Advanced Racket

Today…

- ❑ Map / Fold
- ❑ Tail Calls / Tail Recursion / Tail-Call Optimization
- ❑ Structs
- ❑ Pattern Matching
- ❑ QuasiQuoting / QuasiPatterns
- ❑ Building small interpreter with QuasiPatterns…
- ❑ (Maybe?) Contracts

Exam Grievances

- Grading mistakes sometimes happen: if you believe your exam has been graded **in error** (**Not** because you wish you had gotten more points!)…
  - Type up and print off argument describing the mistake;
  - Attach it to your exam, hand your exam back, I will return within a week
  - I must receive before one week from when you get exam
  - Please don’t ask me to look at your exam before doing this!

- Exam grades were exactly what I thought they would be
- No curve on Midterm 1
- For the rest of the semester, let's do **real** PL
  - By which I mean, **building** languages
  - https://www.youtube.com/watch?v=dht_3NziwSw

Goal in the rest of the class:

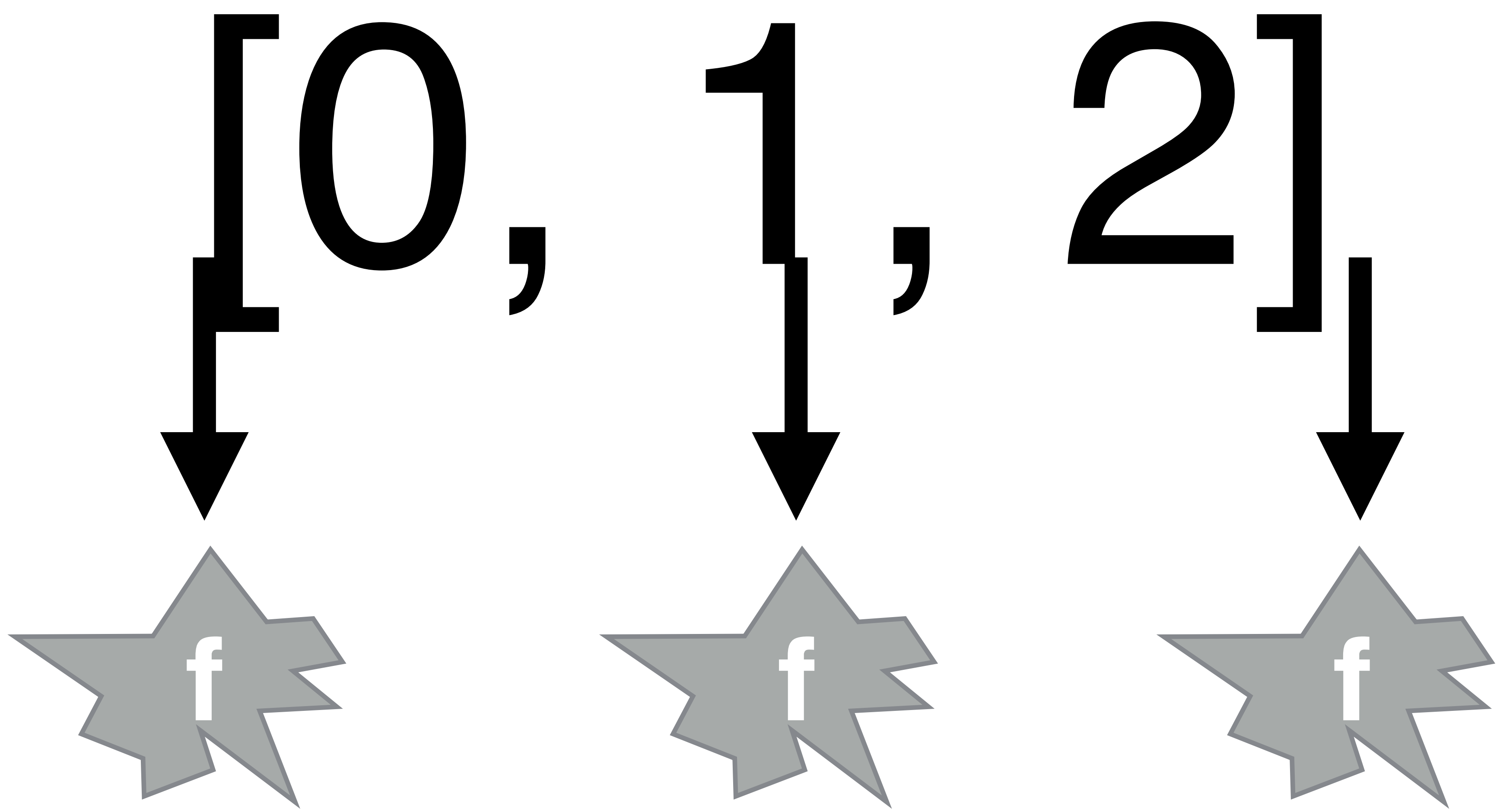
Write interpreters and compilers for (a significant subset of) Racket

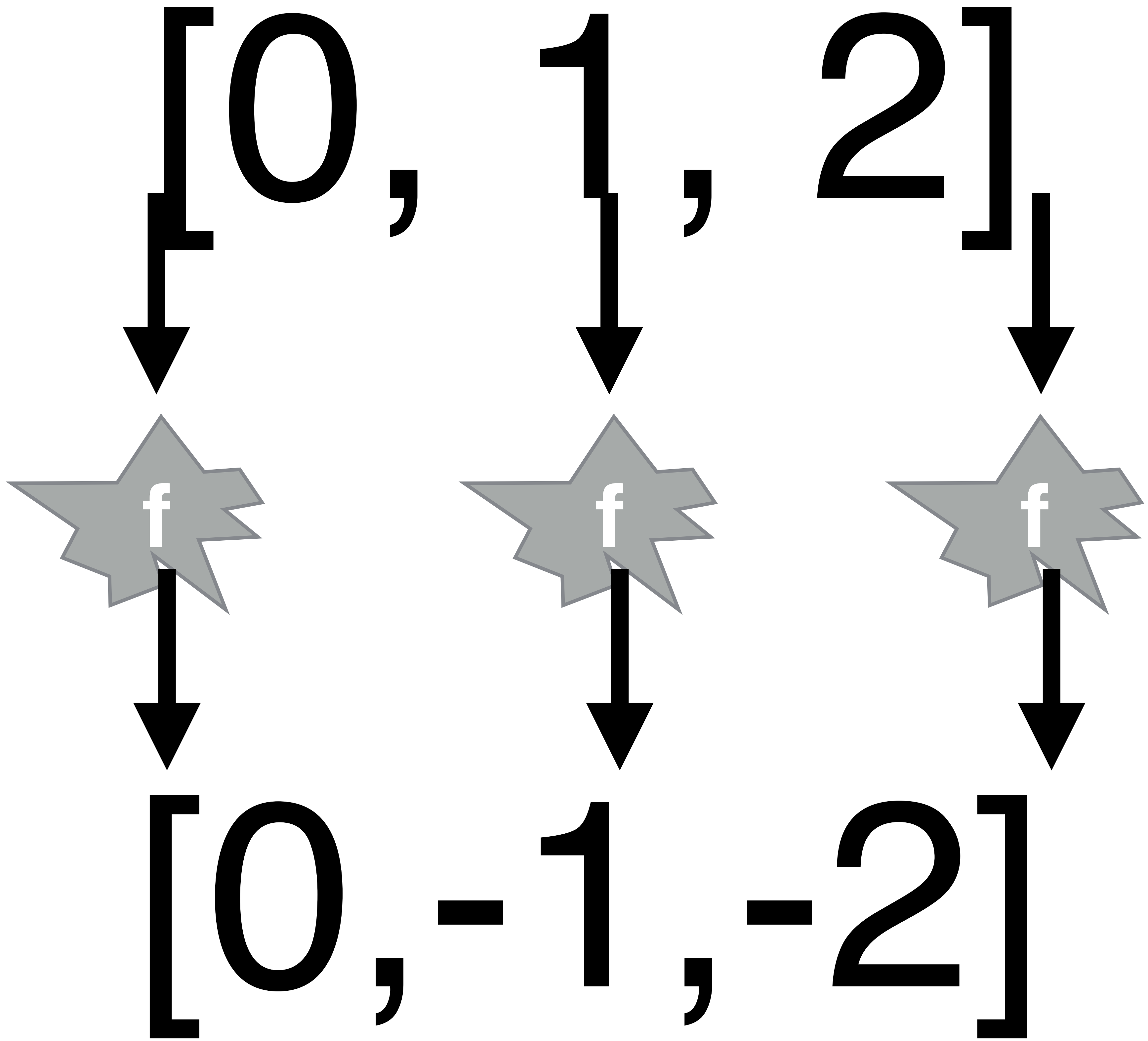
Higher-Order Functions

```
> (map (lambda (str) (string-ref str 0)) '("ha" "ha"))
'(#\h #\h)
```

(map f l) takes a function f and applies f to each element of l

[0, 1, 2]


[0, 1, 2]
f
f
f


[0, 1, 2]
f
f
f
[0,-1,-2]

Folding

Fold "accumulates" a value by iterating over a list

```
(define (fold-right f init seq)
  (if (null? seq)
      init
      (f (car seq)
          (fold-right f init (cdr seq)))))
```

```
(define (fold-right f init seq)
  (if (null? seq)
      init
      (f (car seq)
         (fold-right f init (cdr seq)))))
```

(6 minutes) Calculate the following:

```
(fold-right (lambda (x y) (+ x y)) 0 ‘(1 2 3))

(fold-right (lambda (x y) (- x y)) 0 ‘(1 2 3))

(fold-right (lambda (x y) (cons x y)) ‘() ‘(1 2 3))

(fold-right (lambda (x y) (append y (list x)))
            ‘()
            ‘(1 2 3))
```

Tail Recursion

Tail recursion is the way you make recursion fast in functional languages

Anytime I'm going to recurse more then 10k times, I use tail recursion

(I also do it because it's a fun mental exercise)

Tail Recursion

A function is *tail recursive* if **all** recursive calls are in *tail position*

A call is in tail position if it is the "last thing to happen" in a function

"This will definitely be on the exam, so don't ignore it." — Kris

```
(define (factorial x)
    (if (equal? x 0)
        1
        (* (factorial (- x 1)) x)))
```

This is **not** a tail call: its return value is used by *

(We call such calls **direct-style** calls)

```
(define (factorial x acc)
    (if (equal? x 0)
        acc
        (factorial (- x 1) (* acc x))))
```

This **is** a tail call: it it the last thing done by the function

Note that we “thread through” an accumulator

```
(define (factorial x acc)
    (if (equal? x 0)
        acc
        (factorial (- x 1) (* acc x))))
```

This **is** a tail call: it it the last thing done by the function

Note that we “thread through” an accumulator

```
(define (factorial x acc)
    (if (equal? x 0)
        acc
        (factorial (- x 1) (* acc x))))
```

This **is** a tail call: it it the last thing done by the function

But now factorial has an extra arg: how do I call it?

Within a function body, a **tail call** is a call that never “returns”

A **function** is **tail-recursive** if all recursive calls are tail calls

So why is tail-recursion useful?

```
(define (factorial x)
    (if (equal? x 0)
        1
        (* (factorial (- x 1)) x)))
```

Stack frames

```
(factorial 2)
(* (factorial 1) 2)
(* (* (factorial 0) 1) 2)
(* (* 1 1) 2)
```

The stack holds the “partial results”

```
(define (factorial x acc)
    (if (equal? x 0)
        acc
        (factorial (- x 1) (* acc x))))
```

Stack frames

```
(factorial 2 1)
(factorial 1 2)
(factorial 0 2)
2
```

Stack just propagates final value to original callsite

```
(factorial 2 1)
```

A tail recursive function never “needs” to use the stack

```
(define (factorial x acc)
    (if (equal? x 0)
        acc
        (factorial (- x 1) (* acc x))))
; .. Later
(factorial 2 1)
```

```
>factorial 2 1
```

factorial 2 1

```
(define (factorial x acc)
   (if (equal? x 0)
       acc
       (factorial (- x 1) (* acc x))))
; .. Later
(factorial 2 1)
```

```
>factorial 2 1
      >factorial 1 2
```

factorial 2 1

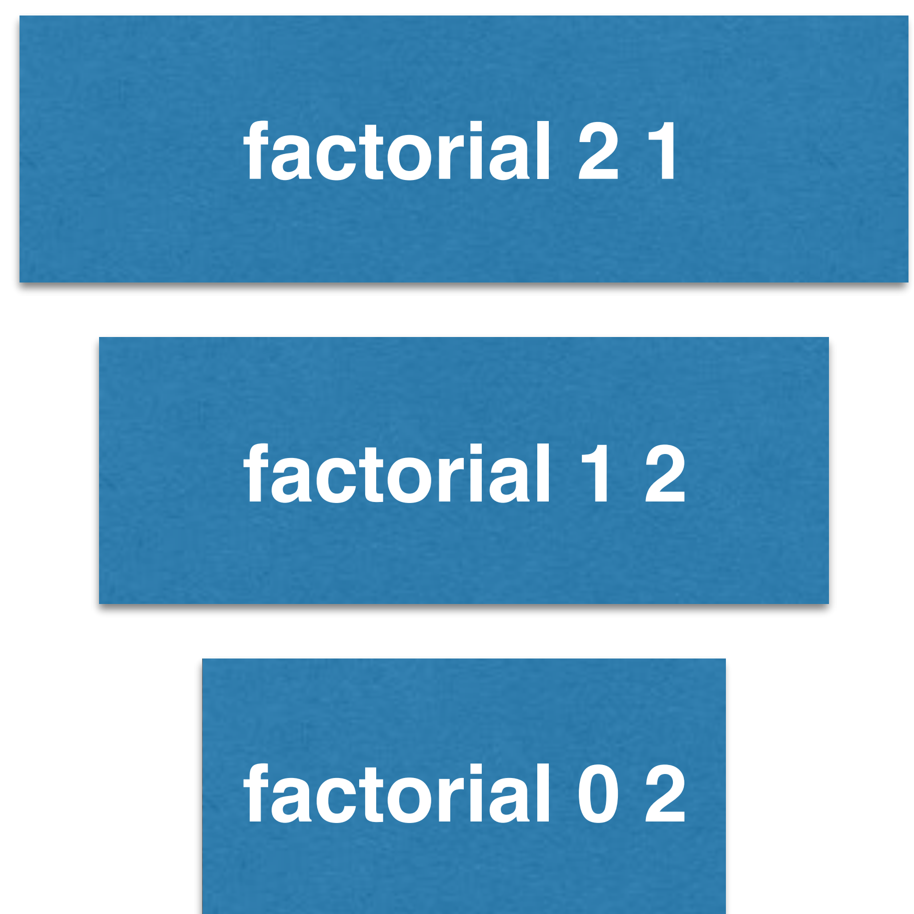
factorial 1 2

```
(define (factorial x acc)
    (if (equal? x 0)
        acc
        (factorial (- x 1) (* acc x))))
; .. Later
(factorial 2 1)
```

>factorial 2 1

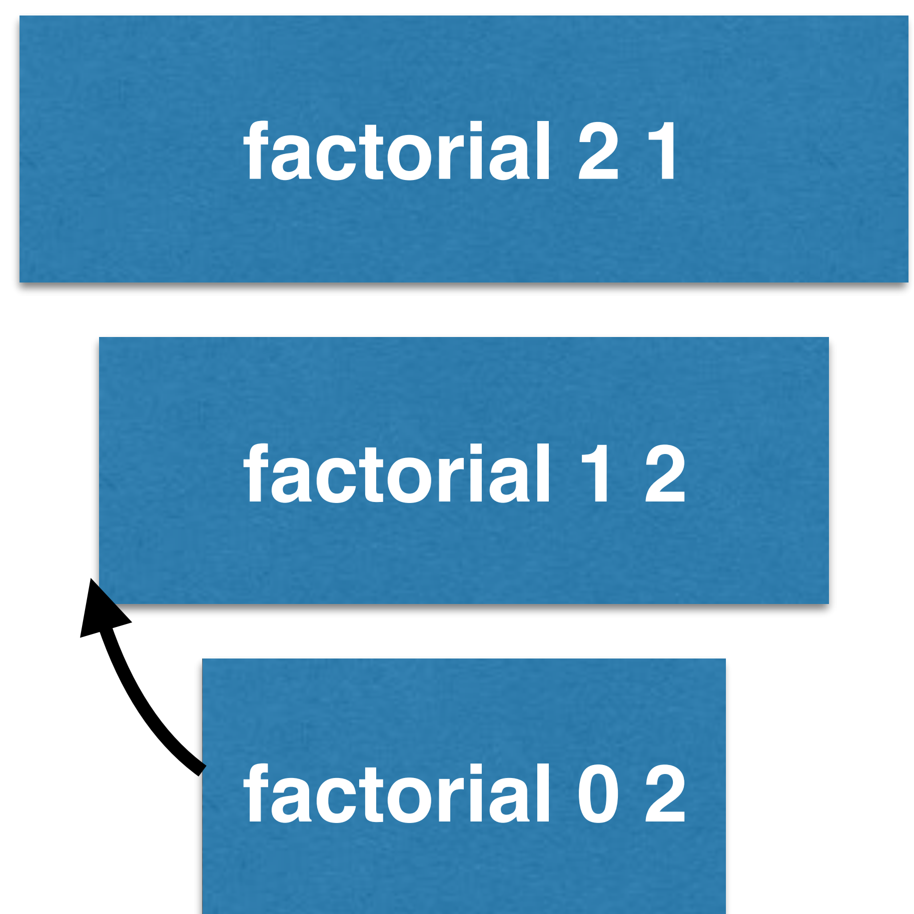
>factorial 1 2

>factorial 0 2



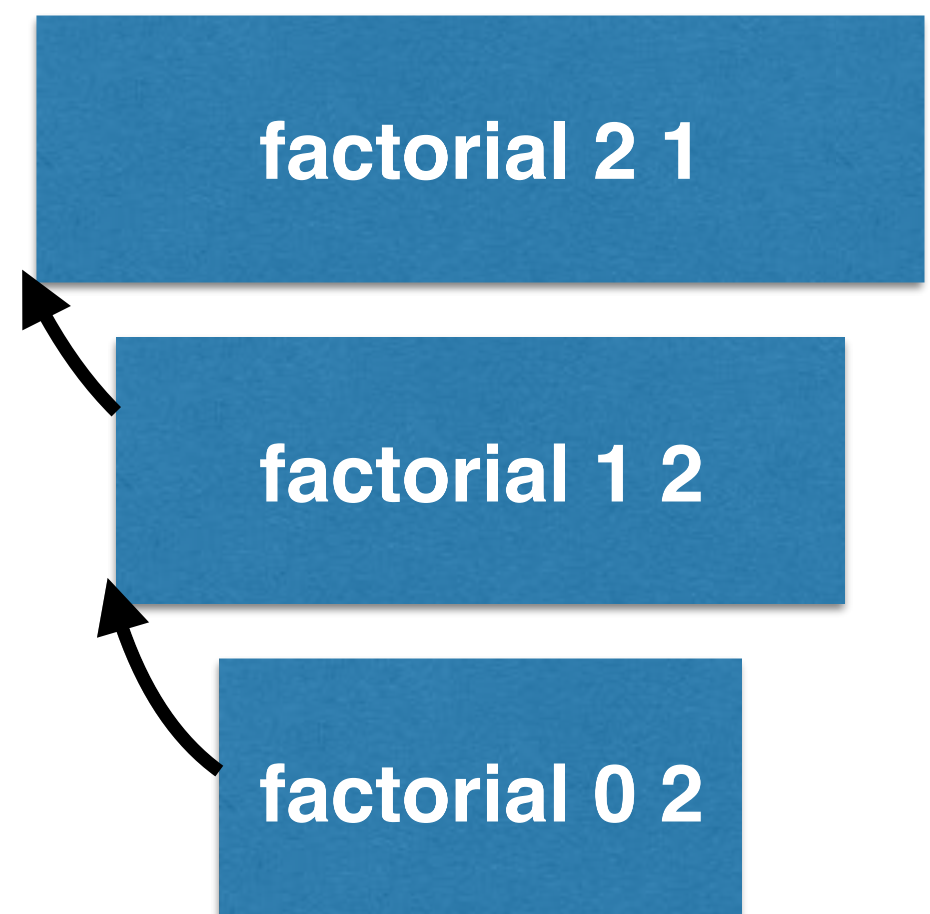
```
(define (factorial x acc)
    (if (equal? x 0)
        acc
        (factorial (- x 1) (* acc x))))
; .. Later
(factorial 2 1)
```

```
>factorial 2 1
    >factorial 1 2
        >factorial 0 2
```



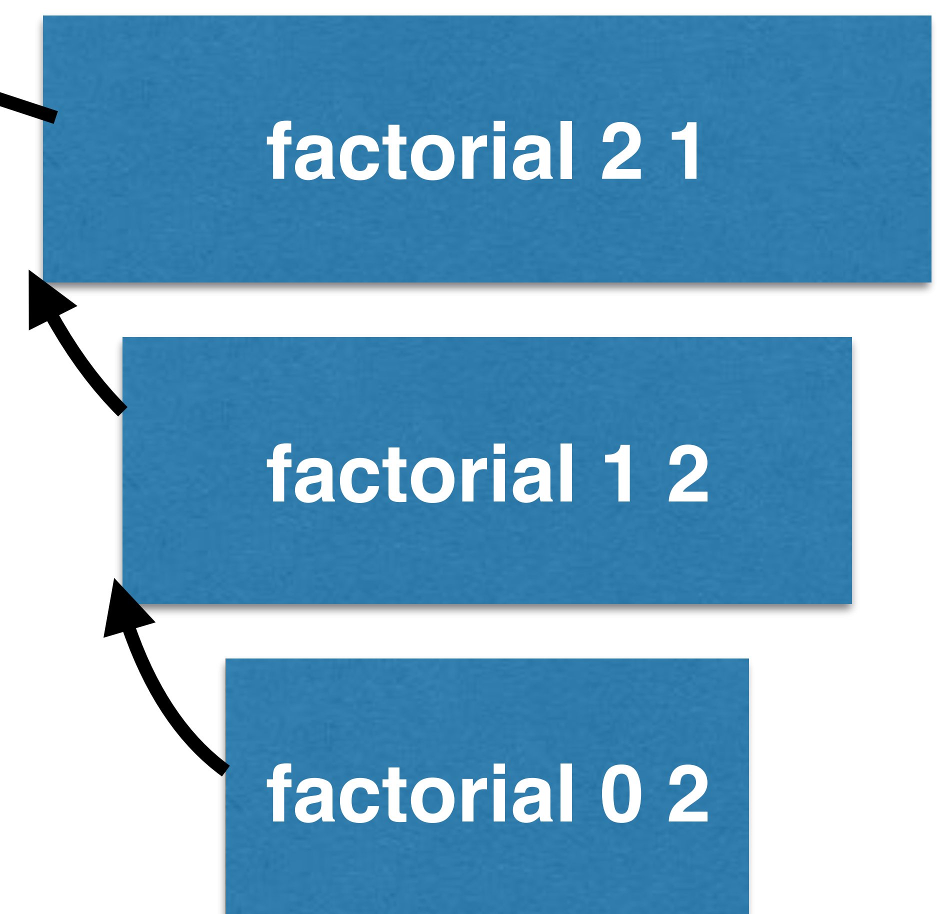
```
(define (factorial x acc)
    (if (equal? x 0)
        acc
        (factorial (- x 1) (* acc x))))
; .. Later
(factorial 2 1)
```

```
>factorial 2 1

        >factorial 1 2

                >factorial 0 2
```



```
(define (factorial x acc)
    (if (equal? x 0)
        acc
        (factorial (- x 1) (* acc x))))
; .. Later
(factorial 2 1)
```

```
>factorial 2 1
        >factorial 1 2
              >factorial 0 2
```



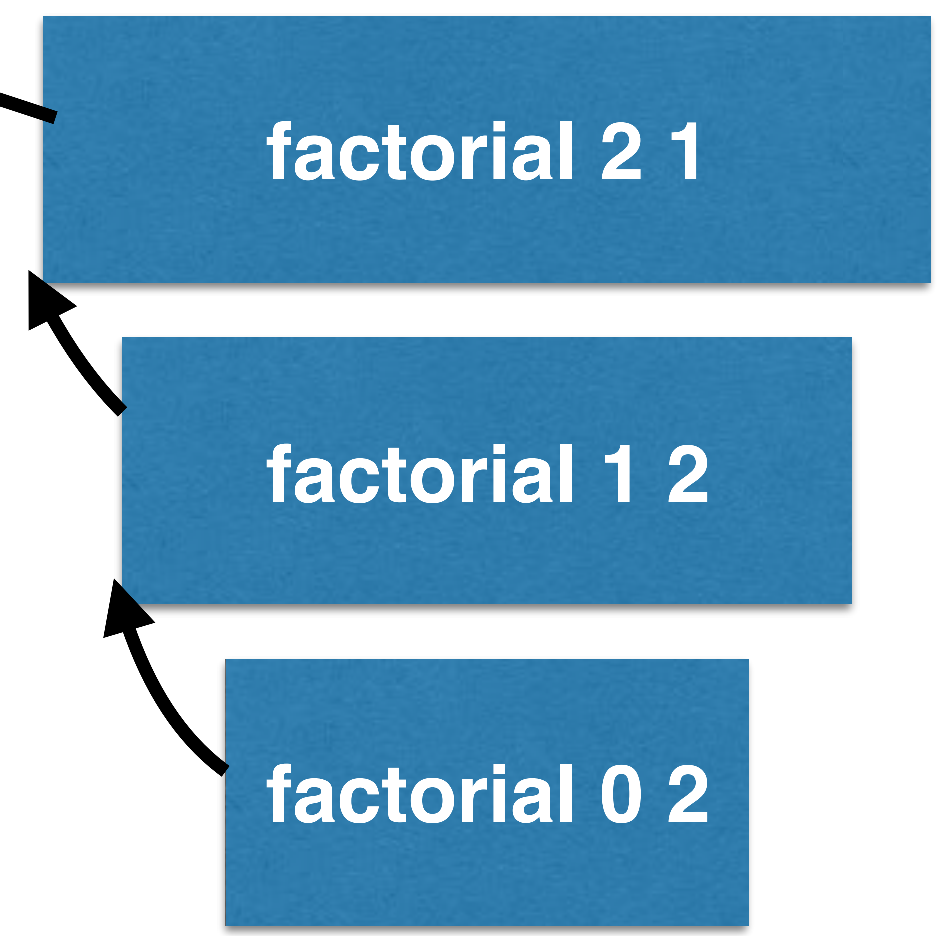
```
(define (factorial x acc)
    (if (equal? x 0)
        acc
        (factorial (- x 1) (* acc x))))
; .. Later
(factorial 2 1)
```

I don’t need the stack at all!

>factorial 2 1

>factorial 1 2

>factorial 0 2



Insight: w/ tail recursion, stack **only used** to propagate results to original caller!

A tail recursive function never “needs” to use the stack

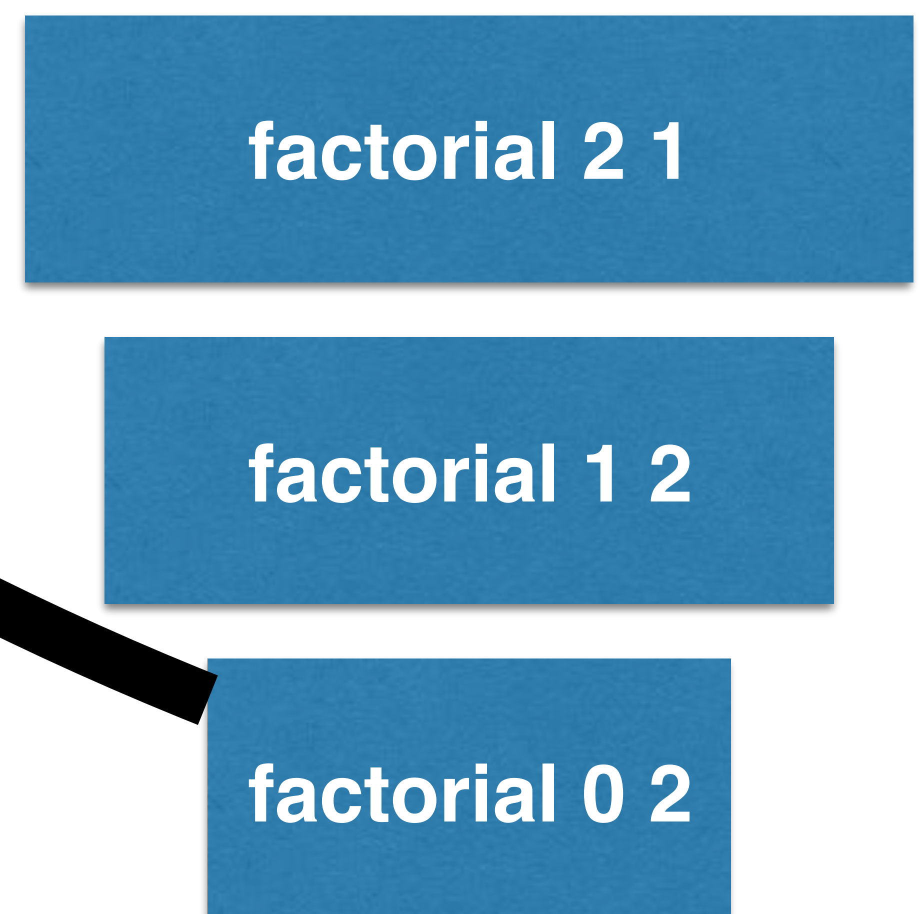
So functional compilers **optimize** tail calls

Basically: tail-recursive functions compile to loops

(This is called **tail-call optimization** or **TCO**)

```
(define (factorial x acc)
    (if (equal? x 0)
        acc
        (factorial (- x 1) (* acc x))))
; .. Later
(factorial 2 1)
```

```
>factorial 2 1
     >factorial 1 2
          >factorial 0 2
```



```scheme
(define (factorial x acc)
    (if (equal? x 0)
        acc
        (factorial (- x 1) (* acc x))))
```

```c
int factorial(int x, int acc) {
  if (x == 0)
    return acc;
  else
    return factorial(x-1,x*acc);
}
```

No TCO

```c
int factorial(int x, int acc) {
  while (true) {
    if (x == 0)
      return acc;
    acc = x*acc;
    x = x-1;
  }
}
```

Tail-Call Optimization

This is (conceptually) what the compiler will do

Many other languages *also* support TCO

- **Guaranteed** by the Scheme (e.g., r6rs) standard
  - "Implementations of Scheme must be *properly tail-recursive*."
- C/C++: Often. Varies by compiler / options
- Python does **not** on principle
  - Stack traces become less helpful!
- JavaScript does depending on browser / engine
  - Safari (March 2018) supports TCO in JS:
    - https://stackoverflow.com/questions/37224520/are-functions-in-javascript-tail-call-optimized
    - Active development in Chrome:
      - https://bugs.chromium.org/p/v8/issues/detail?id=4698

In general, it's a bit tricky to convert a non-tail-recursive function to being tail recursive…

Tail recursion for λ and profit…

To make a function tail recursive…

- add an extra accumulator argument
- that tracks the result you're building up
- then return the result
- might have to use more than one extra arg
- Call function with base case as initial accumulator

This isn't the only way to do it, just a nice trick
that usually results in clean code…

```
(define (factorial x)
    (if (equal? x 0)
        1
        (* (factorial (- x 1)) x)))
```

```
(define (factorial-tail x acc)
    (if (equal? x 0)
        acc
        (factorial-tail (- x 1) (* acc x))))
(define (factorial x) (factorial-tail x 1))
```

Exercise: translate fib into tail-recursive style

```
(define (fib n)
    (cond
      [(= n 0) 1]
      [(= n 1) 1]
      [else (+ (fib (- n 1)) (fib (- n 2)))]))
```

```
(define (max-of-list l)
  (cond [(eq? (length l) 1) 1]
        [(empty? l) (raise 'empty-list)]
        [else (max (first l) (max-of-list (rest l))
)]))
```

Write this as a tail-recursive function

Which of these is tail recursive?

```
(define (fold-right f init seq)
  (if (null? seq)
      init
      (f (car seq)
         (fold-right f init (cdr seq)))))

(define (fold-left f init seq)
  (if (null? seq)
      init
      (fold-left f
                 (f (car seq) init)
                 (cdr seq))))
```

Upshot: if you can, use foldl

```
(define (fold-right f init seq)
  (if (null? seq)
      init
      (f (car seq)
          (fold-right f init (cdr seq)))))

(define (fold-left f init seq)
   (if (null? seq)
       init
       (fold-left f
                  (f (car seq) init)
                  (cdr seq))))
```

```
(define (concat-strings l)
  (foldl (lambda (next_element accumulator)
         (string-append next_element accumulator))
         ""
         …))
```

Question: what goes in … to define `concat-strings`?

Structures, Pattern Matching, and Contracts

5 Programmer-Defined Datatypes

New datatypes are normally created with the `struct` form, which is the topic of this chapter. The class-based object system, which we defer to Classes and Objects, offers an alternate mechanism for creating new datatypes, but even classes and objects are implemented in terms of structure types.

5.1 Simple Structure Types: `struct`

To a first approximation, the syntax of `struct` is

```
(struct struct-id (field-id ...))
```

Examples:

```
(struct posn (x y))
```

The `struct` form binds *struct-id* and a number of identifiers that are built from *struct-id* and the *field-id*s:

- *struct-id* : a *constructor* function that takes as many arguments as the number of *field-id*s, and returns an instance of the structure type.

  Example:

  ```
  > (posn 1 2)
  #<posn>
  ```

- *struct-id*? : a *predicate* function that takes a single argument and returns `#t` if it is an instance of the structure type, `#f` otherwise.

  Examples:

Use **struct** to define a new datatype

```
(struct leaf (elem))

(struct tree (value left right))
```

Copy these

```
(struct leaf (elem) #:transparent)

(struct tree (value left right))
```

```
(leaf 23)

(tree 12 (empty-tree) (leaf 23))
```

Racket automatically generates helpers…

`tree?`

`tree-left`

`tree-right`

Write `max-of-tree`

Use the helpers

Pattern matching

Pattern matching allows me to tell Racket the "shape" of what I'm looking for

Manually pulling apart data structures is laborious

```
(define (max-of-tree t)
  (match t
    [(leaf e) e]
    [(tree v _ (empty-tree)) v]
    [(tree _ _ r) (max-of-tree r)]))
```

Variables are bound in the match, refer to in body

```
(define (max-of-tree t)
  (match t
    [(leaf e) e]
    [(tree v _ (empty-tree)) v]
    [(tree _ _ r) (max-of-tree r)]))
```

Note: match struct w/ (name params…)

```
(define (max-of-tree t)
  (match t
    [(leaf e) e]
    [(tree v _ (empty-tree)) v]
    [(tree _ _ r) (max-of-tree r)]))
```

Define `is-sorted`

Can match a list of x's

```
(list x y z …)

(1 2 3 4)

x = 1 y = 2 z = ‘(3 4)
```

Can match cons cells too…

```
(cons x y)
```

Variants include things like match-let

QuasiQuoting

Quotes build data

`‘(1 2 3)` `‘(1 (2 3) a)`

What if you want to build a list like this…

`‘(1 2 x)`

Where x gets substituted to whatever x is

```
(define x 3)
`(1 (2 3) ,a)
```

QuasiQuote (the backtick) **begins** building a datum: any time it hits an *unquote* (comma) it evaluates the expression

QuasiPatterns

We can also use quasiquoting in a match pattern
We call this a quasi*pattern*

It turns out this lets us build an
implementation of a **little language**!

```
(define (interpret-binary-arith e)
  (match e
    [`(+ ,e1 ,e2) (+ (interpret-binary-arith e1)
                     (interpret-binary-arith e2))]
    [`(- ,e1 ,e2) (- (interpret-binary-arith e1)
                     (interpret-binary-arith e2))]
    [(? number? n) n]
    [else (error "bad expression...")]))
```

Exercise: call `interpret-binary-arith` on the following…

```
3        (+ 2 3)

(+ (- (+ 2 3) 5) (+ 1 (- 2 3)))
```

Quiz

What's the difference between the following two expressions?

```
(interpret-binary-arith
  (+ (- (+ 2 3) 5) (+ 1 (- 2 3))))

(interpret-binary-arith
  '(+ (- (+ 2 3) 5) (+ 1 (- 2 3))))
```

Answer: in one we're cheating. We're not really using our interpreter, we're just using Racket

Contracts

```scheme
(define (reverse-string s)
  (list->string (reverse (string->list s))))
```

Write out the call and return type of this for yourself

```
(define (factorial i)
  (cond
    [(= i 1) 1]
    [else (* (factorial (- i 1)) i)]))
```

What are the call / return types?

What is the pre / post condition?

```
(define (gt0? x) (> x 0))
```

```
(define/contract (factorial i)
  (-> gt0? gt0?)
  (cond
    [(= i 1) 1]
    [else (* (factorial (- i 1)) i)]))
```

Now in tail form…

```
(define (fac-tail i)
  (letrec ([h (lambda (i acc)
                (cond
                  [(= i 0) acc]
                  [else (h (- i 1) (* acc i))]))])
    (h i 1)))
```

Now, let's say I want to say it's equal to factorial…

```
(define/contract (fac-tail i)
  (->i ([x (>=/c 0)])
       [result (x) (lambda (result) (= (factorial x) result))])
  (letrec ([h (lambda (i acc)
                (cond
                  [(= i 0) acc]
                  [else (h (- i 1) (* acc i))]))])
    (h i 1)))
```

```
(->i ([x (>=/c 0)])
     [result (x) (lambda (result) (= (factorial x) result))])
```

```
(define/contract (reverse-string s)
  (-> string? string?)
  (list->string (reverse (string->list s))))
```

```
(define/contract (reverse-string s)
  (-> string? string?)
  (list->string (reverse (string->list s))))
```

```
(<=/c 2)
```

`<=/c` takes an argument `x`, returns a function f that takes an argument `y`, and `f(y) = #t` if `x <= y`

`<=/c` takes an argument `x`, returns a function f that takes an argument `y`, and `f(y) = #t` if `x < = y`

(Note: <=/c is also doing some bookeeping, but we won't worry about that now.)

Challenge: write <=/c

Three stories

```
(define/contract (call-and-concat f s1 s2)
  (-> (-> string? string?) string? string? string?)
  (string-append (f s1) (f s2)))
```

```
(define (reverse-string s)
  (list->string (reverse (string->list s))))
```

Scenario: you call call-and-concat with reverse

Scenario: you call call-and-concat with reverse, 12, and “12"

Now define

```
(define/contract (call-and-concat f s1 s2)
  (-> (-> string? string?) string? string? string?)
  (length (string-append (f s1) (f s2))))
```

Now define

```
(define/contract (call-and-concat f s1 s2)
  (-> (-> string? string?) string? string? string?)
  (length (string-append (f s1) (f s2))))
```

What went wrong?

Now define

```
(define/contract (call-and-concat f s1 s2)
  (-> (-> string? string?) string? string? string?)
  (length (string-append (f s1) (f s2))))
```

What went wrong?

Who is to blame?