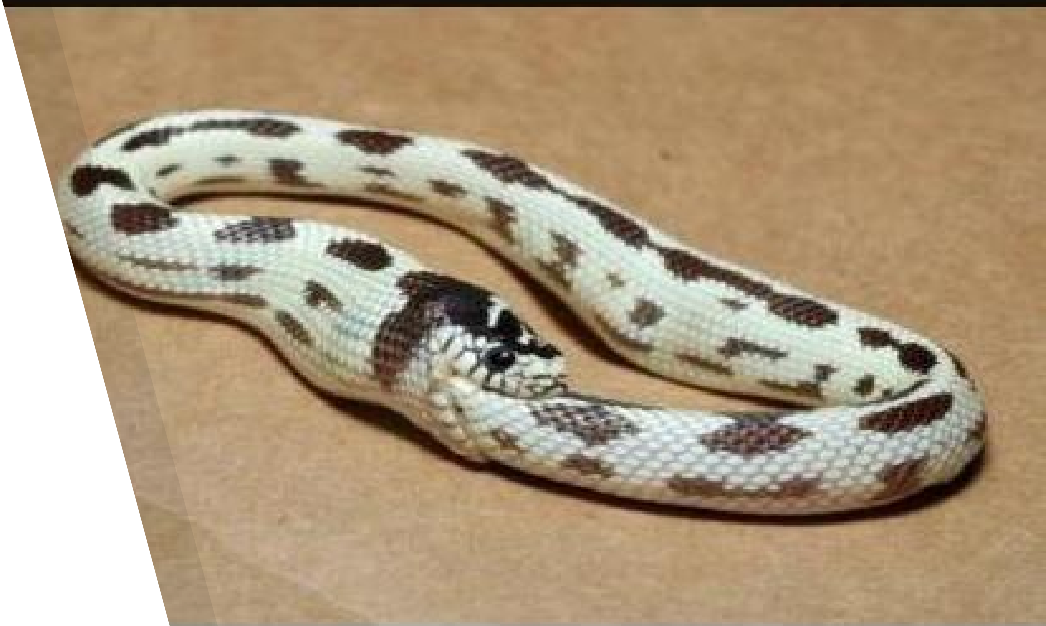
CS 61A
DISCUSSION 9

April 06, 2017

* despite the connotations here, Python isn't actually optimized for tail recursion.



TOPICS FOR TODAY

- Interpreters
- Tail recursion

ANNOUNCEMENTS

- The Scheme project has been released! It’s due on Wed. 4/19 at 11:59pm. (Start early, because the extra credit problem is a pain.)
- Maps composition revisions are due on Sun. 4/16 at 11:59pm.
- Special topics discussion will spend the last 20 minutes on AI, and the 10 minutes before that on bagels. (Voting distribution: 19.67% none, 13.11% graphics, 32.79% AI, 29.51% bagels, 4.92% other)

ATTENDANCE REVIEW

Very few people got this correct.

What does this evaluate to?

```
(car (cdr (list 1 (cons () 2))))
```

```
Answer: (() . 2)
```

ARE YOU HERE YET?

In order to obtain the attendance links, you must first solve a riddle.
(Yes, I do derive joy from this.)

116: What grows downward in CS diagrams but upward in real life?
(noun, singular, 4 letters)

140: What grows downward in memory but upward in real life?
Hint: it also overflows, and when it does it's very useful for getting programming help. (noun, singular, 5 letters)

tiny.cc/<riddle solution><your section number>

INTERPRETERS

Interpreters

tl;dr An interpreter is a program that reads, evaluates, and prints code. It will do this in a loop, i.e. on a line-by-line basis.

There are three steps to the REPL[oop]:

1. **Read**.

Tokenize the code (i.e. break it into appropriately-sized chunks) and *convert each token into the interpreter's preferred representation*. For Scheme, our preferred representation will be `Pairs` – alongside the `nil` object and primitives like numbers. `Pairs` are analogous to the Scheme procedure `cons`.

e.g. `(+ 1 2)` ➔ `Pair('+', Pair(1, Pair(2, nil)))`
e.g. `(- 5)` ➔ `Pair('-', Pair(5, nil))`

Interpreters, continued

2. **Evaluate.**
Turn the expressions from “**Read**” into values. If there are any call expressions (operators being called on operands), we’ll need to *apply* said operator to said operands.

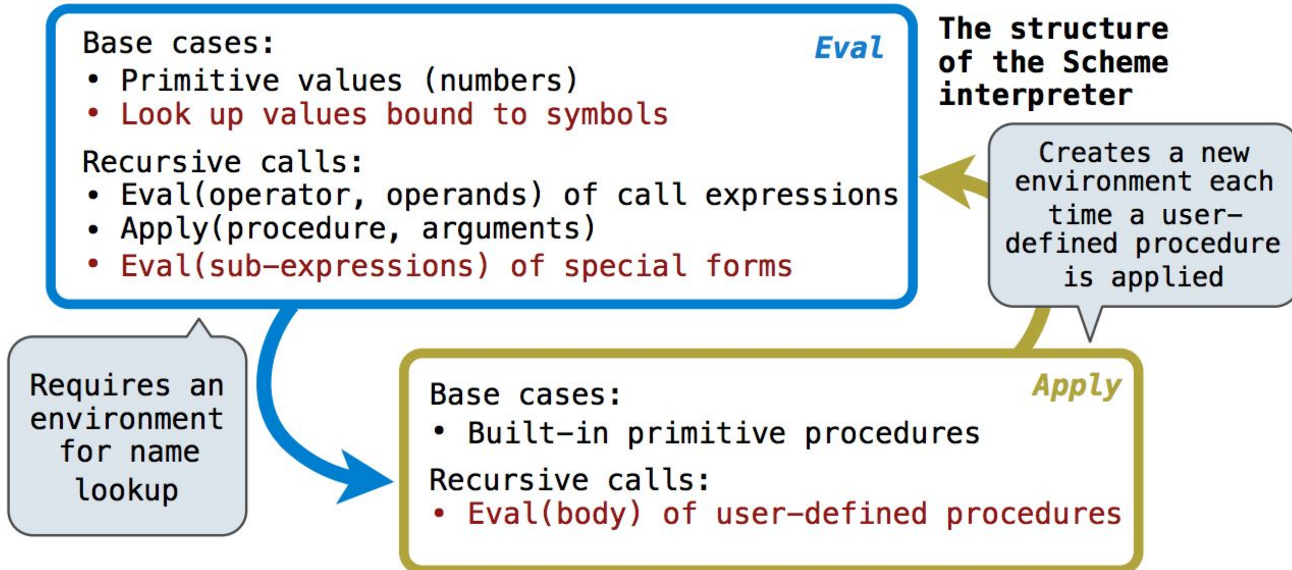
e.g. `Pair(‘+’, Pair(1, Pair(2, nil)))` ➔ 3
e.g. `Pair(‘-’, Pair(5, nil))` ➔ -5

3. **Print.**
Print the evaluated value for the user to see. In this class, you don’t need to worry about implementing this step yourself.

e.g. `3 ➔ <displaying a 3 on the screen>`
e.g. `-5 ➔ <displaying a -5 on the screen>`

Interpreters

Here's a diagram representing the `eval/apply` structure, which is where you'll be doing most of the work when you start implementing interpreters.



Counting calls to `eval` and `apply`

The whole expression is always sent first to `eval`.

- If it's a **primitive**, that's it – its value can immediately be returned.
- If it's a **call expression**, we `eval` the operator and the operands and then pass everything to `apply`.
- If it's a **special form**, we follow some particular evaluation sequence (you should already know it; just think "*what gets evaluated* for the form?").

Note that a call expression's **operator** and **operands** are not necessarily primitives; we might need to follow a long, recursive `eval` sequence when we evaluate them.

Counting calls to `eval` and `apply`

example: **`(+ 2 4 6 8)`**

<u>`eval`</u>

<u>`apply`</u>

Counting calls to `eval` and `apply`

example: `(+ 2 4 6 8)`

<table>
<tr><th><u>eval</u></th><th><u>apply</u></th></tr>
<tr><td>(+ 2 4 6 8)
+
2
4
6
8</td><td>(+ 2 4 6 8)</td></tr>
<tr><td>Total: 6</td><td>Total: 1</td></tr>
</table>

Counting calls to `eval` and `apply`

example: `(+ 2 (* 4 (- 6 8)))`

<u>`eval`</u>

<u>`apply`</u>

Counting calls to `eval` and `apply`

example: **`(+ 2 (* 4 (- 6 8)))`**

<u>`eval`</u>

```
(+ 2 (* 4 (- 6 8)))
+
2
(* 4 (- 6 8))
*
4
(- 6 8)
-
6
8
```

Total: 10

<u>`apply`</u>

```
(- 6 8)
(* 4 -2)
(+ 2 -8)
```

Total: 3

It's true; all of it;
there is no iteration in Scheme.

Instead, there's something better.

TAIL CALL
OPTIMIZATION

Tail call optimization

What is a **tail call**? It's technically defined as a function call in a tail context – but if you find that confusing, just think of it as a call that's the last thing to happen in the function body. ("After the call is finished, nothing else is going to happen here.")

In a language optimized for this type of behavior, we can collapse all such calls into a single frame. Since a tail call is the final action taken inside of its function body, once we initiate the call we don't need to retain any data from the calling frame. Thus, when we execute the tail call we can just overwrite the old frame in memory – we won't have to create a new one.

Hence constant space! (All frames are able to reuse the same region of memory.)
By extension, this also means no stack overflows from excessive recursion depth.

Tail call optimization, *cont*.

In other words, we can “replace” each frame with the frame spawned by a tail call – because the return value will be the same.

(Remember that in Scheme, the return value is always just the last thing in the function body – *which here is the result of the tail call, since the tail call is the last thing in the function body*!
Presumably that was by design. Beautiful, right?)

Tail “recursion”

Tail recursion: making sure that all of your recursive calls are **tail calls**.
Still constant space – they’re tail calls.

(constant space in the same way Python *iteration* is constant space)

Tail call optimization II

the following notes are courtesy of Kyle Raftogianis

- tail calls can close the current stack frame
  - meaning that in general, tail calls do not change the number of stack frames
  - exceptional cases: current frame is the parent frame of some function
  - “tail call optimization”
- non-tail calls always make a new stack frame

This goes for all tail calls, not just *recursive* tail calls.
However, recursive tail calls (i.e. tail recursion) are usually where you’ll see the largest payoff.

Tail recursive candy for the eye

(that is intellectually demanding)

I give you the [non-alternative] `fact` function. I hope you can figure out what it does. (*Hint: orial*)

Ordinary (`O(n)` space):

```
(define (fact n)
  (if (<= n 0) 1
      (* n (fact (- n 1)))
  )
)
```

Tail recursive (`O(1)` space):

```
(define (fact n)
  (define (fact-helper n result)
    (if (<= n 0) result
        (fact-helper (- n 1)
                     (* n result)))
  )
  (fact-helper n 1)
)
```

Let's walk through the env. diagrams

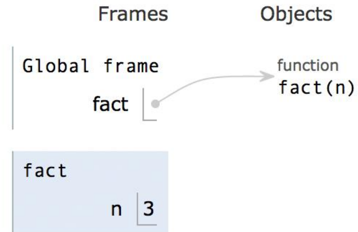
(I guess you're not done with those yet.)

Note that there are no official environment diagrams for Scheme, but I'll assume they're drawn the same way as they are for Python.

For later comparison, we'll first step through a call to the ordinary (non-tail recursive) `fact` function. Hopefully the following environment diagram is beyond elementary for you guys. :)

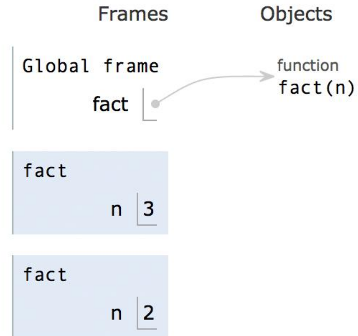
ORDINARY fact (1 / 8)

```
(define (fact n)
  (if (<= n 0) 1
      (* n (fact (- n 1)))
  )
)

(fact 3)
```



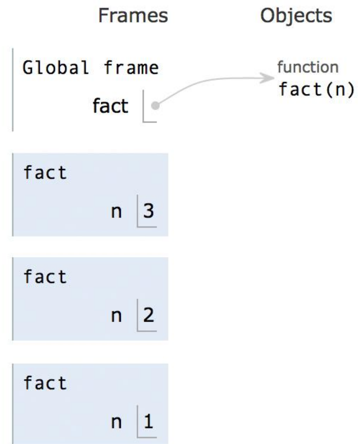
ORDINARY fact (2 / 8)

```
(define (fact n)
  (if (<= n 0) 1
      (* n (fact (- n 1)))
  )
)

(fact 3)
```



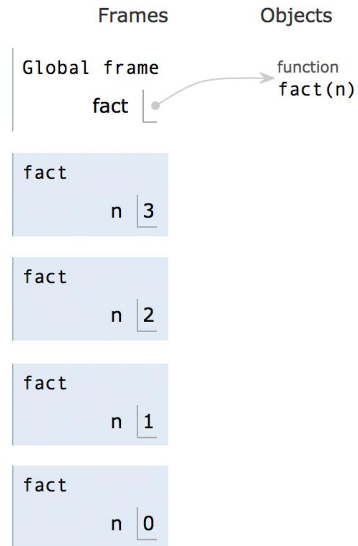
ORDINARY fact (3 / 8)

```
(define (fact n)
  (if (<= n 0) 1
      (* n (fact (- n 1)))
  )
)

(fact 3)
```



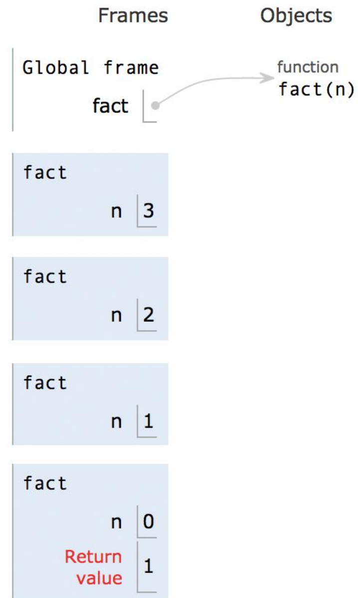
ORDINARY fact (4 / 8)

```
(define (fact n)
  (if (<= n 0) 1
      (* n (fact (- n 1)))
  )
)

(fact 3)
```



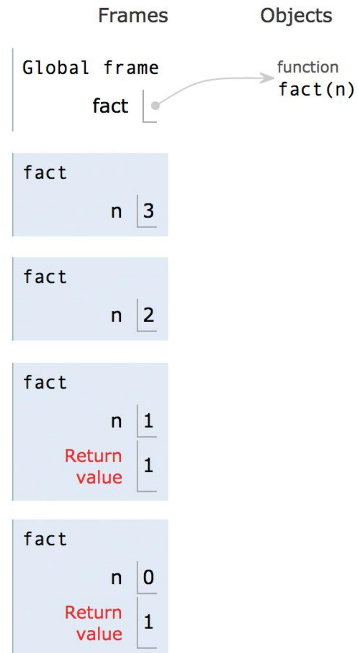
ORDINARY fact (5 / 8)

```
(define (fact n)
  (if (<= n 0) 1
      (* n (fact (- n 1)))
  )
)

(fact 3)
```



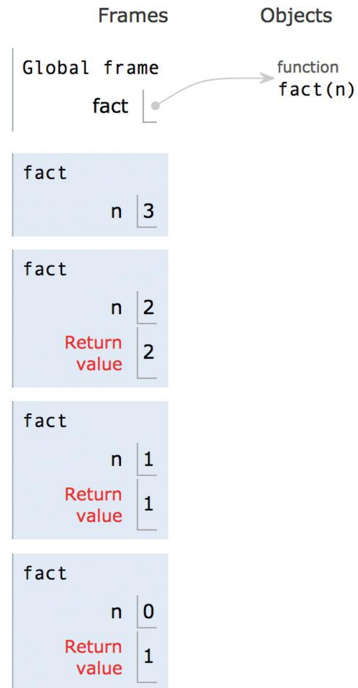
ORDINARY fact (6 / 8)

```
(define (fact n)
  (if (<= n 0) 1
      (* n (fact (- n 1)))
  )
)

(fact 3)
```



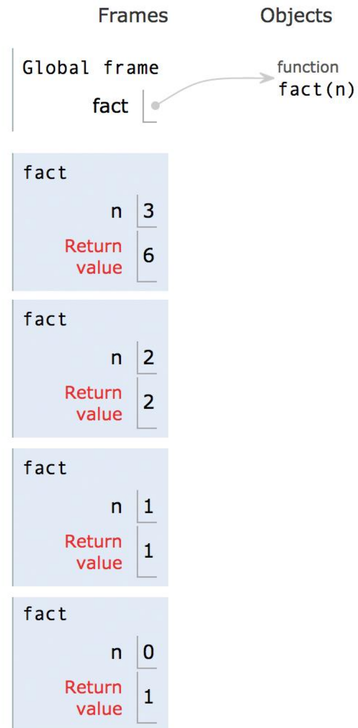
ORDINARY fact (7 / 8)

```
(define (fact n)
  (if (<= n 0) 1
      (* n (fact (- n 1)))
  )
)

(fact 3)
```



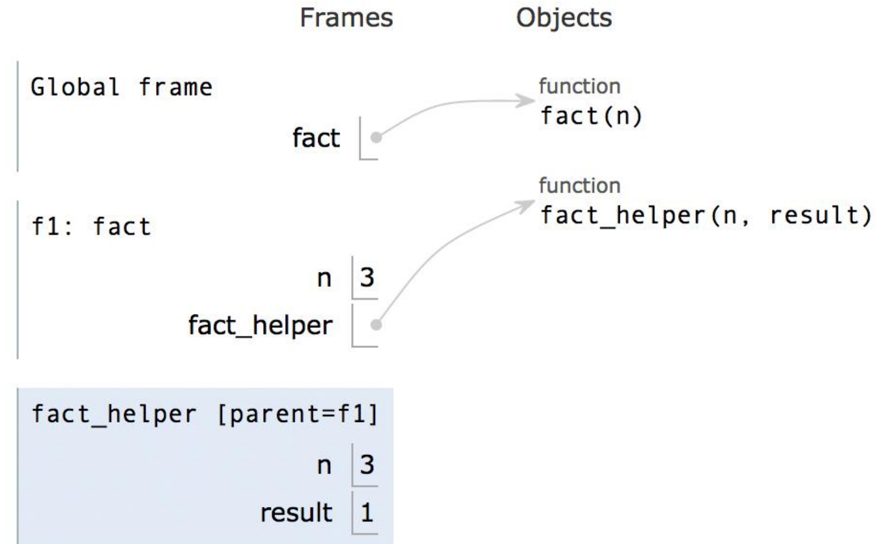
ORDINARY fact (8 / 8)

```
(define (fact n)
  (if (<= n 0) 1
      (* n (fact (- n 1)))
  )
)

(fact 3)
```



TAIL RECURSIVE fact (1 / 7)

```
(define (fact n)
  (define (fact-helper n result)
    (if (<= n 0) result
        (fact-helper (- n 1)
                     (* n result)))
  )
  (fact-helper n 1)
)

(fact 3)
```



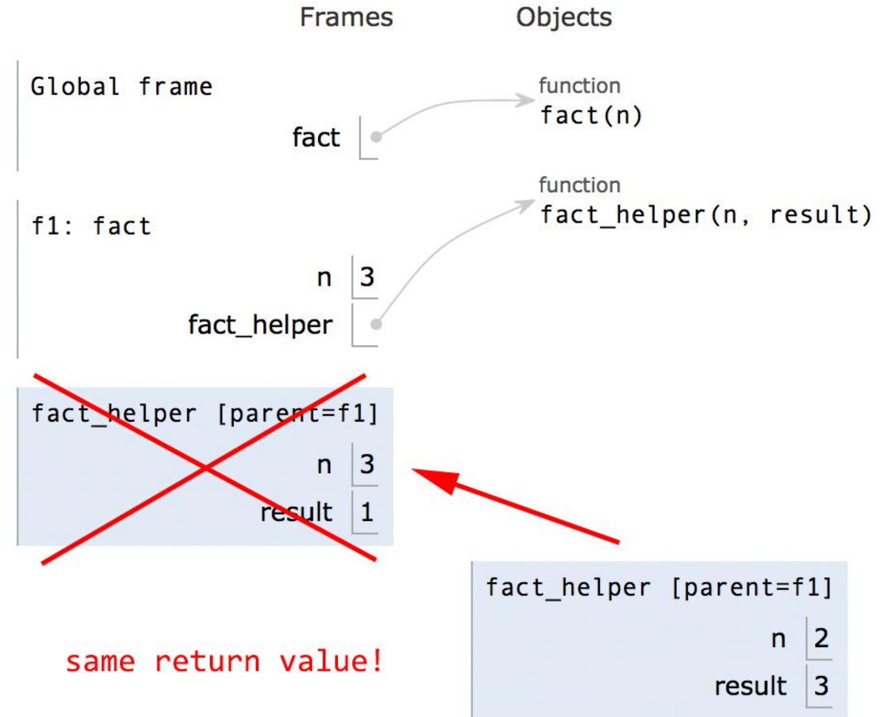
We've just made the first call to fact-helper, i.e. (fact-helper 3 1)

→ *Next call to be made*: (fact-helper (- 3 1) (* 3 1))

TAIL RECURSIVE fact (2 / 7)

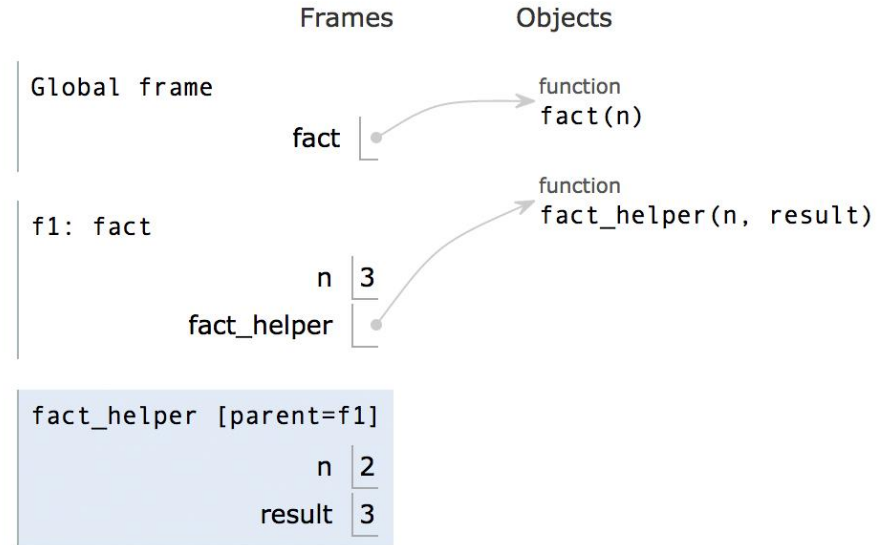
```
(define (fact n)
  (define (fact-helper n result)
    (if (<= n 0) result
        (fact-helper (- n 1)
                     (* n result)))
  )
  (fact-helper n 1)
)

(fact 3)
```

After making the (fact-helper 2 3) call, we don’t need any information from the crossed-out frame anymore.



TAIL RECURSIVE fact (3 / 7)

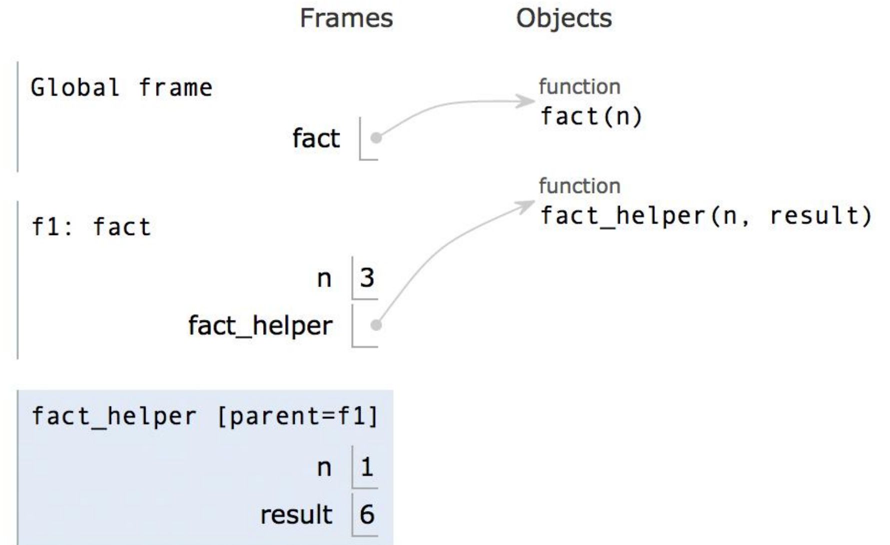
```
(define (fact n)
  (define (fact-helper n result)
    (if (<= n 0) result
        (fact-helper (- n 1)
                     (* n result)))
  )
  (fact-helper n 1)
)

(fact 3)
```



So we "replace" the frame with the new one – that of (fact-helper 2 3).
After all, (fact-helper 3 1) returns whatever (fact-helper 2 3) returns.

TAIL RECURSIVE fact (4 / 7)

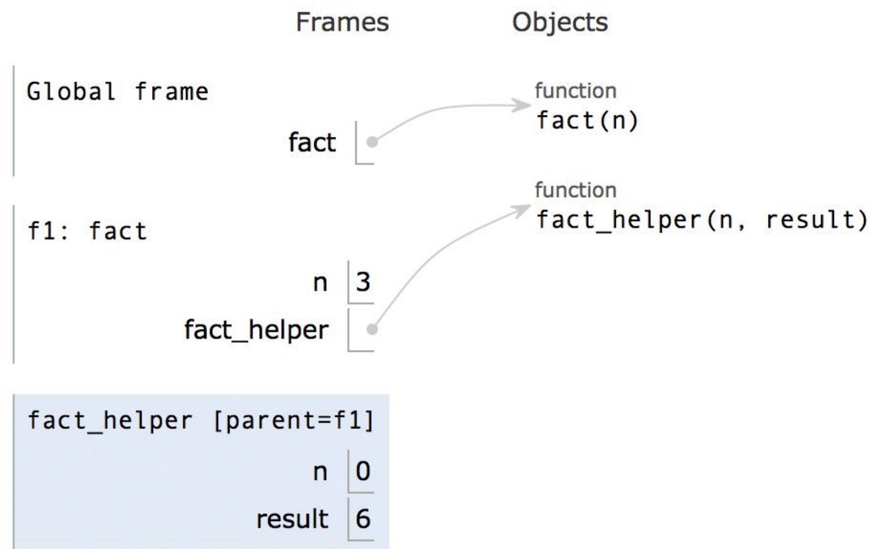
```
(define (fact n)
  (define (fact-helper n result)
    (if (<= n 0) result
        (fact-helper (- n 1)
                     (* n result)))
  )
  (fact-helper n 1)
)

(fact 3)
```



Continuing on, we replace the (fact-helper 2 3) frame with the (fact-helper 1 6) frame.

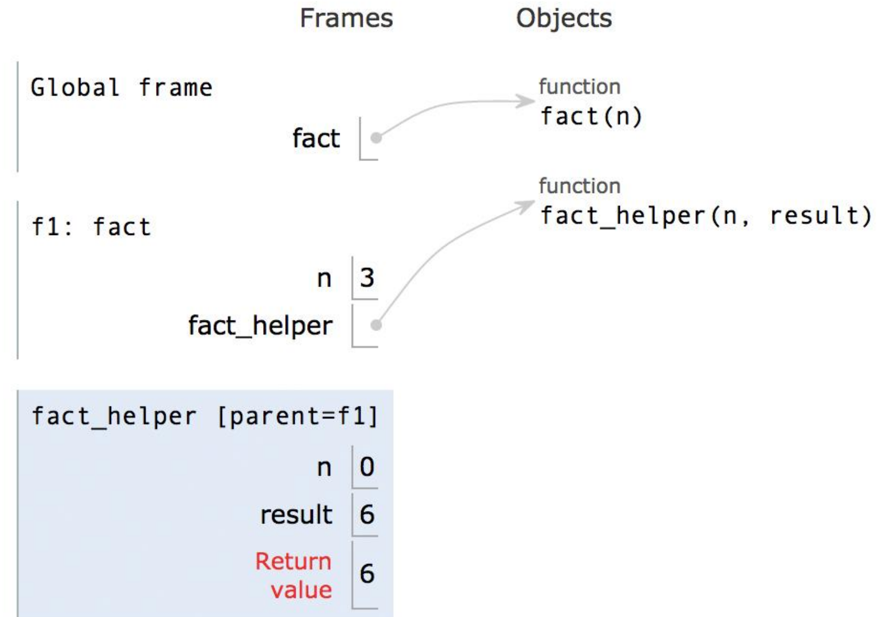
TAIL RECURSIVE fact (5 / 7)

```
(define (fact n)
  (define (fact-helper n result)
    (if (<= n 0) result
        (fact-helper (- n 1)
                     (* n result)))
  )
  (fact-helper n 1)
)

(fact 3)
```



TAIL RECURSIVE fact (6 / 7)

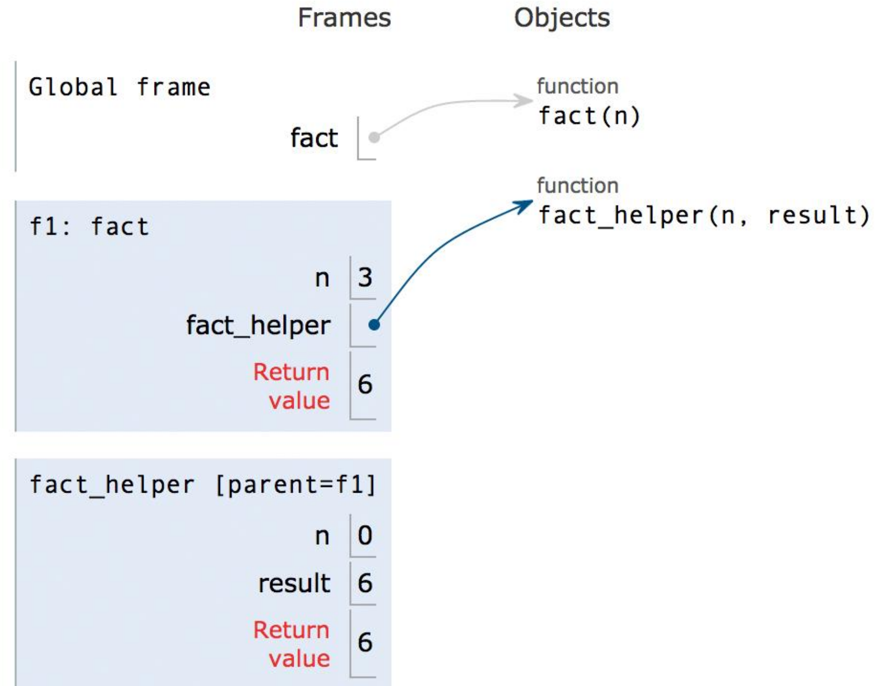
```
(define (fact n)
  (define (fact-helper n result)
    (if (<= n 0) result
        (fact-helper (- n 1)
                     (* n result)))
  )
  (fact-helper n 1)
)

(fact 3)
```



Here we have the return value for *all* of the frames… and we only ever needed to keep a single frame around at once!

TAIL RECURSIVE fact (7 / 7)

```
(define (fact n)
  (define (fact-helper n result)
    (if (<= n 0) result
        (fact-helper (- n 1)
                     (* n result)))
  )
  (fact-helper n 1)
)

(fact 3)
```



Tail contexts

[**Disclaimer**: As far as I'm concerned, just know that it's a "tail context" if it's the termination point of the function, i.e. *nothing else will happen in the function body after the code in the tail context is executed.*]

- **last expression** in
  - `define, lambda, let`
  - `begin`
  - `and, or`
- the **second** or **third expression** in an `if` form
- **last expression** of each **clause** in a `cond` form (but not predicates)

If the expression in any of those contexts is a procedure call, it's a tail call.
It might not be a recursive tail call, but it's a tail call.

Defining tail recursive procedures

Normally you just create a helper function and pass along extra information as arguments (for example, the return value you're building up). Then, as the base case, you can just return this value. And, more importantly, every recursive call can be a tail call!

```
(define (fact n)
  (define (fact-helper n result)
    (if (<= n 0) result
        (fact-helper (- n 1) (* n result))
    )
  )
  (fact-helper n 1)
)
```

Iteration → tail recursion

More notes from Kyle

- iteration is analogous to tail recursion
  - can mechanically turn an iterative function into a tail recursive function
  - can mechanically turn a tail recursive function into an iterative function

- Procedure:
  - each `while` loop becomes a tail recursive helper function
  - variables that change become parameters to the helper function
  - if you go back to the top of the `while` loop, make a tail call

- Python has efficient iteration, Scheme has efficient tail recursion.

Iteration → tail recursion

More notes from Kyle

```
def fib(n):
    i, curr, next = 0, 0, 1
    while i < n:
        curr, next = next, curr + next
        i = i + 1
    return curr

(define (fib n)
  (define (fib-iter i curr next)
    (if (< i n)
        (fib-iter (+ i 1) next (+ curr next))
        curr))
  (fib-iter 0 0 1))
```

Iteration → tail recursion

Literally all from Kyle here

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

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

Iteration → tail recursion

Final instance of copy-paste, direct from Kyle

```
def is_prime(n):
    d = 2
    while d < n:
        if n % d == 0:
            return True
        d += 1
    return False

(define (is-prime n)
  (define (is-prime-iter d)
    (cond ((= d n) #f)
          ((= 0 (modulo n d)) #t)
          (else (is-prime-iter (+ d 1)))))
  (is-prime-iter 2))
```