CS 360
Programming Languages
Day 9

Warmup

- Write a function **average** which finds the average of a list of numbers. Your function **must** use **foldr** and a **lambda** expression (for good practice).

- **Reminder:** (foldr func base list) is the signature of **foldr**

- **Reminder: func** takes two arguments, which can be thought of as the next **value** in the list and some kind of **acc**umulator (ex, a sum function using foldr would use the **acc** variable of **func** to accumulate the sum)

- **Reminder:** (lambda (arg1 arg2 …) (…)) is the signature of **lambda**

- **Hint:** your **foldr** call doesn’t have to do *everything*. Maybe it should just do one thing and you can work with that result to get the final answer?

Reminders

- Project 2 is due next week during office hours.
  - There is one place where you need to use **foldr** (and another where you can if you want to)
  - I used a bit of AI while coming up with solutions, but only for **very specific situations**

- **All projects must be submitted in-person during office hours.**

- Late assignments (ex, turning in Project 1 next week) will affect your "Timeliness" grade

- **Attendance/Participation** self-assessment is due every Friday (I don't grade it until the following Tuesday). It's called **Week X** on Canvas

Environments, Frames, and Bindings

- Racket (perhaps all programming languages) keep track of data structures called **environments**
- An **environment** is a sequence of **frames**
- A **frame** is a table of **bindings**
- A **binding** is a mapping from a **name** (ex, variable or function) to its **value**
  - A single frame can contain at most one binding per name
- Frames point to their **enclosing environment** (unless the frame is **global**)
- To quote SICP:
  - "The value of a variable with respect to an environment is the value given by the binding of the variable in the first frame in the environment that contains a binding for that variable." (pg 321)
  - What if a second frame tries to bind that variable? We'll try to answer that later!
- The global environment consists of a single frame (no enclosing environment)

Eval/Apply

- To evaluate a combination:
  - Evaluate the subexpressions
  - Apply the value of the operator subexpression to the values of the operand subexpressions

- A *procedure* is a pair, consisting of some code and a pointer to an environment. They are *always* created by evaluating a lamda-expression.

- **`(define (my-func arg1 …) (body))`** is called *syntactic sugar* for the actual code: **`(define my-func (lambda (arg1 …) (body)))`**

Call stack

While a program runs, there is a *call stack* of function calls that have started but not yet returned.

- Calling a function **`f`** pushes an instance of **`f`** on the stack.
- When a call to **`f`** to finishes, it is popped from the stack.
- Common to most programming languages.

These *stack frames* store information such as

- the values of arguments and local variables
- information about "what is left to do" in the function (further computations to do with results from other function calls)

Due to recursion, multiple stack frames may be calls to the same function.

The Dangers of Recursion

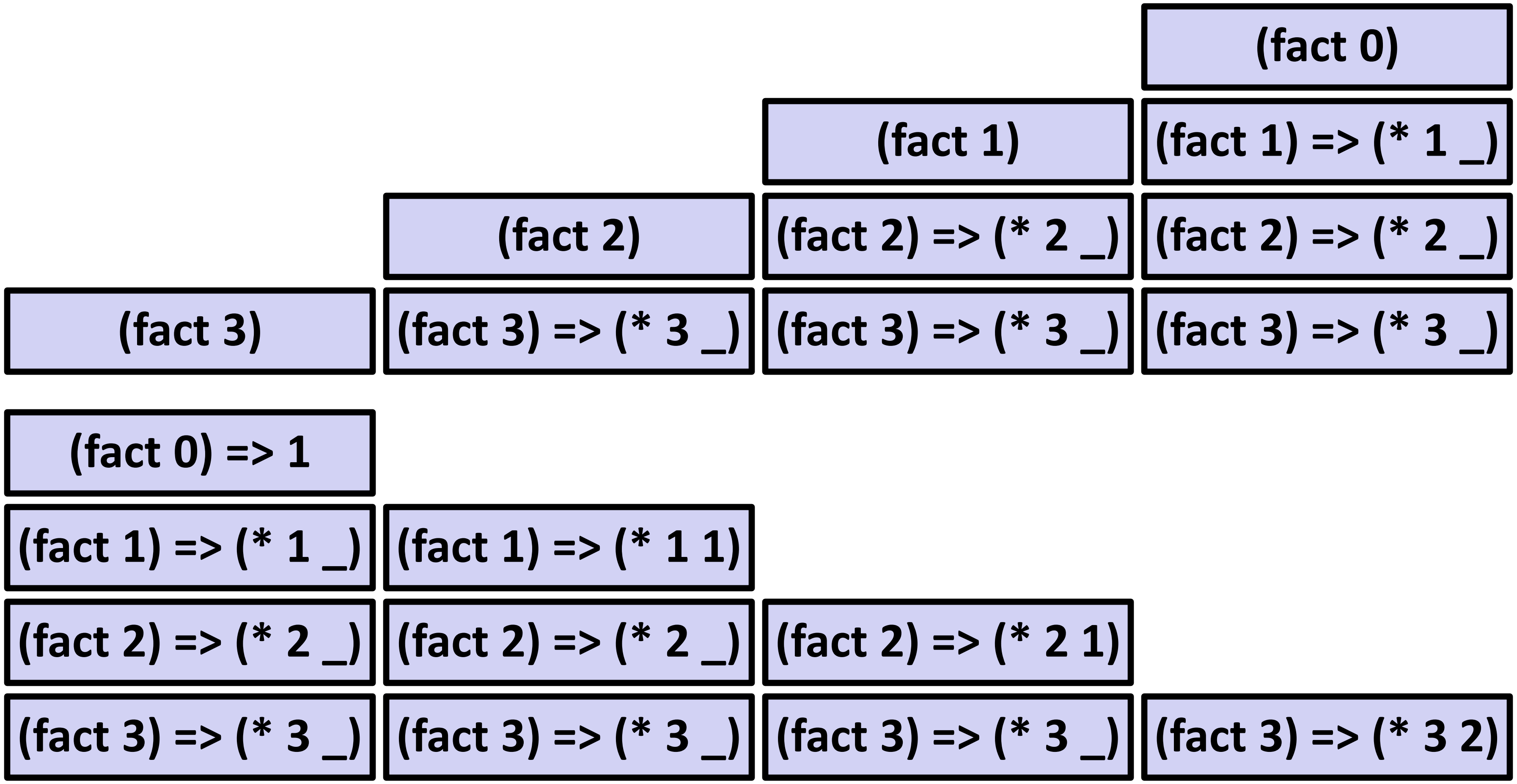
- Consider this definition of **factorial**:
  - `(define (factorial n) (* n (factorial (- n 1))))`

- Can you tell how many stack frames will be required by this function definition?

- Without unbounded memory, we can't rely on this function definition for *every* factorial calls.

Example

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



What's being computed

```
(fact 3)
     => (* 3 (fact 2))
     => (* 3 (* 2 (fact 1)))
     => (* 3 (* 2 (* 1 (fact 0))))
     => (* 3 (* 2 (* 1 1)))
     => (* 3 (* 2 1))
     => (* 3 2)
     => 6
```

The Solution: Tail Recursion

Compare

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

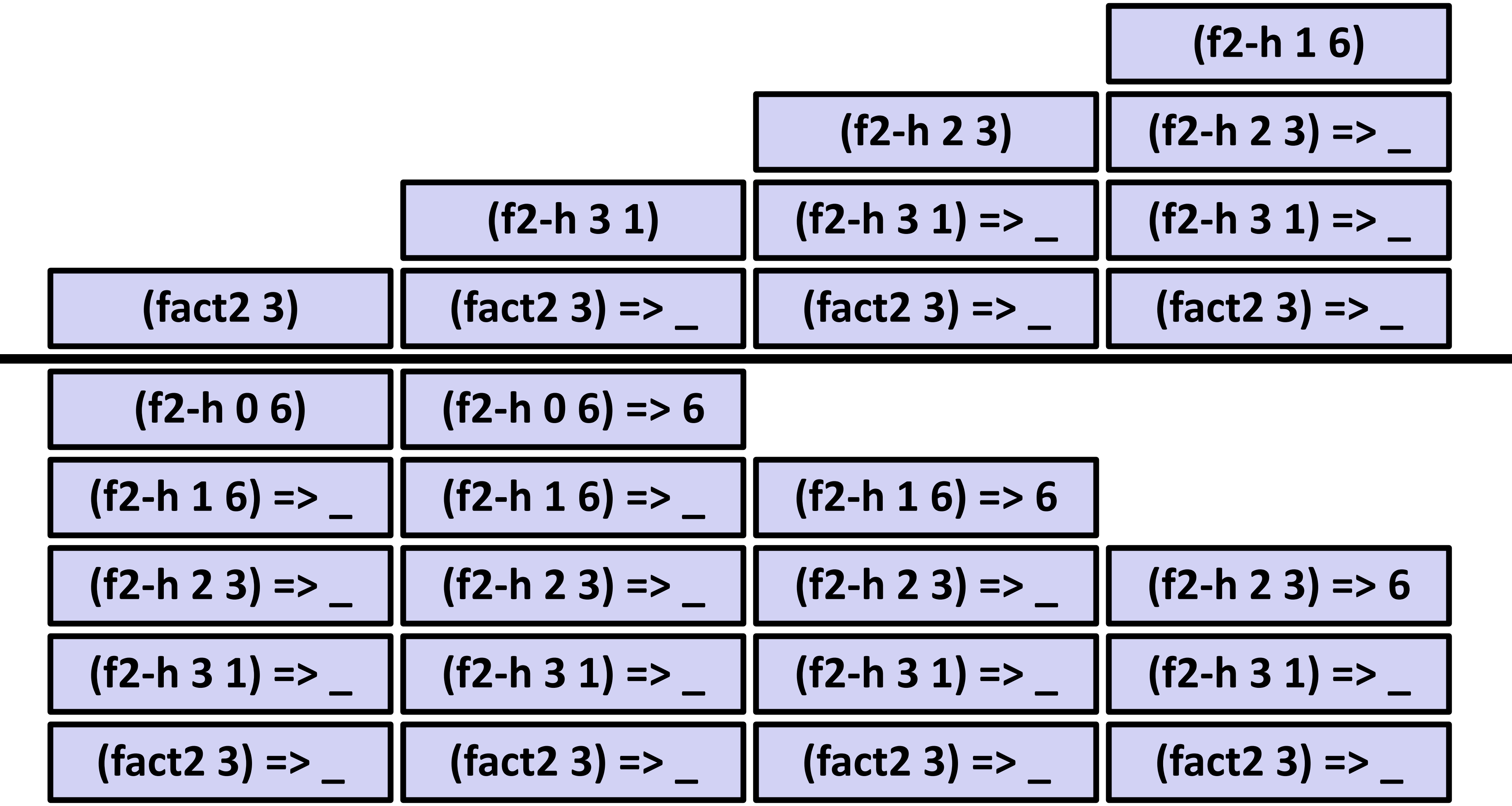
```
(define (fact2 n)

  (define (fact2-helper n acc)
      (if (= n 0) acc
          (fact2-helper (- n 1) (* acc n))))

  (fact2-helper n 1))
```

Still recursive, more complicated, but the result of recursive calls *is* the result for the caller (no remaining multiplication)

```
(define (fact2 n)
   (define (fact2-helper n acc)
        (if (= n 0) acc
                (fact2-helper (- n 1) (* acc n))))
   (fact2-helper n 1))
```



What's being computed

```
(fact2 3)
     => (fact2-helper 3 1)
     => (fact2-helper 2 3)
     => (fact2-helper 1 6)
     => (fact2-helper 0 6)
     => 6
```

An optimization

It is unnecessary to keep around a stack frame just so it can get a callee's result and return it without any further evaluation.

Racket recognizes these situations and treats them differently:

- Pop the caller *before* the call, allowing callee to *reuse* the same stack space.
- Uses same amount of memory as a loop.

Most, if not all functional language implementations do this optimization:

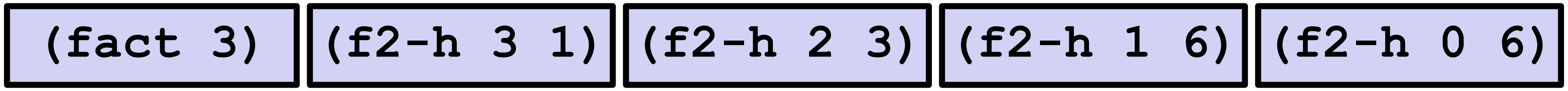
includes Racket, Scheme, LISP, ML, Haskell, OCaml…

What really happens on the call stack

```
(define (fact2 n)

  (define (fact2-helper n acc)
       (if (= n 0) acc
            (fact2-helper (- n 1) (* acc n))))

  (fact2-helper n 1))
```



Tail recursion

- In a functional language, rewriting functions to be ***tail-recursive*** can be much more efficient than "normal" recursive functions.

- In a ***tail-recursive*** function, all recursive calls must be the last thing the calling function does.
  - meaning no additional computation is done with the result of the callee (the recursive call).

- Functional languages will automatically optimize these tail-calls so they reuse the same stack space repeatedly.

Key to understanding tail recursion

- Most (singly-)recursive functions involve a recursive call and a computation involving the result of that recursive call.
  - e.g., for factorial, we multiply the result of the recursive call by *n*.
  - Normally we think about doing the **recursive call first** and the **computation second**.

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

- Tail-recursive functions do the **computation first** and the **recursive call second (last)**.

```
(define (fact2 n)
  (define (fact2-helper n acc)
      (if (= n 0) acc
          (fact2-helper (- n 1) (* acc n))))
  (fact2-helper n 1))
```

Methodology for tail-recursion

- Straight-forward to turn a "normally" recursive function into a tail-recursive one:
  - Create a helper function that takes an ***accumulator.***
  - Old base case's return value becomes initial accumulator value.
  - Final accumulator value becomes new base case return value.

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

Old base case's return value becomes initial accumulator value.

```
(define (fact2 n)

  (define (fact2-helper n acc)
      (if (= n 0) acc
          (fact2-helper (- n 1) (* acc n))))

  (fact2-helper n 1))
```

Final accumulator value becomes new base case return value.

Are these functions tail-recursive?

```
(define (get-nth lst n)
  (if (= n 0) (car lst)
    (get-nth (cdr lst) (- n 1))))

(define (good-max lst)
  (cond
    ((null? (cdr lst))
      (car lst))
    (#t
      (let ((max-of-cdr (good-max (cdr lst))))
        (if (> (car lst) max-of-cdr)
          (car lst) max-of-cdr)))))
```

Try these…

Write a tail-recursive sum function (i.e., a function that takes a list and computes the sum of all the elements).

Write a tail-recursive max function (i.e., a function that returns the largest element in a list).

Write a tail-recursive Fibonacci sequence function (i.e., a function that returns the n'th number of the Fibonacci sequence).

```
(fib 1) => 1
(fib 2) => 1
(fib 3) => 2
(fib 4) => 3
(fib 5) => 5
```

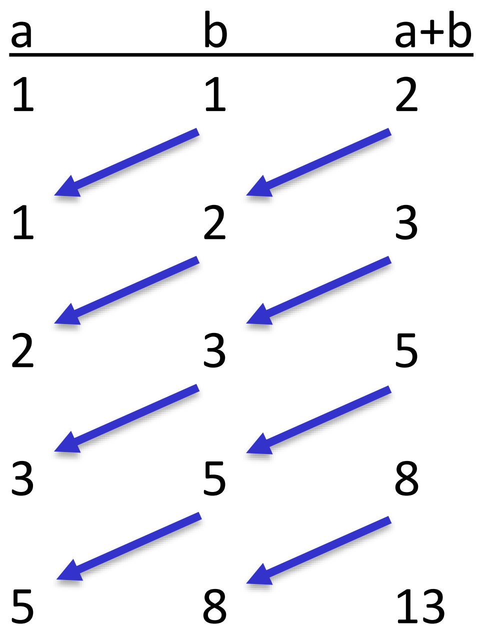
In general, **`(fib n) = (+ (fib (- n 1)) (fib (- n 2)))`**

```
(define (sum-tr lst)
  (define (sum-tr-helper lst acc)
    (if (null? lst) acc
      (sum-tr-helper (cdr lst) (+ (car lst) acc))))
  (sum-tr-helper lst 0))
```

```
(define (max-tr lst)
  (define (max-tr-helper lst max-so-far)
    (cond
      ((null? lst) max-so-far)
      ((> max-so-far (car lst))
          (max-tr-helper (cdr lst) max-so-far))
      (#t (max-tr-helper (cdr lst) (car lst)))))
  (maxtr-helper (cdr lst) (car lst)))
```

Fibonacci as a while loop

1 1 2 3 5 8 13



a	b	a+b
1	1	2
1	2	3
2	3	5
3	5	8
5	8	13

```
def fib(n):
  ctr = 1
  a = 1
  b = 2
  while ctr < n:
    ctr += 1
    old_a = a  # hold onto value of a
    a = b
    b = old_a + b
  return a
```

Fibonacci as a while loop

```
def fib(n):
  ctr = 1
  a = 1
  b = 2
  while ctr < n:
    ctr += 1
    old_a = a  # hold onto value of a
    a = b
    b = old_a + b
  return a
```

```
(define (fib-tr n)
  (define (fib-helper a b ctr)
    (if (= ctr n) a
        (fib-helper b (+ a b) (+ ctr 1))))
  (fib-helper 1 1 1))
```

```
def fib(n):
  ctr = 1
  a = 1
  b = 2
  while ctr < n:
    ctr += 1
    old_a = a  # hold onto value of a
    a = b
    b = old_a + b
  return a
```