Tail Recursion Review

Spring 2016

Quick note on Orders of Growth

- O(n), O(log n), O(1), etc.
- Orders of Growth describe functions
  - Typically use for runtime
  - Can be used for other things
- This section focuses on memory usage

Memory Usage in Python

```
def fact(n):
  if n == 0:
    return 1
  else:
    return n*fact(n-1)
```

```
def fact(n):
  total = 1
  while n > 0:
    total = total*n
    n -= 1
  return total
```

Memory Usage in Python

```
def fact(n):
  if n == 0:
    return 1
  else:
    return n*fact(n-1)
```

O(n) space: n frames

```
def fact(n):
  total = 1
  while n > 0:
    total = total*n
    n -= 1
  return total
```

O(1) space: 2 variables

So… What does this mean?

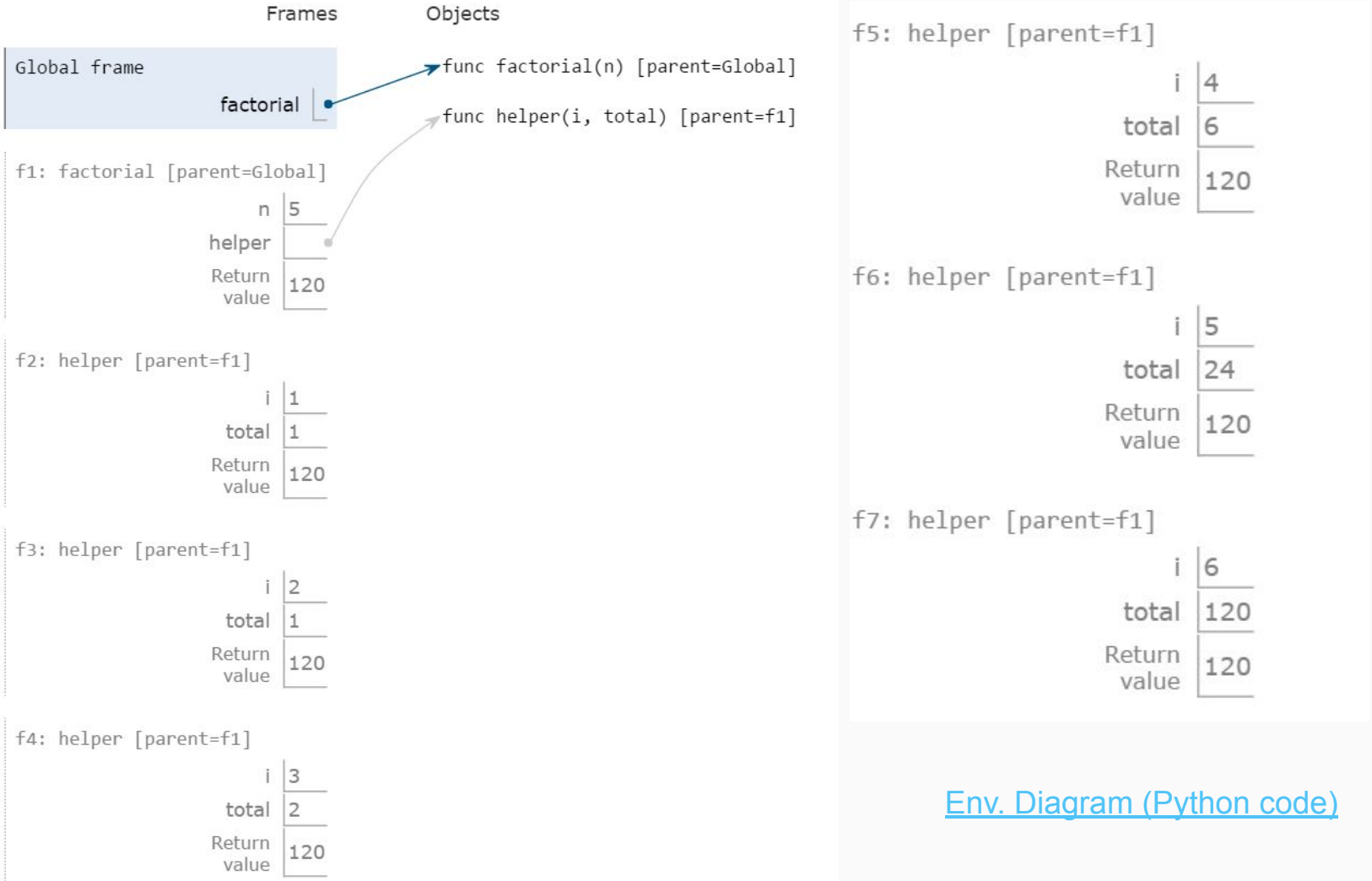
- Two functions can have same runtime and different memory usage
- Iterative functions use less space because they only have one frame

But what about Scheme? We have no iteration!

Tail Recursion Optimization!

Let's walk through an example:

```
(define (factorial n)
  (define (helper i total)
    (if (> i n) total
      (helper (+ i 1) (* total i))
  (helper 1 1))
```



[Env. Diagram (Python code)]

What does *this* mean?

- We don't need to keep all of the extra frames around
- We can just get rid of them when we're done with them
- If we do this, we get constant space!

How do we know what we can delete?

Rule of thumb: If a function call is returned *directly*, the frame can be deleted.

Ex:

(helper (+ i 1) (* i total))

Yes!

(* n (fact (- n 1)))

No!

Must occur in a tail context

Last expression in:

- define
- begin
- and
- or

Non-predicates of `if` (2nd or 3rd)

Last expression of each clause in `cond` (but not predicates)

Is this tail recursive?

```
(define (find s v)
  (cond ((null? s) False)
        ((= v (car s)) True)
        ((find (cdr s) v) True)
        (else False)))
```

Is this tail recursive?

```
(define (find s v)
  (cond ((null? s) False)
        ((= v (car s)) True)
        ((find (cdr s) v) True)
        (else False)))
```

No - We don't return the recursive call. The conditional of a `cond` is not a tail context.

Is this tail recursive?

```
(define (find s v)
  (cond ((null? s) False)
        ((= v (car s)) True)
        (else (find (cdr s) v))))
```

Is this tail recursive?

```
(define (find s v)
  (cond ((null? s) False)
        ((= v (car s)) True)
        (else (find (cdr s) v))))
```

Yes - We return the recursive call. The **end** of a `cond` is a tail context.

Reverse Tail Recursion

Practice Problem: Write a tail-recursive version of `reverse` in Scheme.

```
(define (reverse xs)
   'YOUR-CODE-HERE
)
```

Python Reverse

```
def reverse(xs):
    result = Link.empty
    while xs is not Link.empty:
        result = Link(xs.first, result)
        xs = xs.rest
    return result
```

Scheme Solution

```
(define (reverse xs)
 (define (reverse-iter xs result)
   (if (null? xs)
     result
     (reverse-iter (cdr xs) (cons (car xs) result)))
 (reverse-iter xs nil))
```

Counting Stars (Summer 2015)

```
scm> (count 3 '(1 3 4 3))
2
scm> (count 42 '(4 2))
0
```

```
(define (count num lst)
  (define (helper lst total)


  )
(helper ______ _______))
```

Counting Stars (Summer 2015)

```
(define (count num lst)
  (define (helper lst total)
    (cond ((null? lst) total)
    ((= (car lst) num) (helper (cdr lst) (+ total 1)))
    (else (helper (cdr lst) total))))
(helper lst 0))
```

Filter

```
scm> (filter is-odd? '(1 2 3 4 5))
(1 3 5)

(define (filter fn lst)
  'YOUR-CODE-HERE



)
```

Filter

```
(define (filter fn lst)
  (define (helper lst result)
    (cond
      ((null? lst) result)
      ((fn (car lst)) (helper (cdr lst) (cons (car lst)
result))
      (else (helper (cdr lst) result))))
  (helper (reverse lst) nil))
```