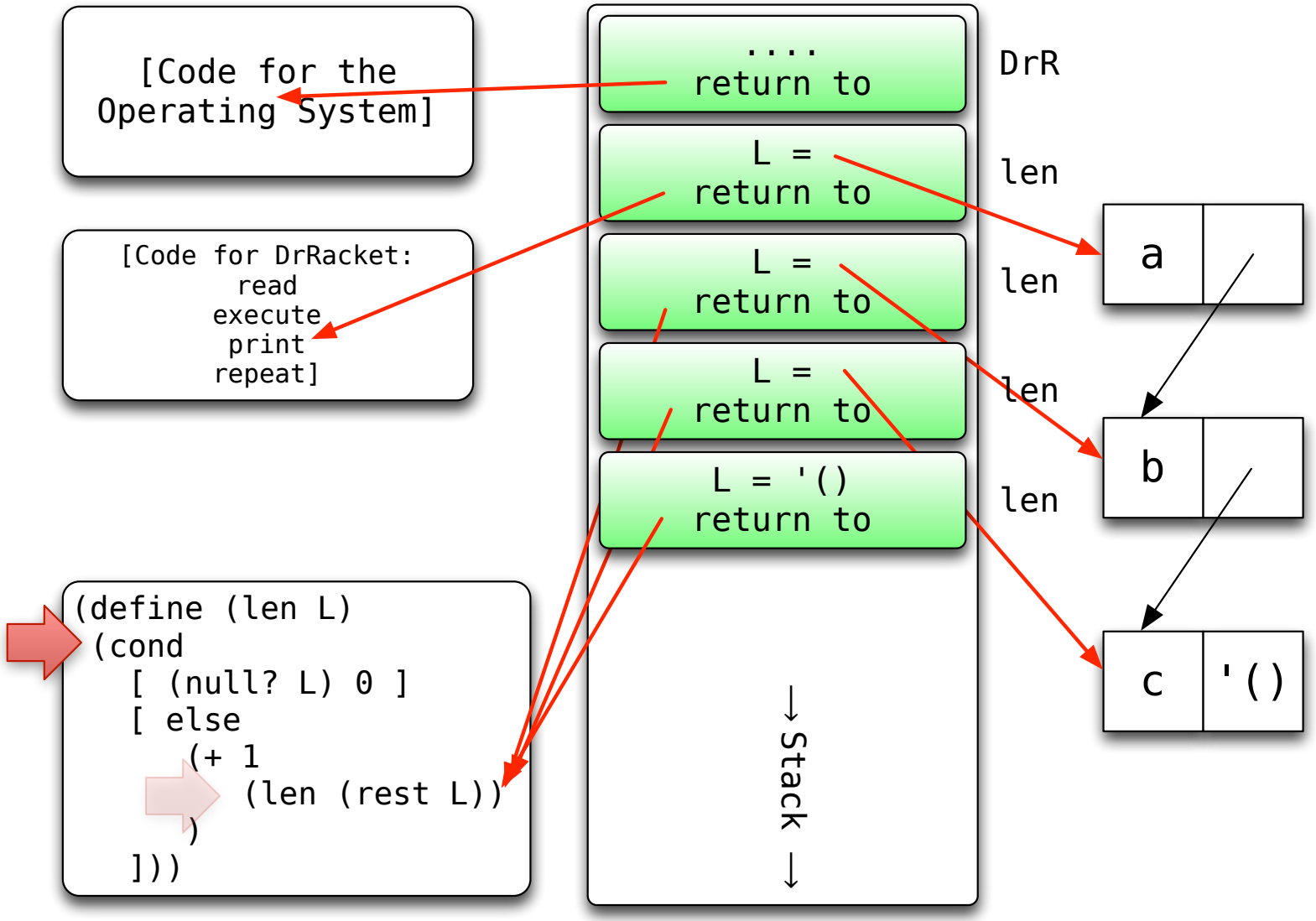
Recursion on a Stack: (len '(a b c))



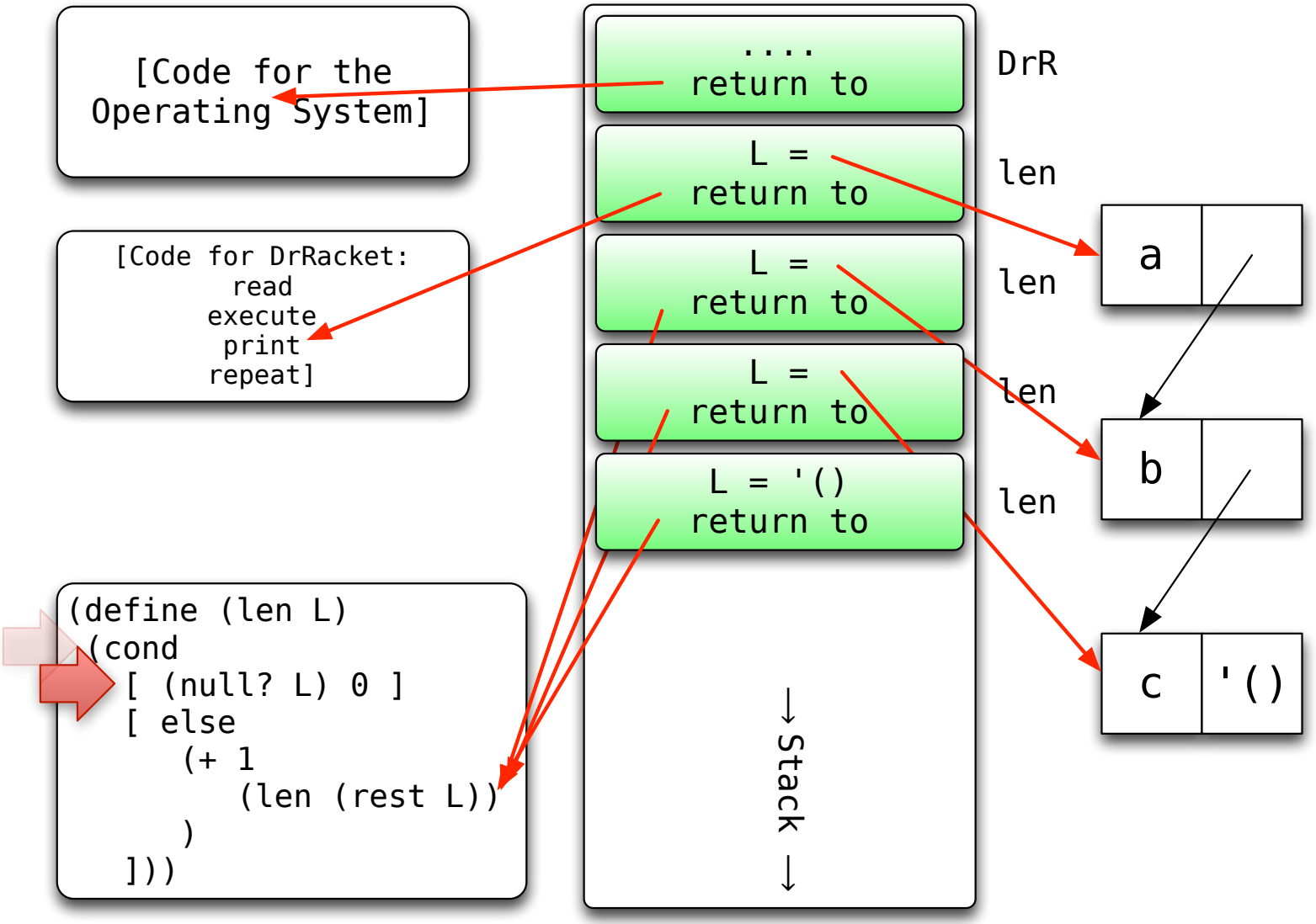
Recursion on a Stack: (len '(a b c))



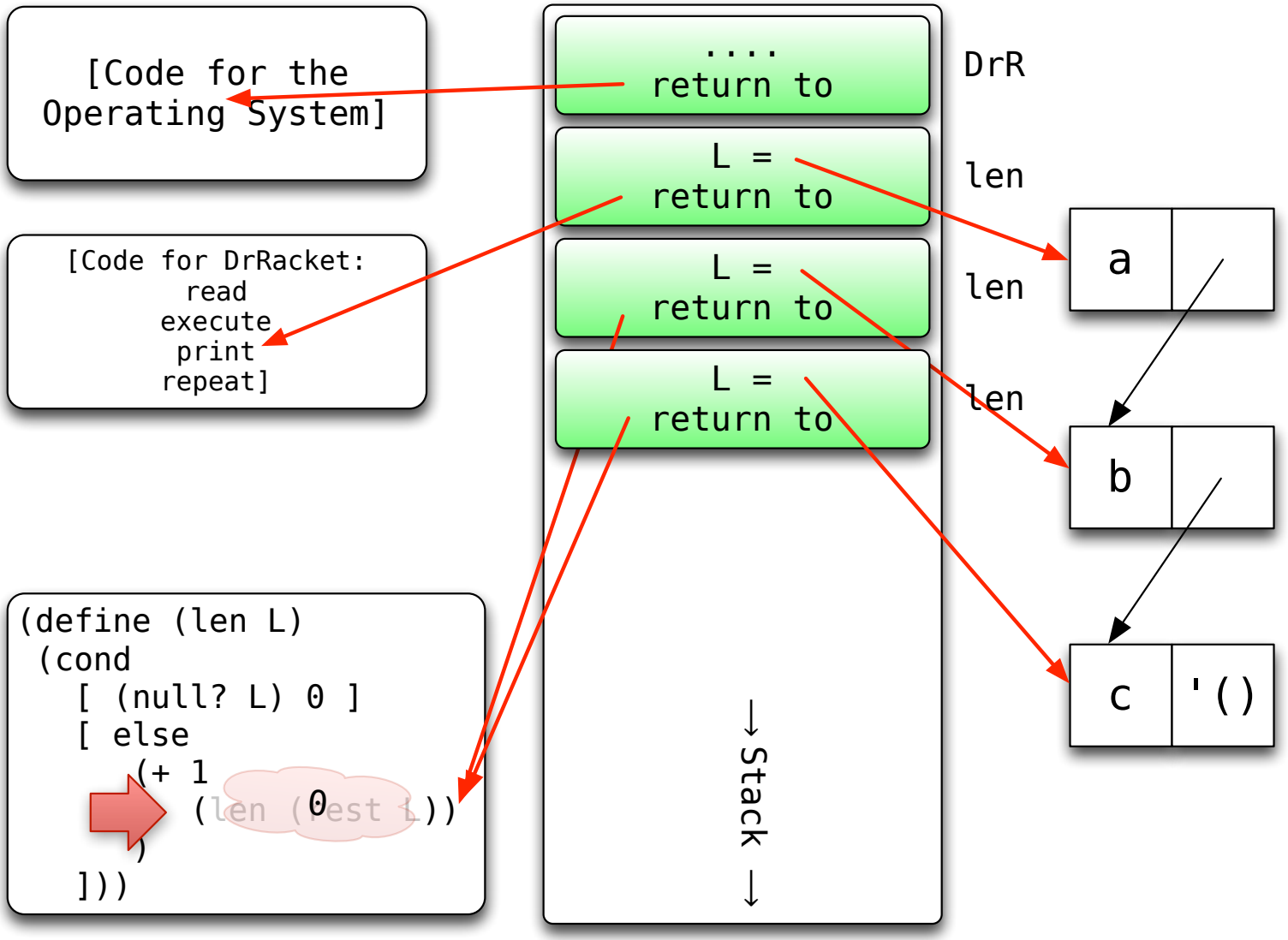
Recursion on a Stack: (len '(a b c))



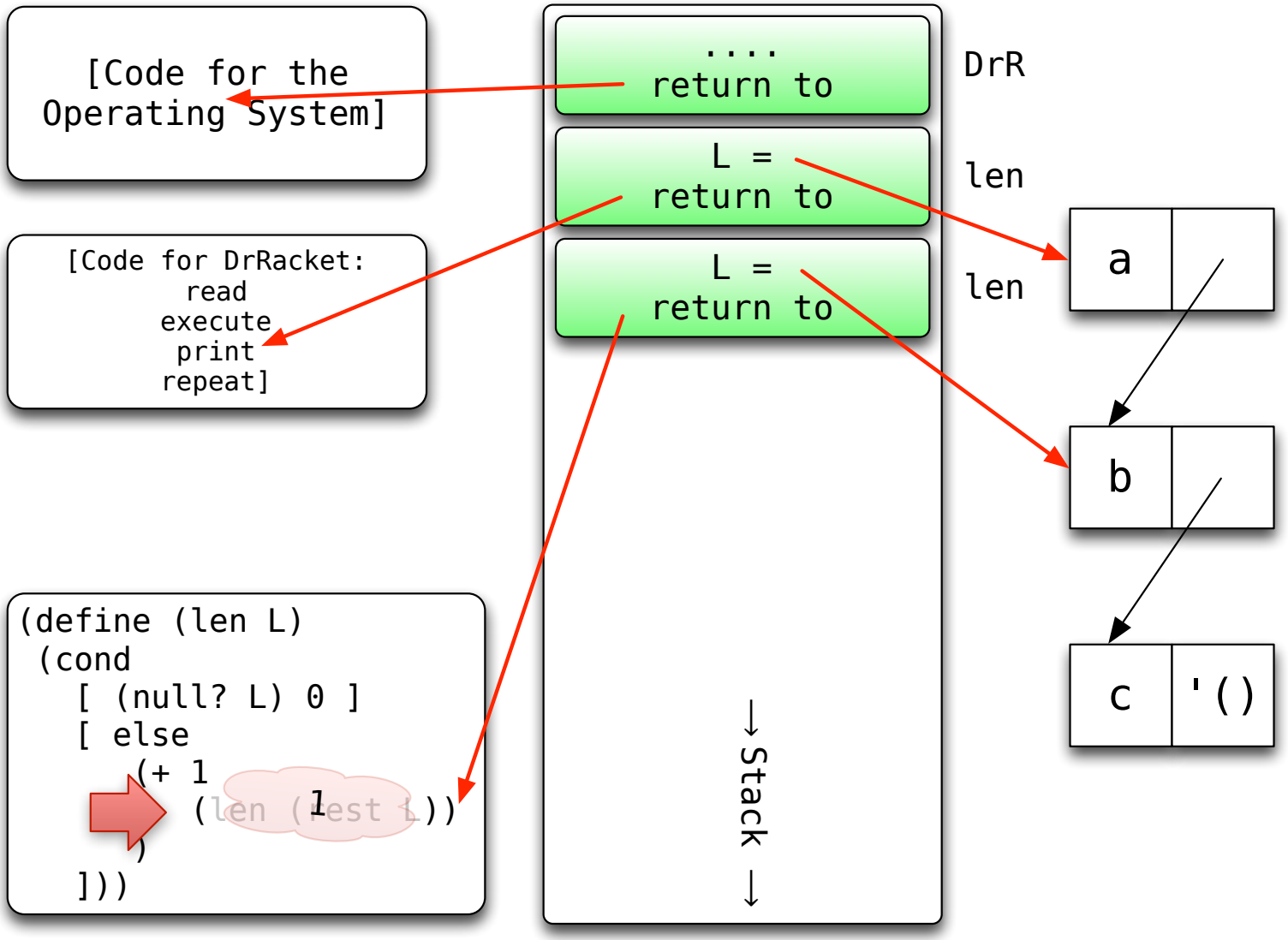
Recursion on a Stack: (len '(a b c))



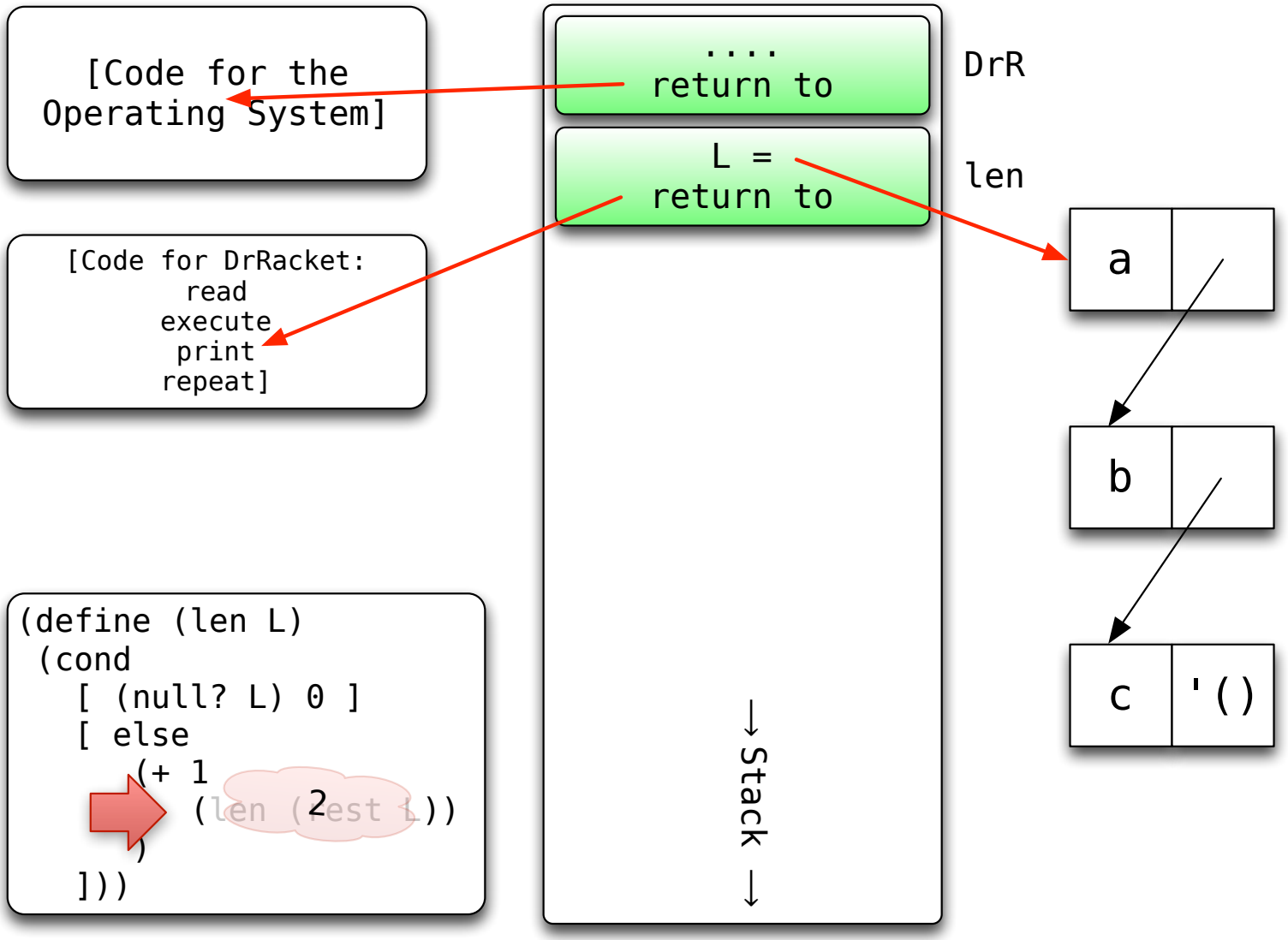
Recursion on a Stack: (len '(a b c))



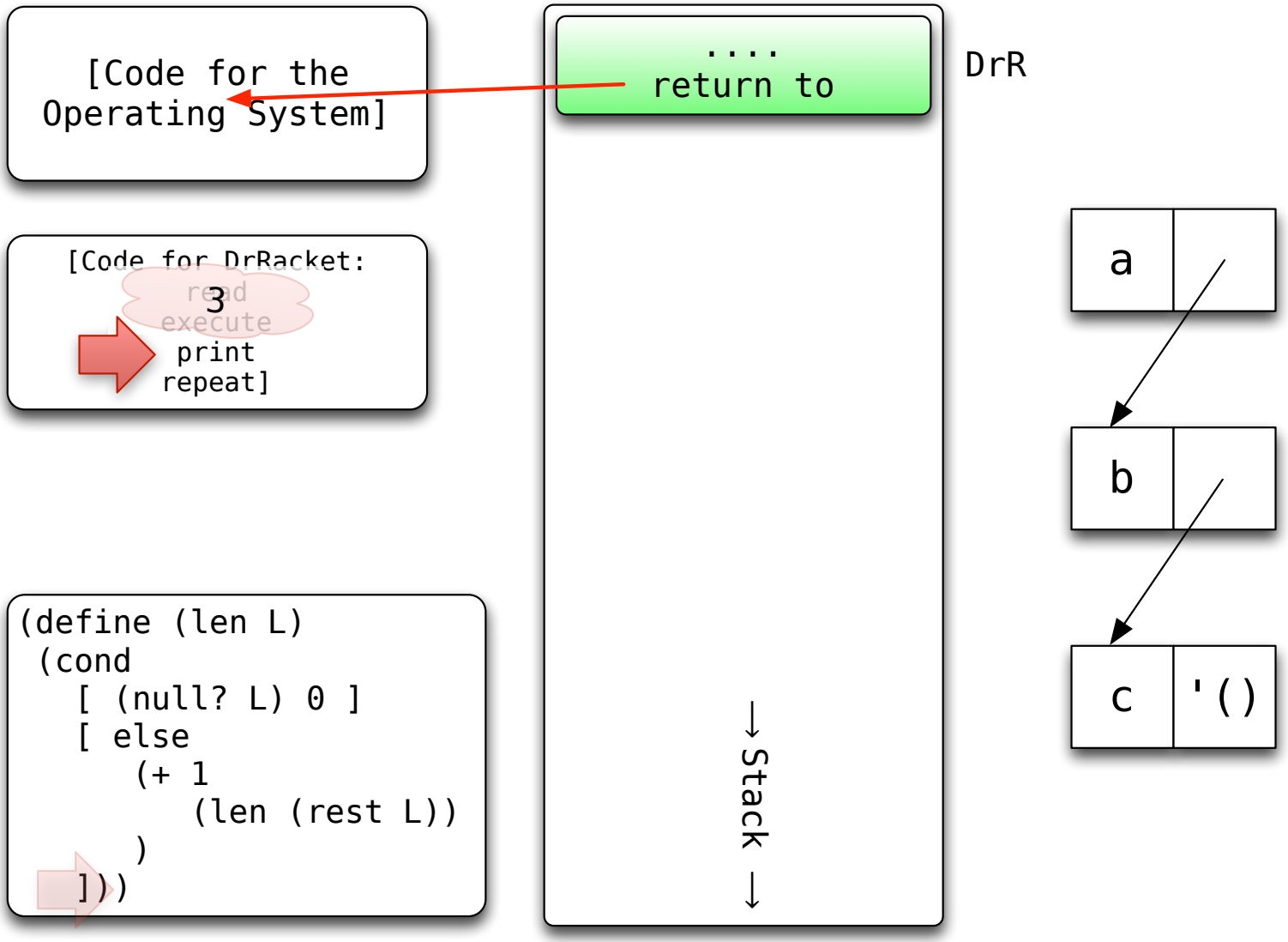
Recursion on a Stack: (len '(a b c))



Recursion on a Stack: (len '(a b c))



Recursion on a Stack: (len '(a b c))



Two Approaches To Using Recursion

"General Recursion"

- ✓ Take in a number (or list)
- ✓ Recurse on precessor (or `rest`)
- ✓ Compute answer for the given input

```
(define (fac N)
 (cond
  [ (< N 2)  1 ]
  [ else (* N (fac (- N 1))) ]
 )
)
```

"Tail Recursion"

- ✓ Add an extra "accumulator" argument
- ✓ The accumulator summarizes the numbers/list elements seen so far
- ✓ At the end, return the accumulator.

```
(define (fac N)   (fac-help N 1)

(define (fac-help N A)
 (cond
  [ (< N 2)  A ]
  [ else (fac-help (- N 1) (* N A))
 )
)
```

Comparing (fac 4)

General Recursion: $(4 \times (3 \times (2 \times 1)))$

```
(fac 4)  =  (4 * (fac 3))
         =  (4 * (3 * (fac 2)))
         =  (4 * (3 * (2 * (fac 1))))
         =  (4 * (3 * (2 * 1)))
         =  (4 * (3 * 2))
         =  (4 * 6)
         =  24
```

Tail Recursion: $(((4 \times 3) \times 2) \times 1)$

```
(fac 4)  =  (fac-help 4 1)
         =  (fac-help 3 4)
         =  (fac-help 2 12)
         =  (fac-help 1 24)
         =  24
```

Recall: Reverse

How efficient is it?

```
(define (rev L)
 (cond
   [ (null? L)  '() ]
   [  else      (append (rev (rest L))
                        (list (first L)))  ] ))
```

This definition is $O(n^2)$, where n is the length of the list. All the **append** operations are expensive.

Tail-Recursive Reverse

How efficient is it?

```
(define (rev L)
  (rev-help L '()))

;; L contains the remaining elements to reverse
;; A contains the reverse of the elements we've seen so far
(define (rev-help L A)
 (cond
   [ (null? L)  A ]
   [  else (rev-help (rest L) (cons (first L) A)) ]
 )
)
```

Since `cons`, `first`, `rest`, etc. are all $O(1)$, for a list of length n we can define the number of steps $T(n)$ needed for reverse by $T(0) = 1$ and $T(n) = 1 + T(n-1)$. Thus, $O(n)$.

Comparing (rev '(1 2 3 4))

General Recursion

```
(rev '(1 2 3 4))
  =  (append (rev '(2 3 4)) '(1))
  =  (append (append (rev '(3 4)) '(2)) '(1))
  =  (append (append (append (rev '(4)) '(3)) '(2)) '(1))
  =  (append (append (append (append (rev '()) '(4)) '(3)) '(2)) '(1))
  =  (append (append (append (append '() '(4)) '(3)) '(2)) '(1))
  =  (append (append (append '(4) '(3)) '(2)) '(1))
  =  (append (append '(4 3) '(2)) '(1))
  =  (append '(4 3 2) '(1))
  =  '(4 3 2 1)
```

Tail Recursion

```
(rev '(1 2 3 4)  =  (rev-help '(1 2 3 4) '())
                 =  (rev-help '(2 3 4) '(1) )
                 =  (rev-help '(3 4) '(2 1) )
                 =  (rev-help '(4) '(3 2 1) )
                 =  (rev-help '() '(4 3 2 1) )
                 =  '(4 3 2 1)
```