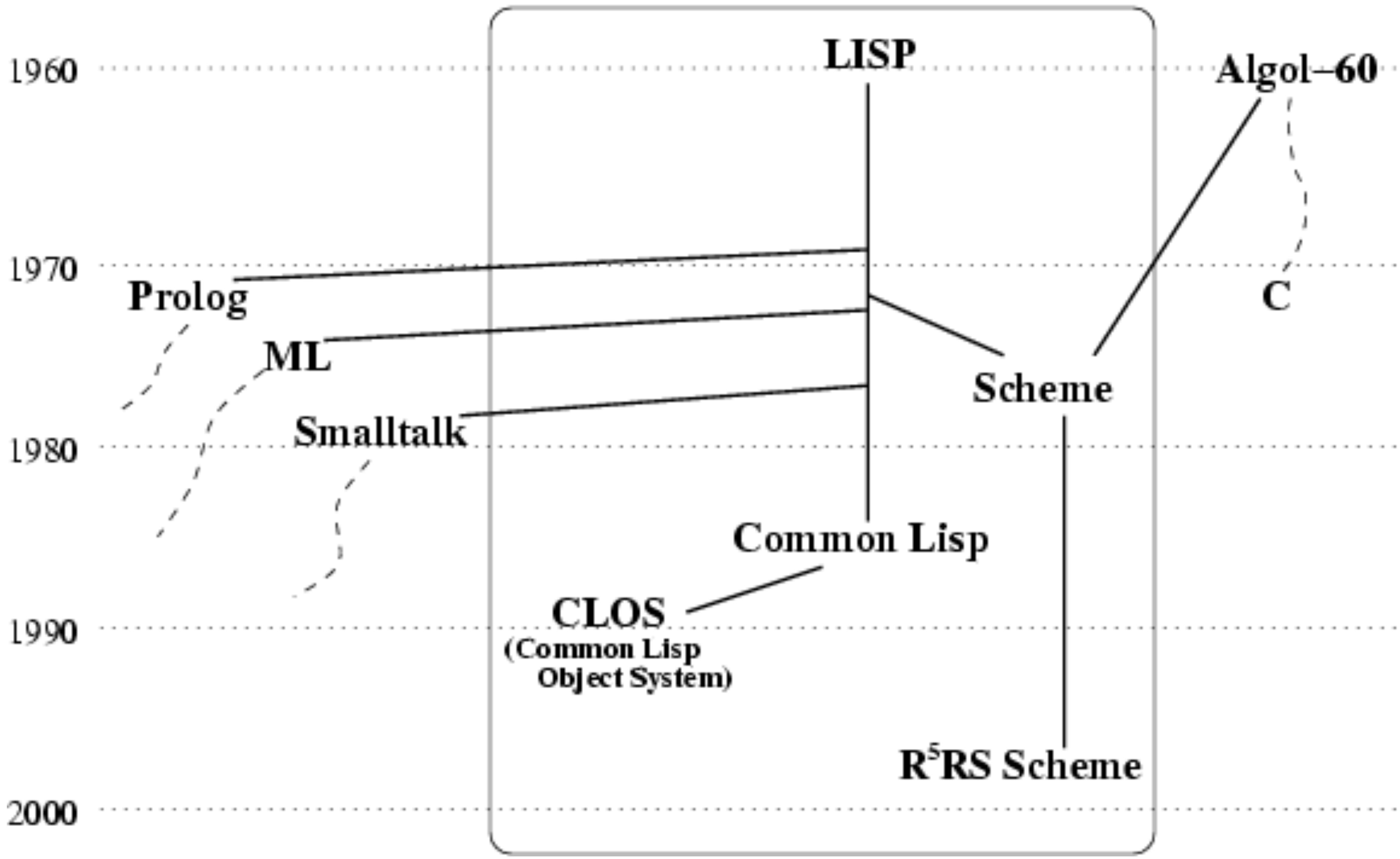
Introducing Racket

λ

A brief tour of history…



1960
1970
1980
1990
2000
LISP
Algol–60
Prolog
C
ML
Scheme
Smalltalk
Common Lisp
CLOS
(Common Lisp
Object System)
R5RS Scheme

We wanted a language that allowed **symbolic manipulation**

Scheme

The key to understanding LISP is understanding S-Expressions

Racket

List of either **atoms** or **S-expressions**

(this (is an) (s) expression)

List of either **atoms** or **S-expressions**

(this (is an) (s) expression)

List of either **atoms** or **S-expressions**

(this (is an) (s) expression)

atom

List of either **atoms** or **S-expressions**

(this (is an) (s) expression)

S-expression

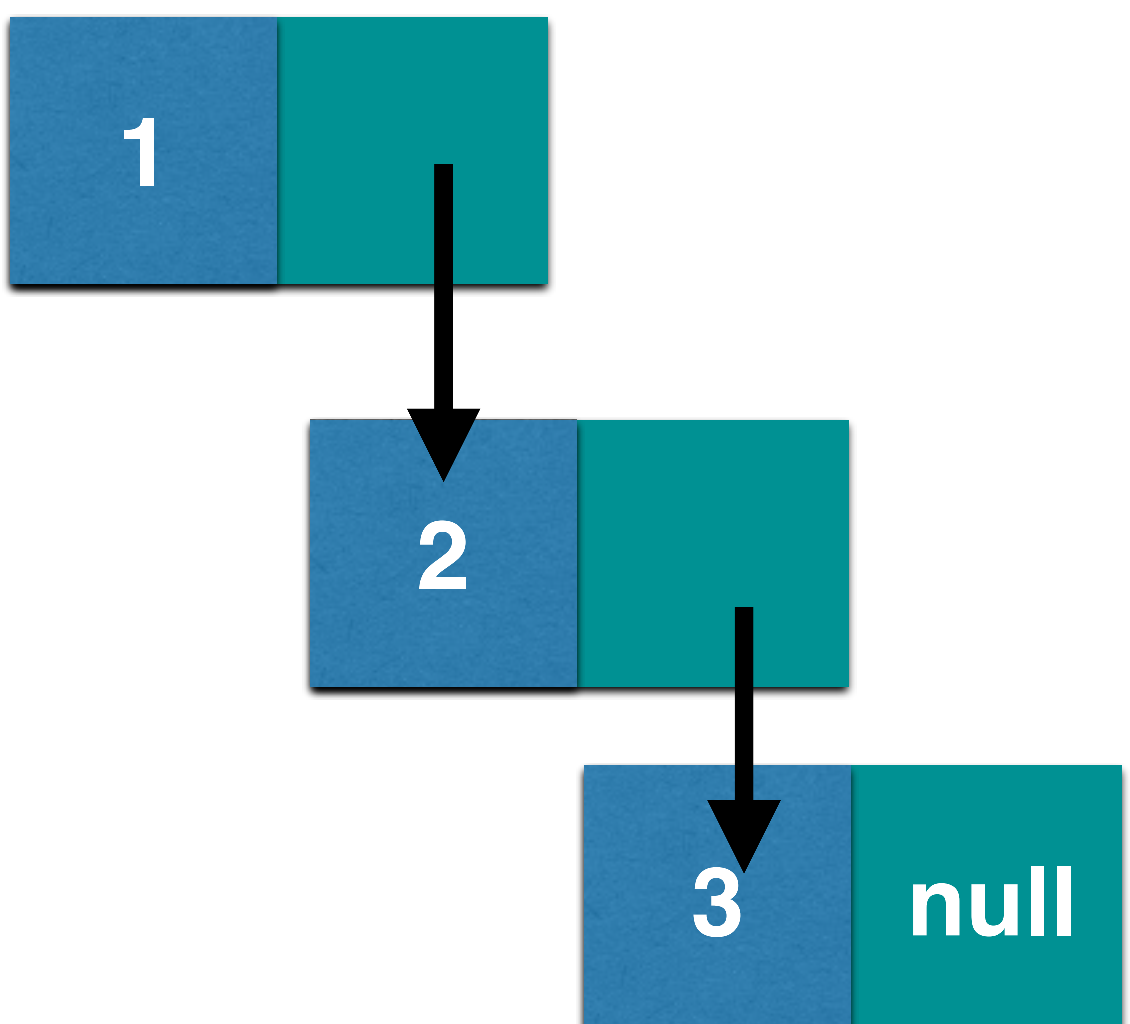
List of either **atoms** or **S-expressions**

(this (is an) $\boxed{\text{(s)}}$ expression)

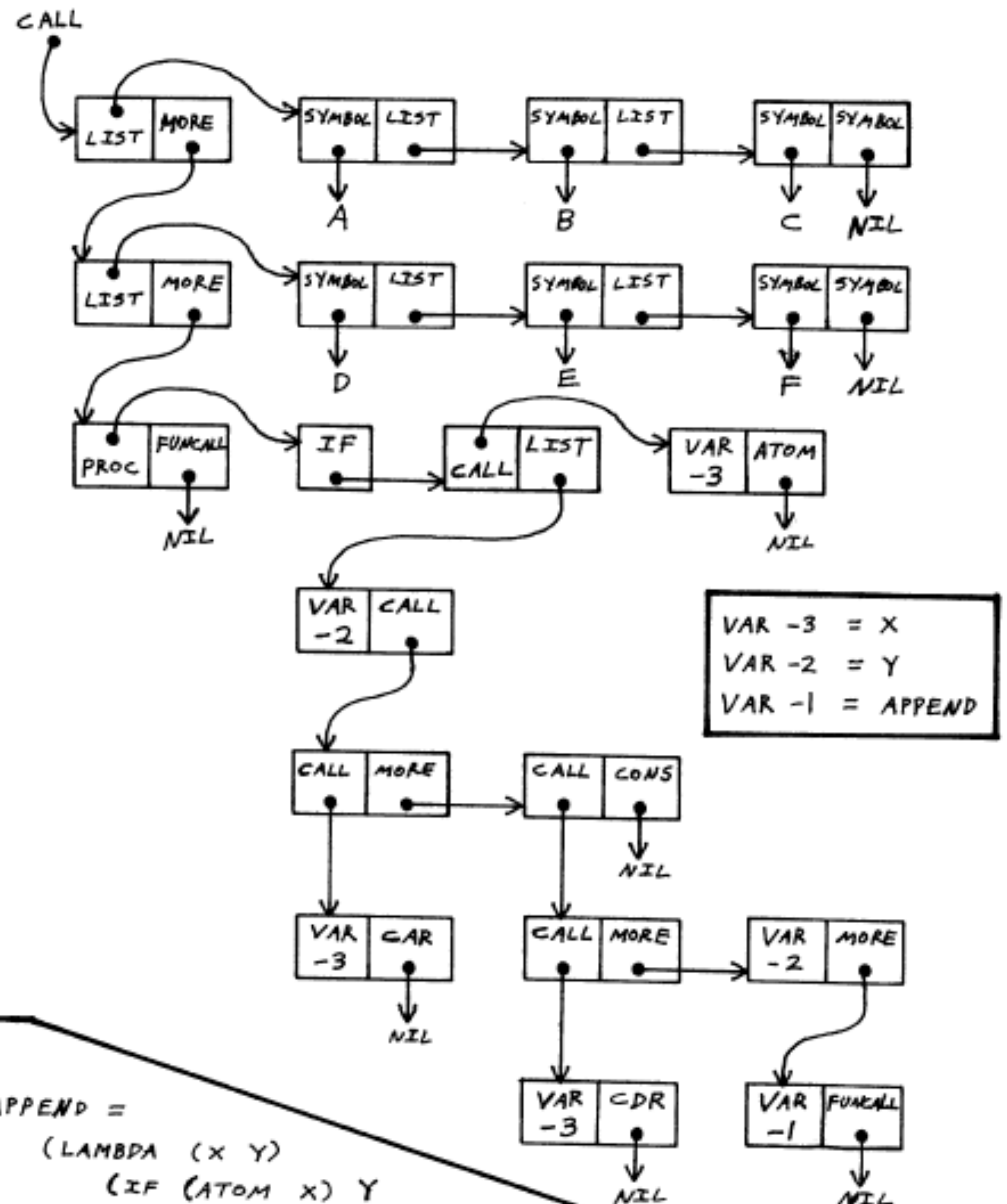
also an S-expression

LISP represents these as linked lists

(1 2 3)

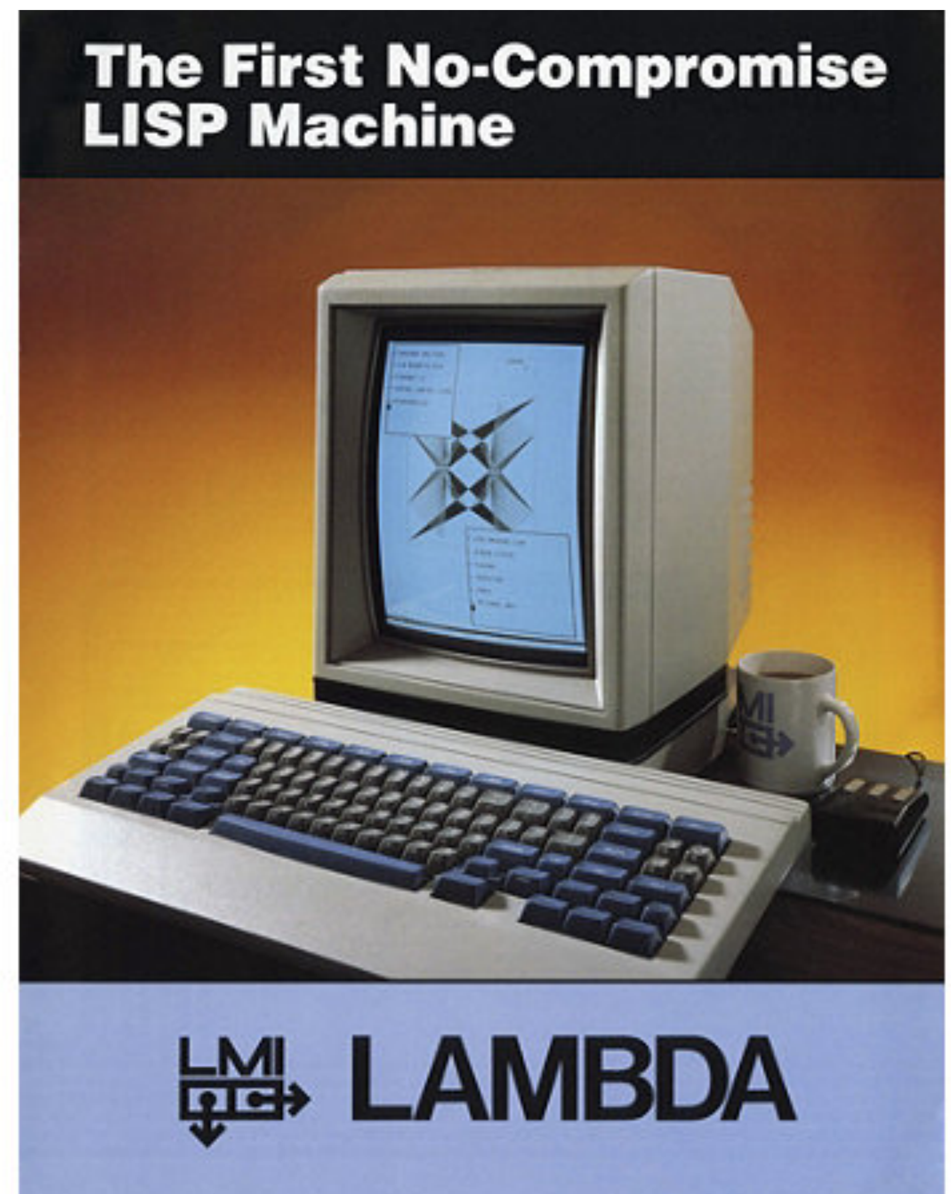


SIMPLE Expression for (APPEND '(A B C) '(D E F))




```
APPEND =
   (LAMBDA (X Y)
      (IF (ATOM X) Y
          (CONS (CAR X)
                (APPEND (CDR X) Y))))
```


LAMBDA


The First No-Compromise
LISP Machine
LMI
LAMBDA


QUOTE
OVER STRIKE
CLEAR INPUT
CLEAR SCREEN
HOLD OUTPUT
STOP OUTPUT

So how do we write programs in this?

A few terms

- LISP: The original language, grew very large over time
  - E.g., included an object system
- Scheme: Minimal version of LISP, partly used for teaching
- Racket: 90s reboot of Scheme, added many new features
  - Mostly compatible w/ Scheme

Tenants of Scheme

- Use recursion for computation, no mutable variables
- Basic abstraction is a list (made up of cons cells)
- Code is data

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

If you get stuck, use the debugger…!

Racket is *dynamically typed*

```
> (length 2)
length: contract violation
  expected: list?
  given: 2
> 
```

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

- Everything in parenthesis
- Prefix notation
- No variable assignment
- Recursion instead of loops
- No types
- No return

Here's what most confused me…

```
> (lambda x x)
#<procedure>
> (lambda (x) x)
#<procedure>
> (lambda (x) x) 1
#<procedure>
1
> ((lambda (x) x) 1)
1
> ((lambda x x) 1)
'(1)
> 
```

```
(define (bad_fib x)
    (cond
       [(eq? 0 x) 1]
       [(eq? 1 x) 1]
       [else 0])
)
```

Define `max`

- cond
- <
- >
- equal?

Define `max-of-list`

- empty?
- first
- rest
- length?

You can create functions with lambda

```
(lambda (x) (- x))

(lambda (str) (string-ref str 0))
```

```
(let ([x 1])
  (+ x 1))
```

Rewrite this in terms of lambda!

```
(let* ([x 1]
       [y (+ x 1)])
 (list y x))
```

Transform..

```
(let ([x 1])
  (+ x 1))
```

```
(lambda (x)
  (+ x 1)) 1
```

Transform..

```
(let ([x 1])
  (+ x 1))
```

```
((lambda (x)
   (+ x 1)) 1)
```

Let is λ

Lots of other things are λ too…

```
(define (f x) x)
```

shorthand for…

```
(define f (lambda (x) x))
```

```
(define (f x) x)

(define (f x y) x)

...

(define f (lambda (x) x))

(define f (lambda (x y) x))
```

```
(display “Hello”)
```

Define `acrostic`

Define hyphenate

```
> (hyphenate '("Kristopher" "Kyle" "Micinski"))
"Kristopher-Kyle-Micinski"
> |
```

Using higher order functions…

If you give me a function, I can use it

```
(define twice
  (lambda (f)
    (lambda (x)
      (f (f x)))))
```

Challenge: figure out how I would use `twice` to add 2 to 2

Use Racket's `add1` function

```
(add1 (add1 2))
```

Explain how twice works to someone next to you

When **listening**, push the person for clarification when you get confused

If you can’t figure it out, get help from someone around you

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

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

The following is **not** tail recursive

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

The following **is** tail recursive

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

This isn’t merely trivia!

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

factorial 2 1

factorial 1 2

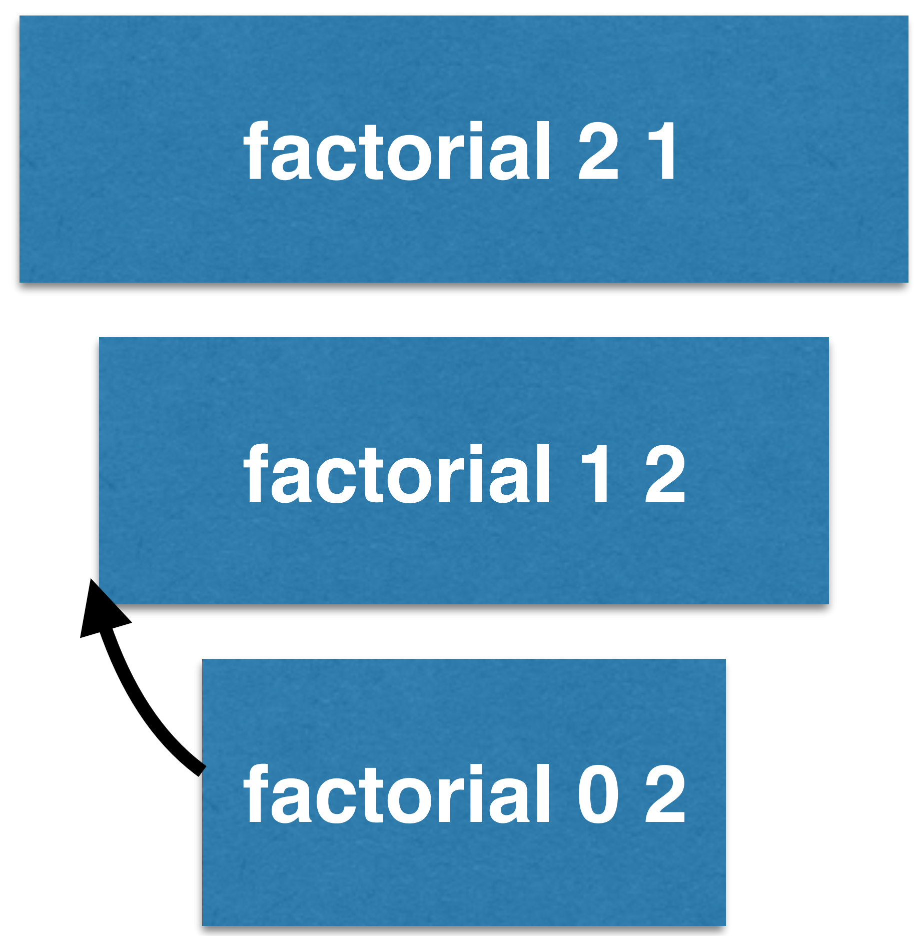
factorial 0 2

```
(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

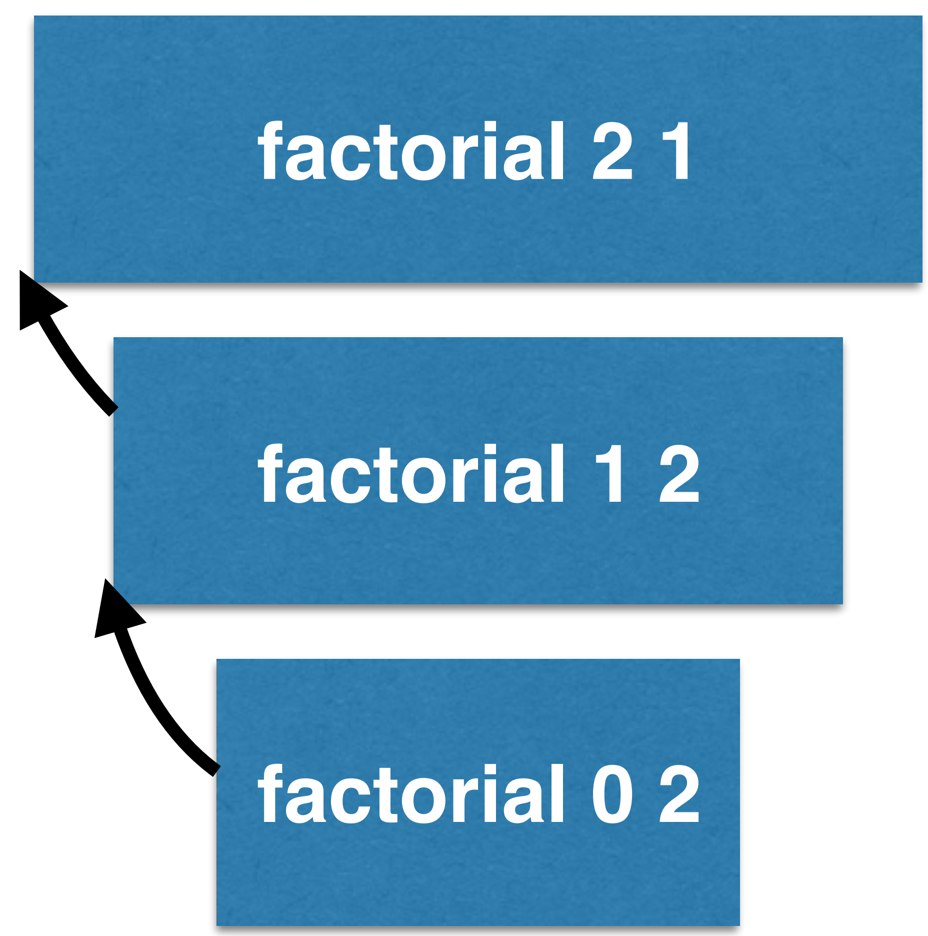


```
(define (factorial x acc)
    (if (equal? x 0)
        acc
        (* (factorial (- x 1) (* acc x)))))
; .. Later
(factorial 2 1)
```

>factorial 2 1

    >factorial 1 2

        >factorial 0 2

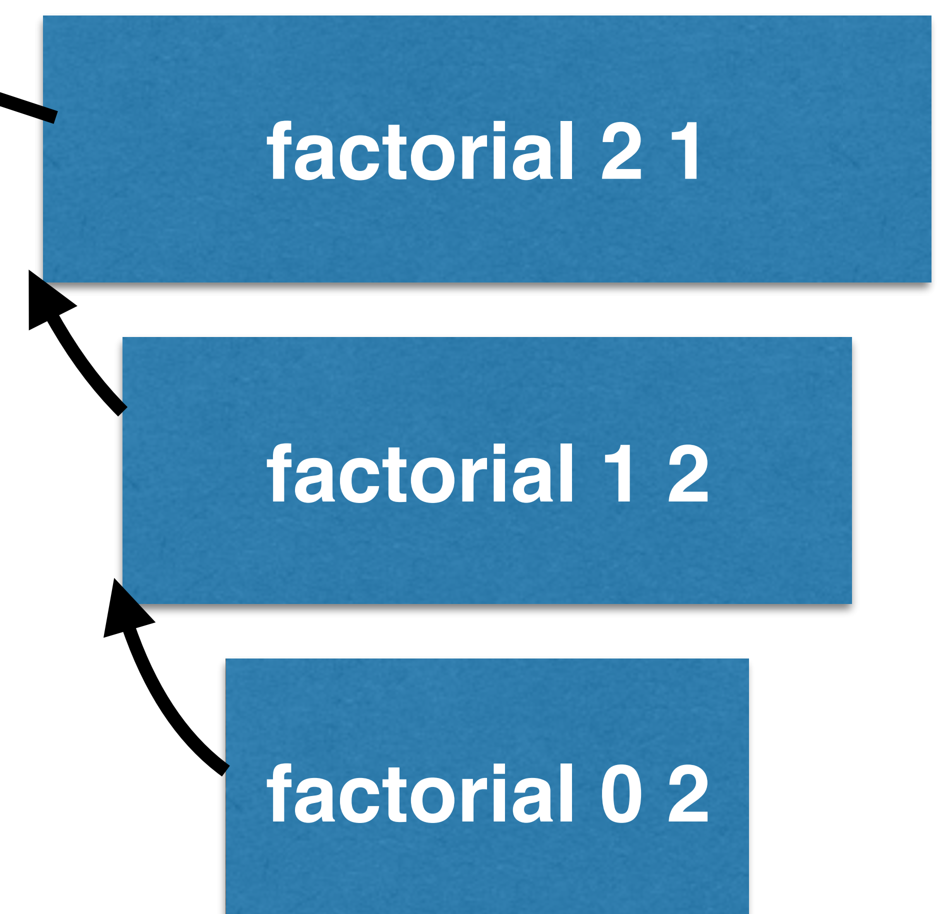


```
(define (factorial x acc)
    (if (equal? x 0)
        acc
        (* (factorial (- x 1) (* acc x)))))
; .. Later
(factorial 2 1)
```

>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)
```

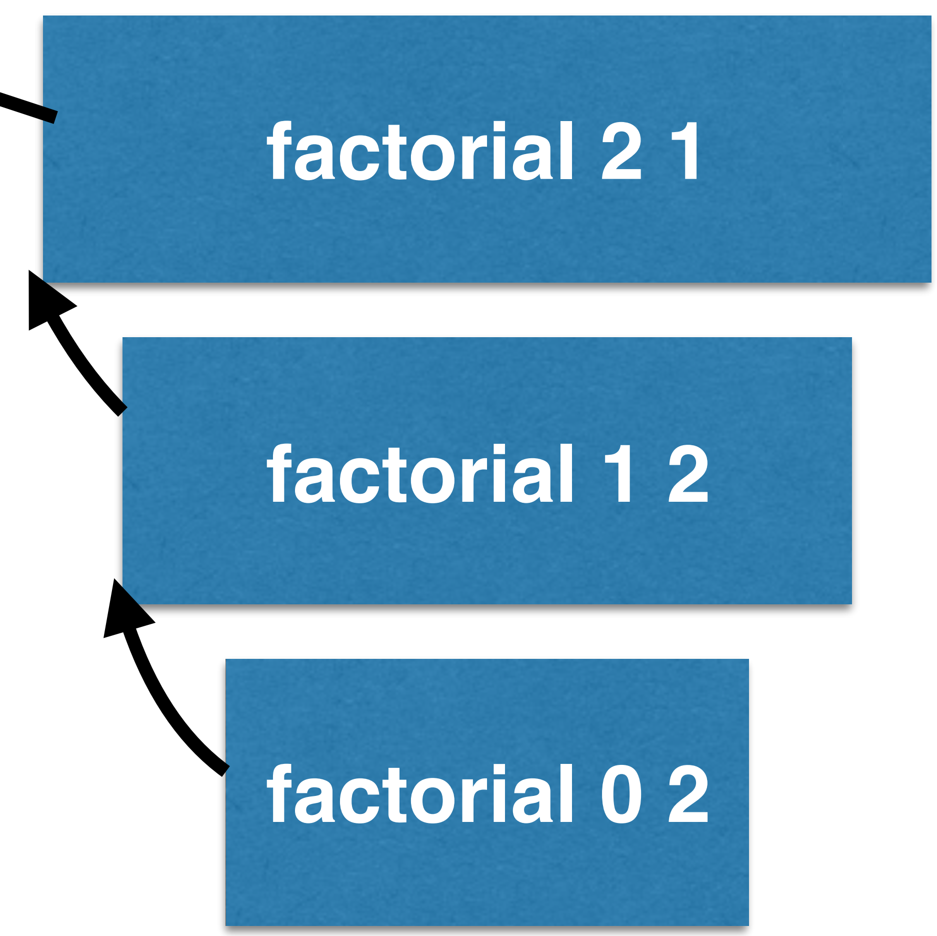
But wait!

I don't need the stack at all!

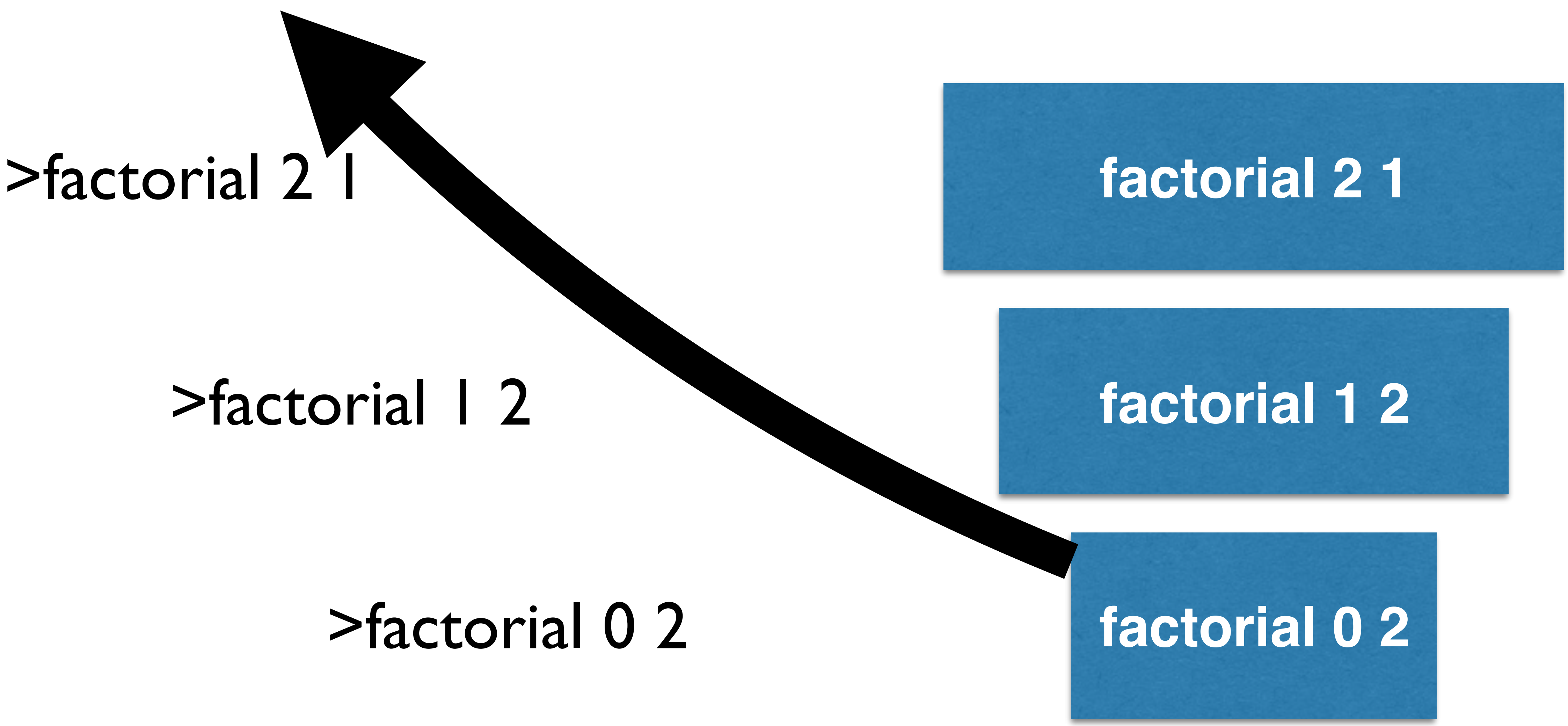
>factorial 2 1

>factorial 1 2

>factorial 0 2



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



Why **couldn't** I do that with this?

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

So basically, tail calls are GOTO

(In that they you don’t pay a stack penalty)