Course Website:

`https://kmicinski.com/cis352-s22`

CIS352 – Spring 2022

Kris Micinski (Asst. Prof)
Davis Silverman, Chang Liu (TAs)

- Main Course Objective
- Logistics, Instructors, Course Delivery
- Syllabus
- Projects
- Exams
- Course FAQs

Course Objective

The main goal of this course is to teach you to **write completely correct code** that you can clearly explain and easily understand

Course Objective

The main goal of this course is to teach you to **write completely correct code** that you can clearly explain and easily understand

We do this through **five coding projects**

And assess written skills through **exam questions**

Logistics

This is a **flipped-classroom**. You watch about ~80min video per week (1-2 per lecture). There is a **participation quiz** for each class

We **expect** you have watched the videos **before** lecture; lectures will involve problem-solving guided by examples

We expect you to be checking **Slack** often

`https://kmicinski.com/cis352-s22`

Instructors

Asst. Prof Kristopher Micinski is lead instructor

Davis Silverman (PhD) and Chang Liu (MS) are TAs

We are currently coordinating times for office hours; they will be posted on the syllabus in the first week

Syllabus

Most up-to-date syllabus always available at:

`https://kmicinski.com/cis352-s22/syllabus`

Grade-Calculator

We **encourage** you to use the **grade calculator**

Projects

This course has projects (with **deadlines**) that are assigned and graded via an **autograder**

`https://autograder.org`

You are expected to use the **Git interface** to the autograder; Autograder credentials will be sent out by the **first week**

We **try** to make projects sync up with the material presented at the corresponding time in the course

We **hope** you will reach out to us on Slack / office hours

Obviously: **start early**. The students who struggle most are those who put projects off them get cumulatively behind

Project Grading

- Each project is graded on a percent scale; your grade is the % of tests that pass (18/20 tests passing = 90%)
- Projects always due at 11:59PM Syracuse time
- Projects up to 72 hours after deadline—15% penalty (max 85%)
- Projects up to end of course—25% penalty
  - I.e., you can **always** get a 75%

Exams

- There are **four quizzes** and **one final**
- There are only **12 questions total**
- Question 1, 2, … will always be the same **topic**
  - Question 1 on each quiz will be about Racket forms/callsites
- You always get your **max** score of any question
- I.e., you can **raise your grade** (you will have 5 attempts at Q1)
- There are only as many questions as topics introduced so far
  - I.e., fewer attempts at Q12
- All questions are graded out of 10
  - Your final **exam grade** is average of all 12 questions

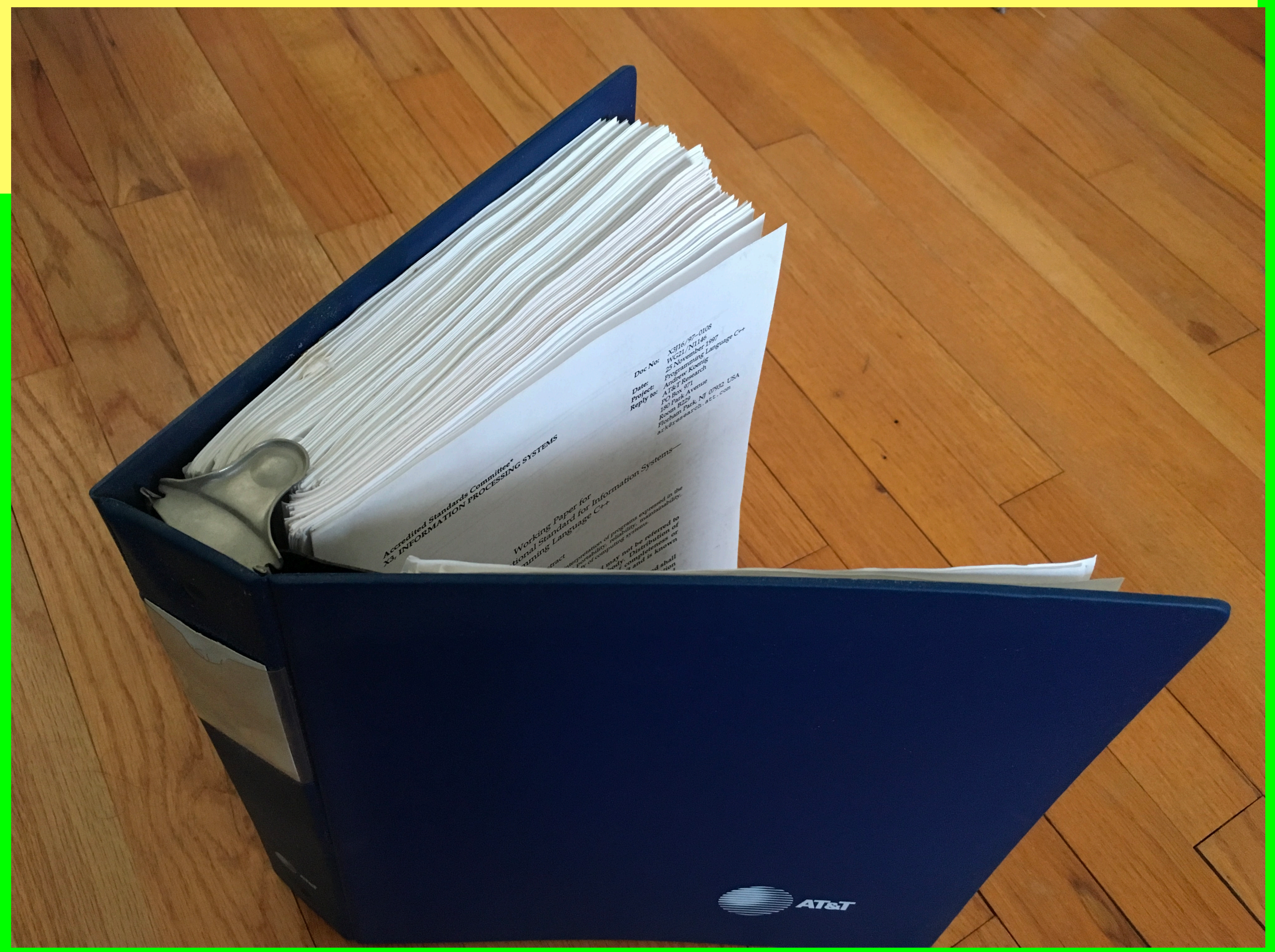
Participation

- There are lots of “participation points” available
  - Last semester, highest-participating student got 43
- <20 participation points = “minus” to your grade (A to A-)
- [20,30) participation points = no change to grade
- >= 30 participation points = “plus” to your grade (A- to A)
  - No A+ available—but I will track it for recommendations / refs

Course FAQs

Q: Why teach Racket and not C++ / Java / JS / Rust / …
A: We have chosen to teach a language that you can **fully-understand** so you can explain *precisely* how your code works

We **do** see value in C++/… and we will specifically comment on idioms in those languages when possible—but languages such as C++ are so complex (spec is 1000s of pages) we would need *multiple* classes to properly cover it



Course FAQs

Q: Why not start with type theory?
A: Our intro course, CIS252, covers Haskell—a strongly-typed language built on algebraic datatypes. Here, we focus on building **interpreters** that give *ground truth* behavior; we build up to types—as a means to rule out dynamic errors—at the end

Course FAQs

Q: Why emphasize functional programming / disallow `set!`

A: Functional programming is **simpler** (i.e., **more restrictive**), and thus easier to reason about. We will discuss how to implement state later on in the course, but we start by forcing students to program in a restricted purely-functional model because there are fewer opportunities for mistakes

Thanks! I’m excited for an enriching semester and look forward to helping you hack on projects

Racket Basics

CIS352 — Spring 2022

Kris Micinski

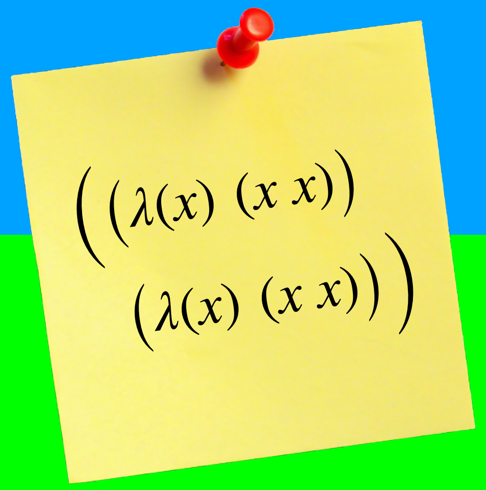
Racket

- **Dynamically-Typed:** variables are untyped, values typed
- **Functional:** Racket emphasizes functional style
  - Compositional—emphasizes black-box components
  - Immutability—requires automatic memory management
- **Imperative:** allows data to be modified, in carefully-considered cases, but doesn't emphasize "impure" code

Racket

- **Object-oriented**: racket has a powerful object system
- **Language-oriented**: Racket is really a language toolkit
- **Homoiconic:** the same structure used to represent **data** (lists) is also used to represent **code**

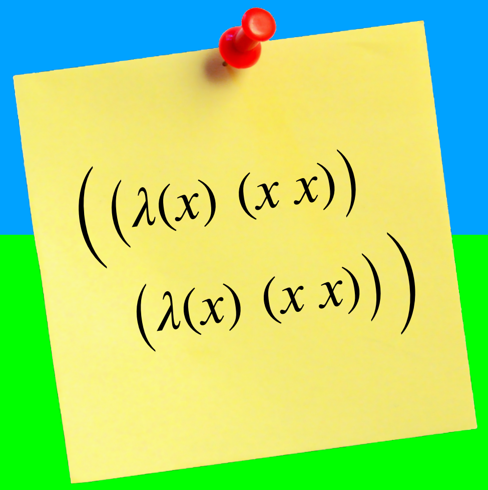
Example



Calculating the slope of a line in Racket

```
(define (calculuate-slope x0 y0 x1 y1)
  (/ (- y1 y0) (- x1 x0)))
```

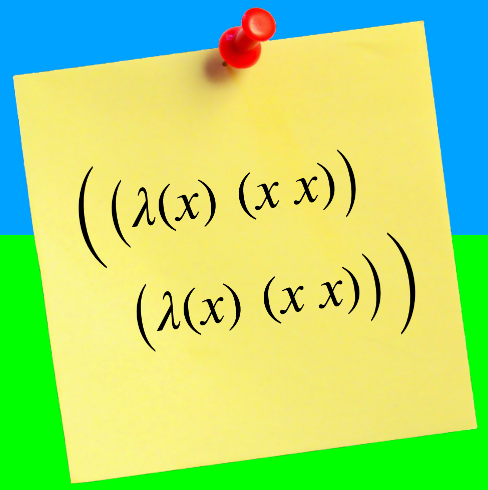
Example




```
(define (calculuate-slope x0 y0 x1 y1)
  (/ (- y1 y0) (- x1 x0)))
```

Prefix notation

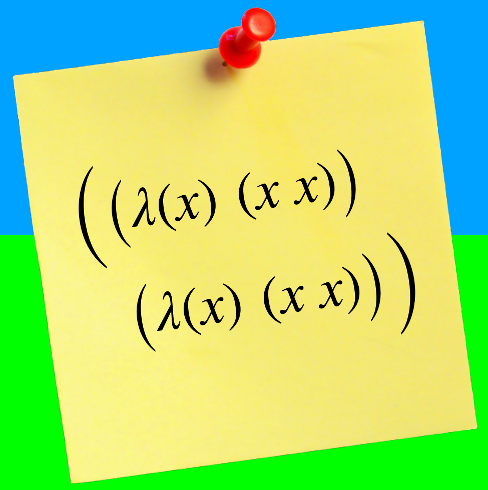
Example



Functions defined via prefix notation, too

```
(define (calculuate-slope x0 y0 x1 y1)
  (/ (- y1 y0) (- x1 x0)))
```

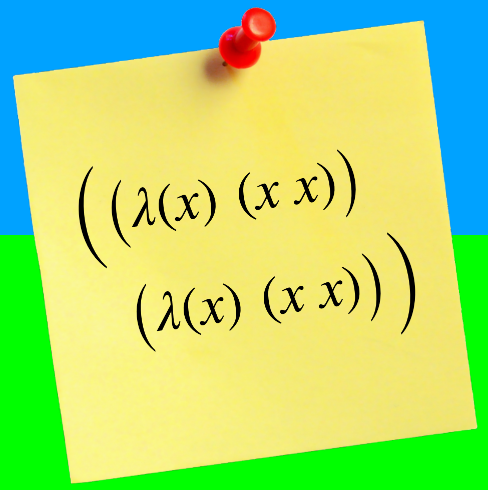
Example



Calls to user-defined functions also in prefix notation

```
(define (calculuate-slope x0 y0 x1 y1)
  (/ (- y1 y0) (- x1 x0)))
// C - calculate-slope(0,0,3,2);
(calculate-slope 0 0 3 2)
```

Example



Note: preferred style puts closing parens at end of blocks

```
(define (calculuate-slope x0 y0 x1 y1)
  (/ (- y1 y0) (- x1 x0)))

(calculate-slope 0 0 3 2)
```

Basic Types

- **Numeric tower**. Numeric types gracefully degrade
  - E.g., `(* (/ 8 3) 2+1i)` is `16/3+8/3i`
  - Note that `2+1i` is a **literal** value, as is `2.3`
- **Strings** and **characters** (“`foo`” and `#\a`)
- **Booleans** (`#t` and `#f`) including logical operator (e.g., `or`)
  - Note that operators “short circuit”

Basic Types contd.

- **Symbols** are interned strings `'foo`
  - Implicitly only one copy of each, unlike (say) strings
  - Impact on space / memory usage
- The `#<void>` value (produced by `(void)`)

Exercise

Compute the sum of the following:

- 2/3 and 1.5
- 3+8i and 3i
- 0 and positive infinity (`+inf.0`)

Exercise

Compute the sum of the following:

- `(+ 2/3 1.5)`

  `2.1666666666666665` (N.B., result is **inexact**)

- `(+ 3+8i 0+3i)`

  `3+11i`

- `(+ 0 +inf.0)`

  `+inf.0`

Racket Forms and Callsites

CIS352 — Spring 2022

Kris Micinski

Forms

- A **form** is a recognized syntax in the language
  - `(if …)`, `(and …)` are forms
  - But +, `list` refer to functions
  - Core forms defined by the language (if/and/define/…)
  - You can define new forms too! More on this later…
- Scheme prefers to give a small number of general forms.

Forms

- The tag just after the open-paren determines the form:
  - (define foo value) — Define a variable
  - (define (foo a0 a1 …) body) — Define a function
  - (if guard e-true e-false), (or e0 e1 …), etc
- By default, otherwise, (e0 e1 …) is a **function call**

Value and Expressions

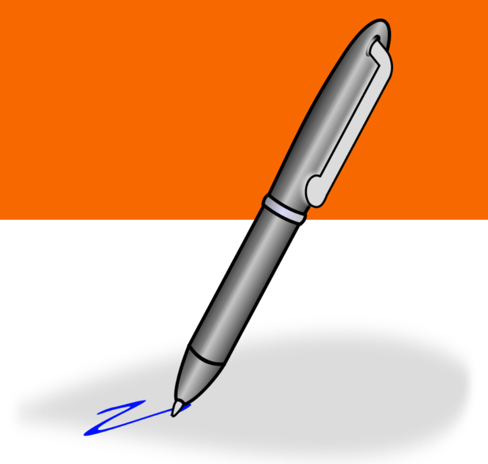
- Every language has a set of **values**
  - Primitive objects representable at runtime
  - Expressions evaluate to values
  - Numbers, strings, but also functions (closures)
- An **expression** is any syntax that evaluates to a value
  - Very important term to know!

Exercise

Which of the following are expressions:

- (define x 23)
- x
- (+ x 3)
- (define (foo x) (+ x 1))
- (if x (foo x) (bar x))

Exercise



Which of the following are expressions:

- (define x 23) — Doesn’t evaluate to a value
- **x**
- **(+ x 3)**
- (define (foo x) (+ x 1)) — Doesn’t eval to value
- **(if x (foo x) (bar x))**

Exercise

Define a function that takes an argument, x, and returns:

- x times 2, if x is greater than 0
- x times -2, otherwise

Exercise

```
(define (f x)
  (if (< x 0)
      (* 2 x)
      (* -2 x)))
```

Exercise

Define a function that takes an argument, x, and returns:

- x divided by 2, if x is even
- x times 3 plus 1, if x is odd

Hint: use = and `modulo` to check if x is even/odd

Exercise

```
(define (collatz x)
  (if (= 0 (modulo x 2))
      (/ x 2)
      (+ 1 (* 3 x))))
```

Definitions and the Environment

CIS352 — Spring 2021

Kris Micinski

Definitions

- The form **define** is used to define variables
- Define comes in two forms
  - `(define id expr)` — Define variable `id` as expr
  - `(define (f a0 …) body …+)`
    - Define a function f with arguments a0, …
    - At least one body (typically only one)

Exercise

- Define a variable named x to be 42
- Define a function foo, which behaves as the identity function

The Environment

- The **environment** at some point in the program includes the set of variables in scope (accessible) at that point
- Every syntactic point has a (potentially) unique environment

```
(define x 23)
(+ x 1) ;; x is 23
(define y 24)
(+ x y) ;; x & y defined
```

Environments Nest

- Note that environments are hierarchical
- Definitions inside a function do not escape the function
- This relates to **lexical scope** which we will define soon

```
(define y 5)
(define (foo)
   (displayln y) ;; 5
   (define y 4)
   y)            ;; 4
(foo)            ;; 4
y                ;; 5
```

Exercise

What does the following function return:

```
(define (foo)
  (define + 1)
  (define / (* 2 +))
  (- + /))
```

Exercise

What does the following function return:

-1

Upshot: “built-in” functions are not special

```
(define (foo)
  (define + 1)
  (define / (* 2 +))
  (- + /))
```

Let

- Definitions with define are not expressions
- `(let ([var e]) e-body)`
  - Expression: evaluates `e-body` with `var` defined as `e`
  - Can have more than one var

```
(let ([x 2])
  (+ x 3))    ;; 5

(let ([x 2]
      [y 3])
  (+ x y))    ;; 5
```

Let

- Let does not allow simultaneous bindings to see each other
- I think of it as “parallel let”

```
(let ([x 2]
      [y x])  ;; bad
  (+ x y))    ;; 5
```

Let*

- Let* lets you define a sequence of variables
- I think of it as "sequential let"

```
(let* ([x 2]
       [y x])  ;; good
  (+ x y))     ;; 5
```

Textual Reduction

CIS352 — Spring 2021

Kris Micinski

This lecture takes place on the whiteboard.

Case Splitting and Lists Intro

CIS352 — Spring 2021

Kris Micinski

Cond

- Cond allows multiple guards to be checked
- ```
  (cond [guard0 body0]
        [guard1 body1]
         …
        [else bodyelse]) ;; optional
  ```
- Checks each guard sequentially, evaluates first body

```
(define (foo x)
  (cond [(= x 42) 1]
        [(> x 0)  2]
        [else     3]))
```
```

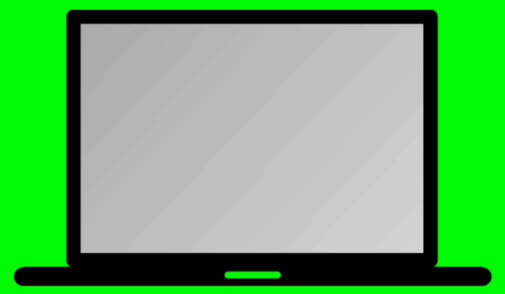
Exercise

The absolute value of a number x is:

- x is x is greater than 0
- 0 if x = 0
- -x if x is less than 0

Translate this definition into a function using cond

Exercise



The absolute value of a number x is:

- x is x is greater than 0
- 0 if x = 0
- -x if x is less than 0

Translate this definition into a function using cond

```
(define (abs x)
  (cond [(> x 0) x]
        [(= x 0) 0]
        [(< x 0) (- x)]))
```

Exercise

Say we have the following:

```
(cond [g_0 b_0]
      [g_1 b_1]
       …
      [else b_else])
```

How can we rewrite the above to use only if?

Exercise

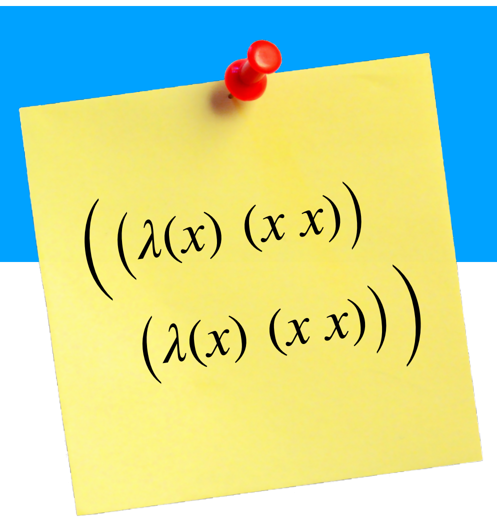
Say we have the following:

```
(cond [g₀ b₀]
      [g₁ b₁]
       …
      [else belse])
```

How can we rewrite the above to use only if?

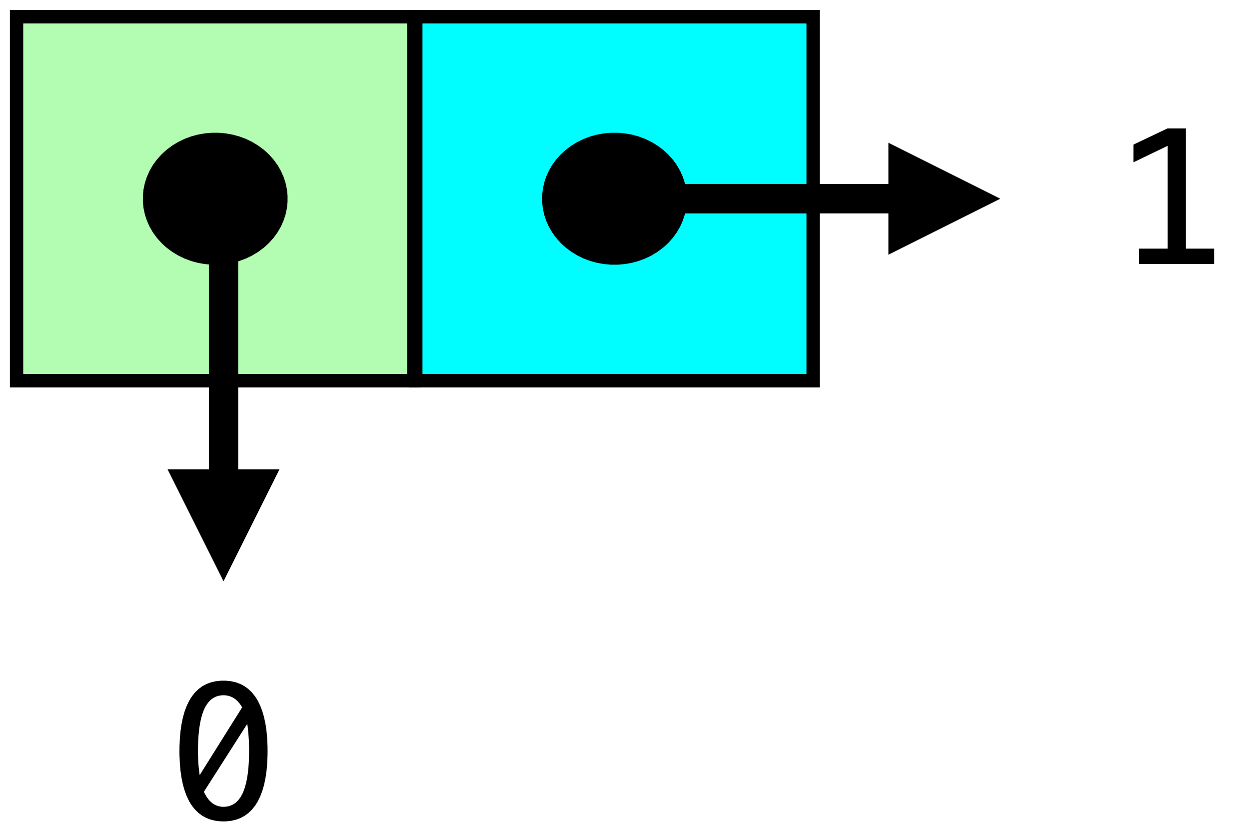
```
(if g₀ b₀
 (if g₁ b₁
   …
   (if gn-1 bn-1 belse) …))
```

Example

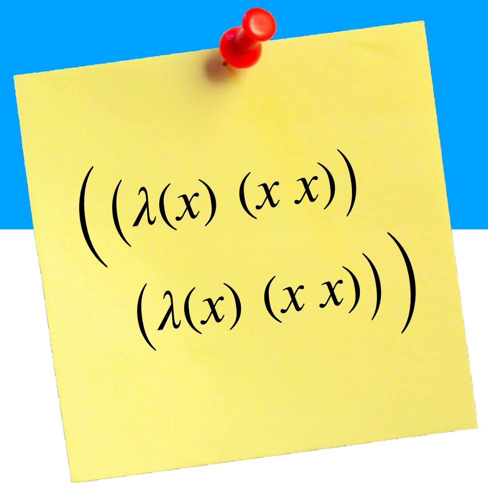


The function **cons** builds a cons cell / pair

```
(cons 0 1)
```

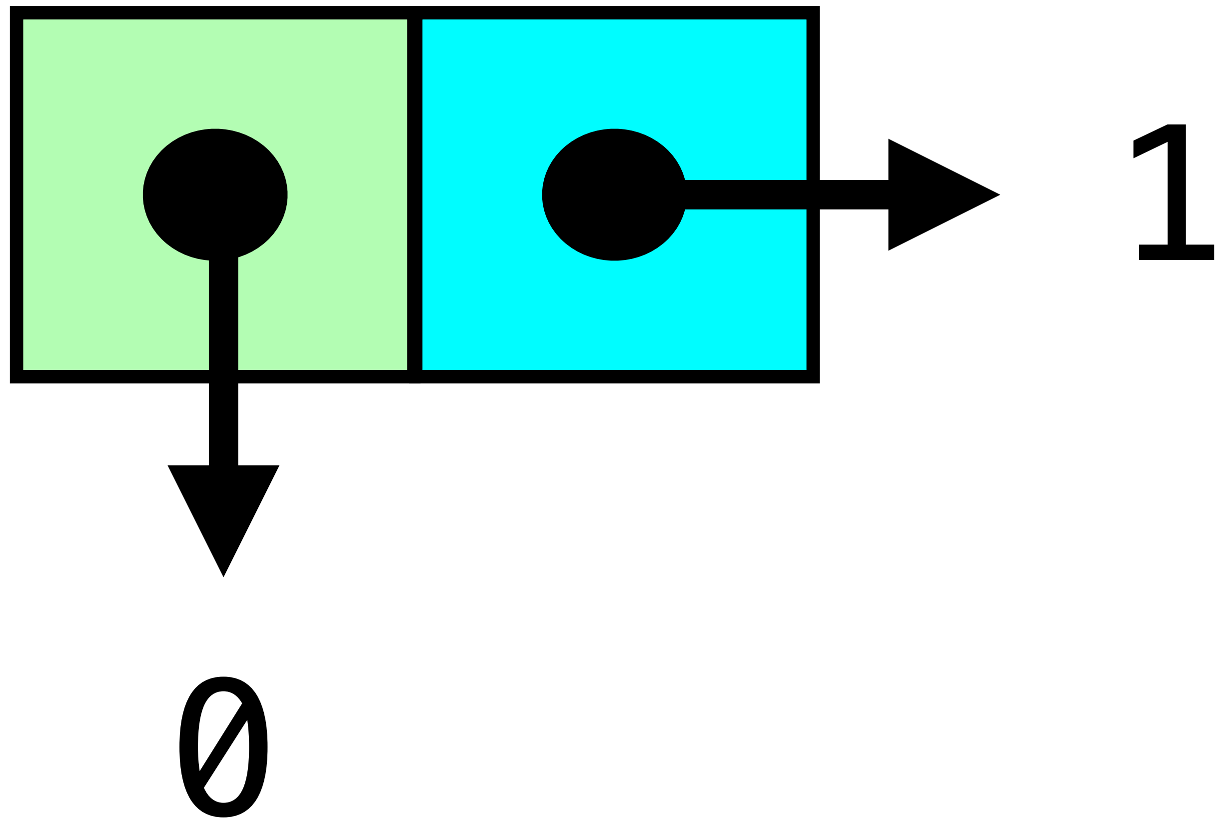


Example

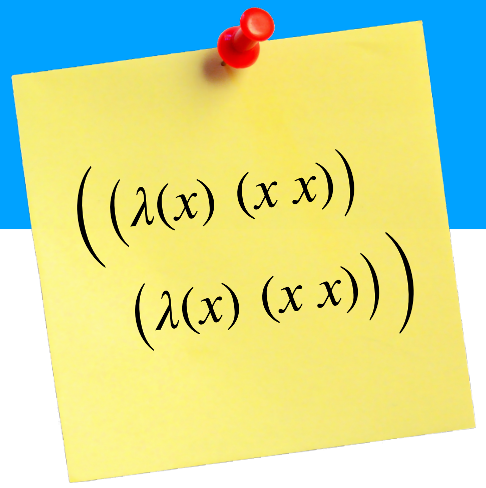


The function **car** gets the left element

`(car (cons 0 1))` is `0`

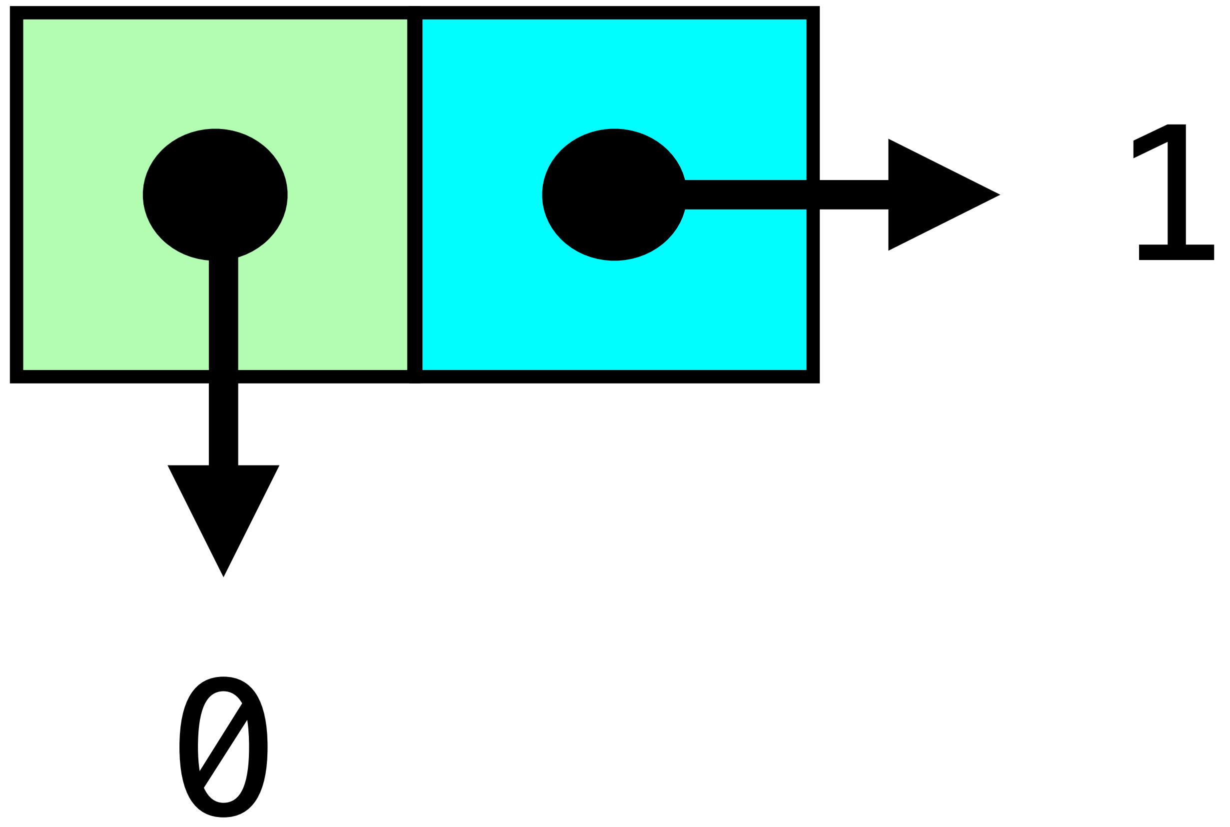


Example

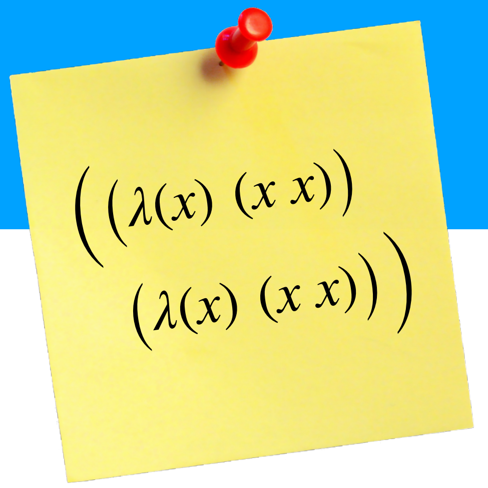


The function **cdr** gets the right element

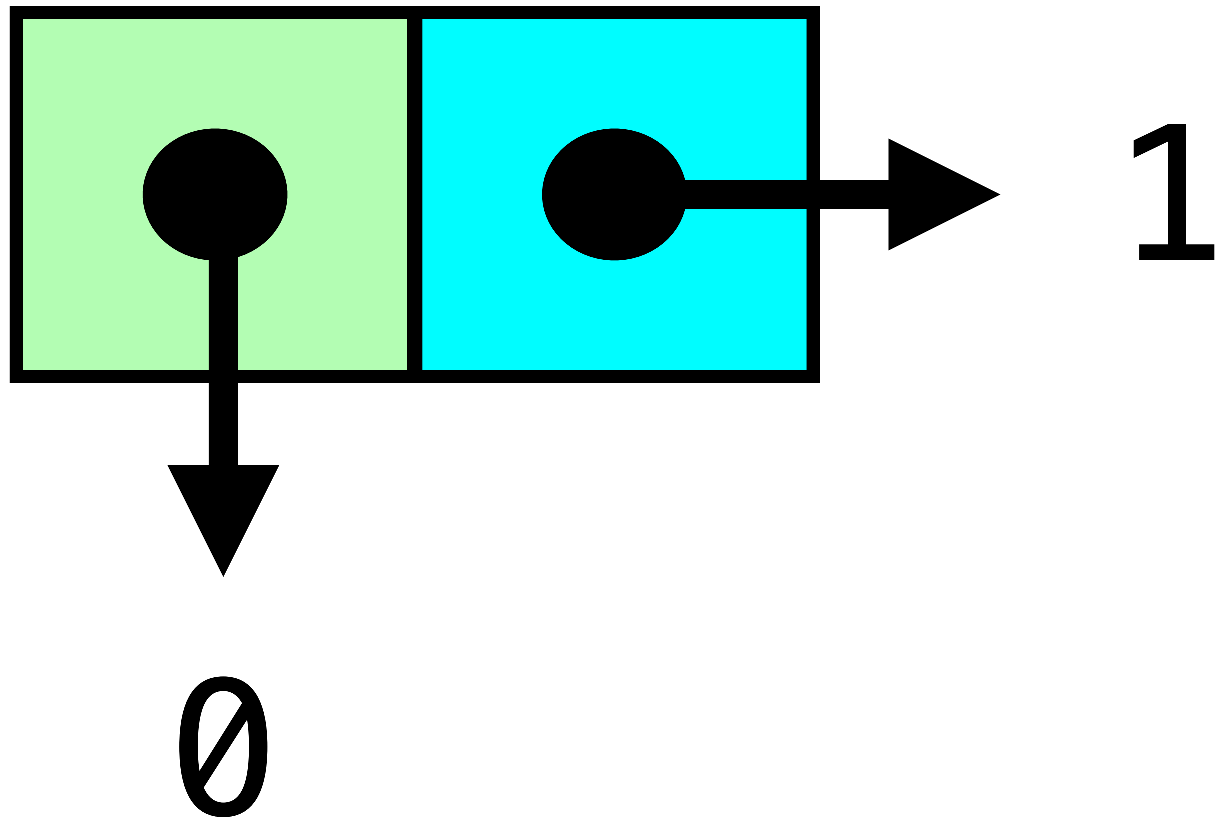
`(cdr (cons 0 1))` is `1`



Example



`(cdr (cons 0 1))` is `1`



The names **car** and **cdr** come from the original implementation of LISP on the IBM 704

Lists

- Racket has **lists**—sequences of cons cells ending w/ `‘()`
- The **empty list** (or “null") is special, `‘()`
- Many ways to build them
  - `(list 1 2 3) ;; Variadic function`
  - `‘(1 2 3)      ;; Datum representation`
- There are **three** operations on lists
  - `empty?` / `null?`
  - `first` / `car`
  - `rest` / `cdr`

Lists continued…

- Using empty?, car, and cdr, we can write many utilities
  - All definable ourselves, also in Racket by default
  - (length l) — Length of l
  - (list-ref l i) — Get ith element of list (0-indexed)
  - (append l0 l1) — Append l1 to the end of l0
  - (reverse l) — Reverse the list
  - (member l x) — Check if x is in l

Exercise

Using cond, write a function that takes a list l and an index x and returns…

- The first element if x = 0
- The second element if x = 1
- The third element if x = 2
- Otherwise return 'unknown

Case Splitting and Lists Intro

CIS352 — Spring 2021

Kris Micinski

Lambdas

CIS352 — Spring 2021

Kris Micinski

First-Class Functions

- In Racket, functions are **first-class** values
- Can be bound to vars, returned from fns, etc..
- Languages w/ functions as values are **functional**

Lambdas (in Racket)

- `(lambda (x0 x1 …) body)`
  - Anonymous function: bind x0, … in body
  - Can appear at any callsite (just like an identifier)

```
(define f (lambda (x) x))
(define (double g)
  (lambda (x) (g (g x))))
```

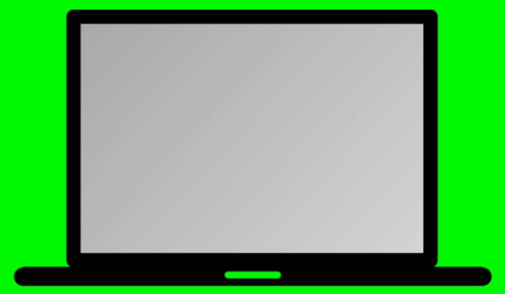
Exercise

```
(define f (lambda (x) x))
(define (double g)
  (lambda (x) (g (g x))))
```

Evaluate the following expressions:

- `(f 1)`
- `((double f) 42)`
- `((double (lambda (x) (* x 2))) 2)`

Exercise



Write a function, (foo f), that:

- Accepts a function f, maps ints to ints
- ((foo f) x) = (f |x|), |x| is abs. value of x

Textual Reduction of Lambdas

- Previously, we assumed **environment** of definitions
- Instead, can think of **lambdas** as primitive
- Environment maps identifiers to lambdas

```
(define (f x) x)
;; equiv
(define f (lambda (x) x))
```

Textual Reduction of Lambdas

- After reducing all args to values, substitute (into the body) the actual arguments in place of the formal arguments.

```
((lambda (x y) x) (+ 1 1) 3)
=> ((lambda (x y) x) 2 3)
=> 2
```

Exercise

Use textual reduction to reduce the following:

```
((((lambda (x) x) (lambda (x) x))
  ((lambda (x) x) (lambda (x) x)))
 (+ 1 2))
```

Hint: remember, in **applicative order** we always evaluate the **leftmost, innermost** application. In other words, we process (e0 e1 …) by reducing e0 … to values in order, then applying.

Exercise

Use textual reduction to reduce the following:

```
((((lambda (x) x) (lambda (x) x))
  ((lambda (x) x) (lambda (x) x)))
 (+ 1 2))
```

If this sounds complicated, you would be right to just think about it as “left to right”

Languages w/o First-Class Functions

- In modern times, somewhat hard to imagine
- C is a good example: procedural but **not** functional
- C callsites: quasi-functional behavior via fn pointers
  - But not really: C doesn't have **closures**

```
// The C library QuickSort function
void qsort(void *base, // array to sort
           int items,  // really size_t
           int elem_size,
           // pointer to compare fn
           int (*compare)(void*, void*))
```

Cons Diagrams and Boxes

CIS352 — Spring 2021

Kris Micinski

Derived Types

- **S-expressions** (**symbolic** expression)
    - Untyped lists that generalize neatly to trees:

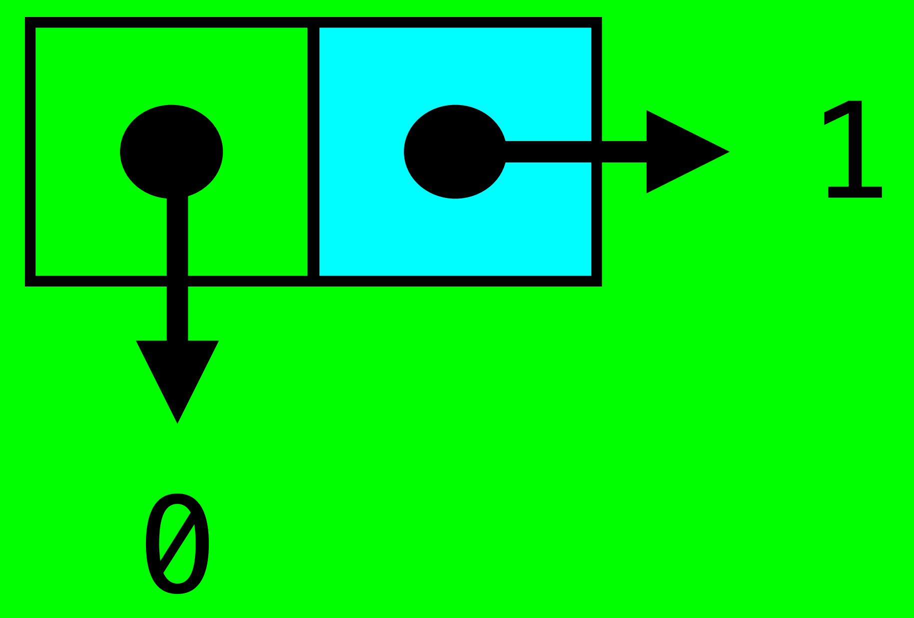
```
(this (is an) s expression)
```

- Computer represents these as **linked** structures
    - Cons cells of head & tail `(cons 1 2)`

Derived Types

- Racket also has **structural** types
  - Defined via **struct**; aids robustness
  - We will usually prefer agility of “tagged” S-expressions
- Also an elaborate object-orientation system (we won’t cover)

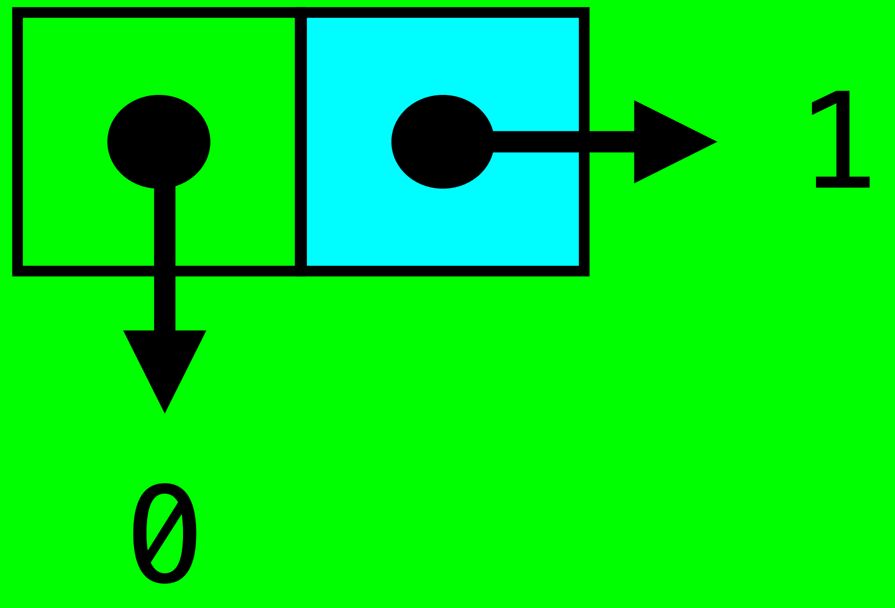
`(cons 0 1)`



The function **cons** builds a cons cell

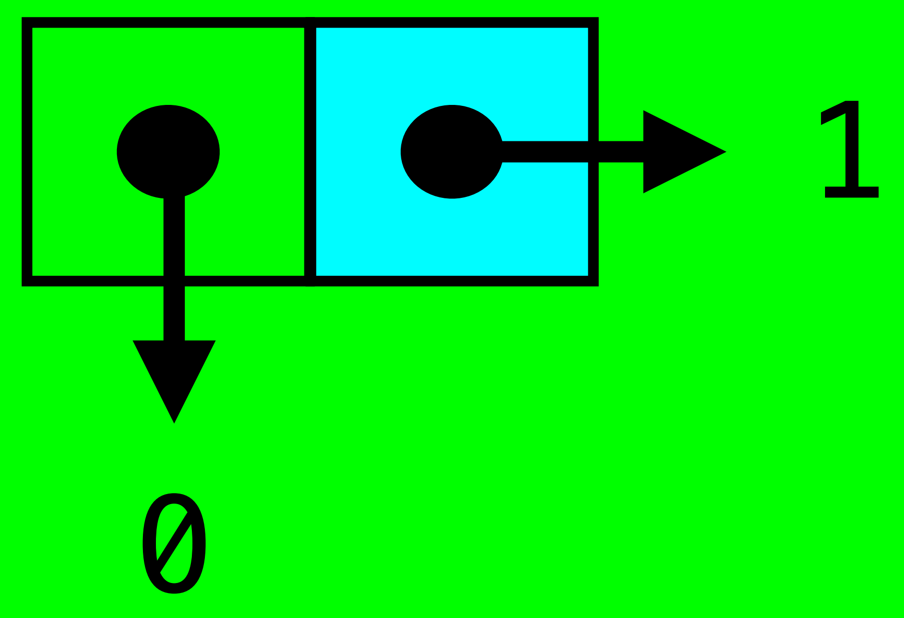
The function **car** gets the left element

`(car (cons 0 1))` is `0`



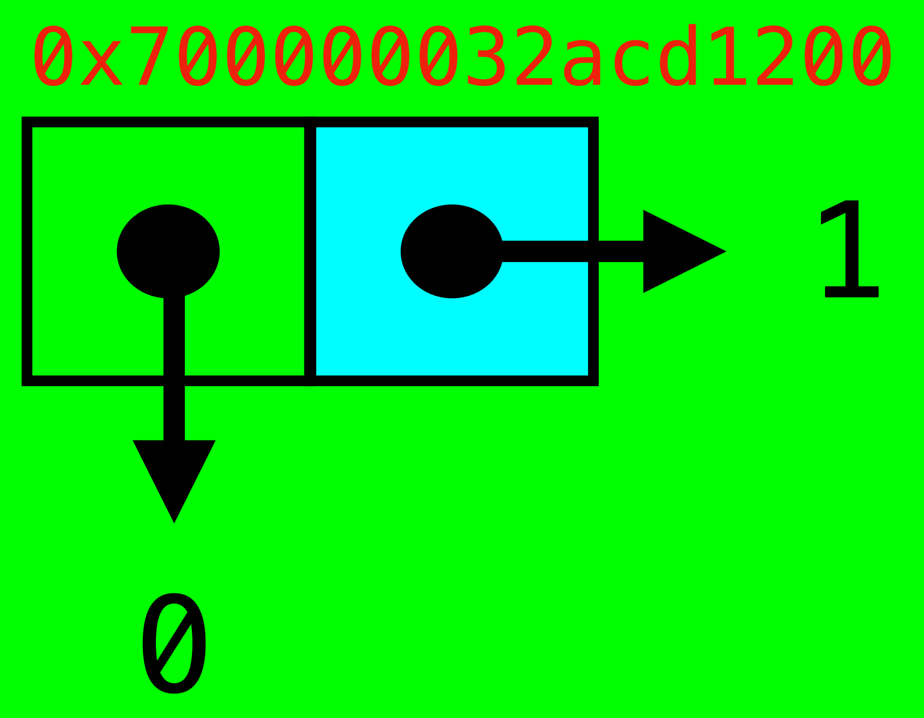
The function **cdr** gets the left element

`(cdr (cons 0 1))` is `1`



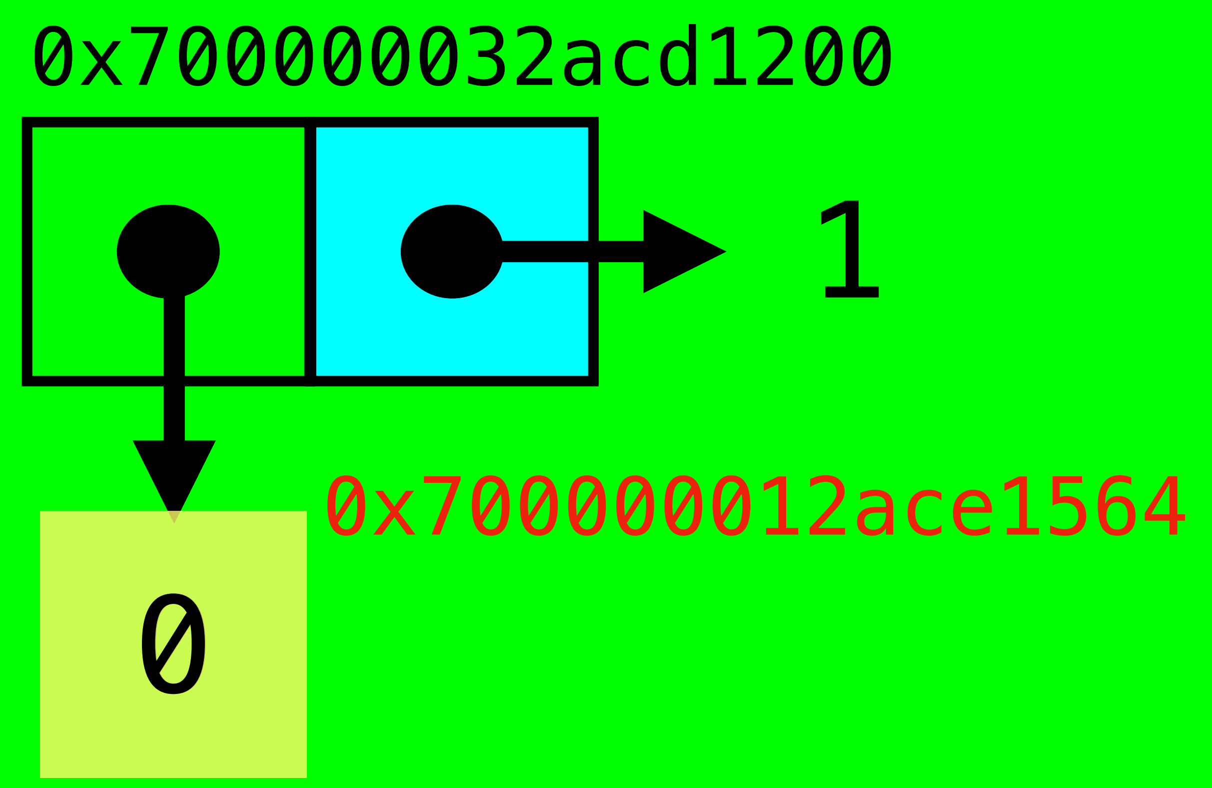
At runtime, each cons cell sits at an **address** in memory

`(cdr (cons 0 1))` is `1`



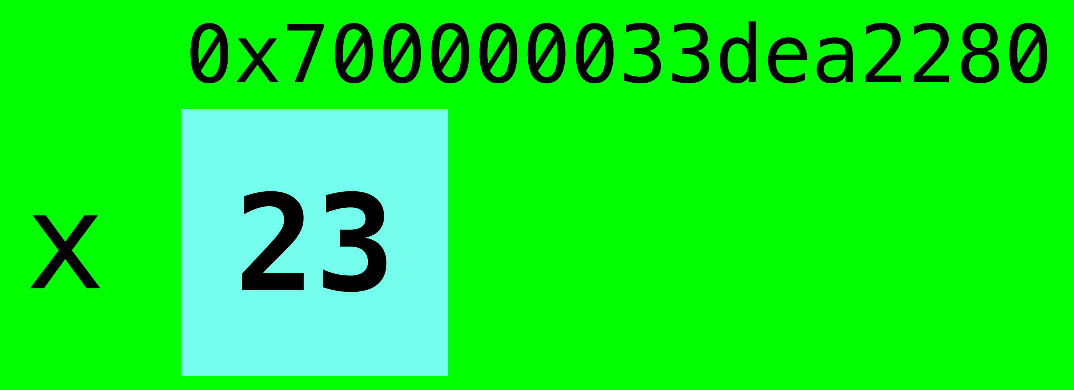
In fact, numbers are **also** stored in memory locations.

They are thus said to be a “boxed” type



Actually, every Racket variable stores a value in some "box" (i.e., memory location)

```
(define x 23)
(displayln x)
(set! x 24)
(displayln x)
```



Actually, every Racket variable stores a value in some "box" (i.e., memory location)

```
(define x 23)
(displayln x)
(set! x 24)
(displayln x)
```

0x700000033dea2280

x **23**

```
Console output…
> 23
```

Actually, every Racket variable stores a value in some "box" (i.e., memory location)

```
(define x 23)
(displayln x)
(set! x 24)
(displayln x)
```

`0x700000033dea2280`

x **24**

x's value **changes** to 24

```
(define x (vector 1 2 3))
(vector-set! x 1 0)
x
;; ‘#(1 0 3)
```

Vectors (similar to arrays) are mutable, and give O(1) indexing and updating

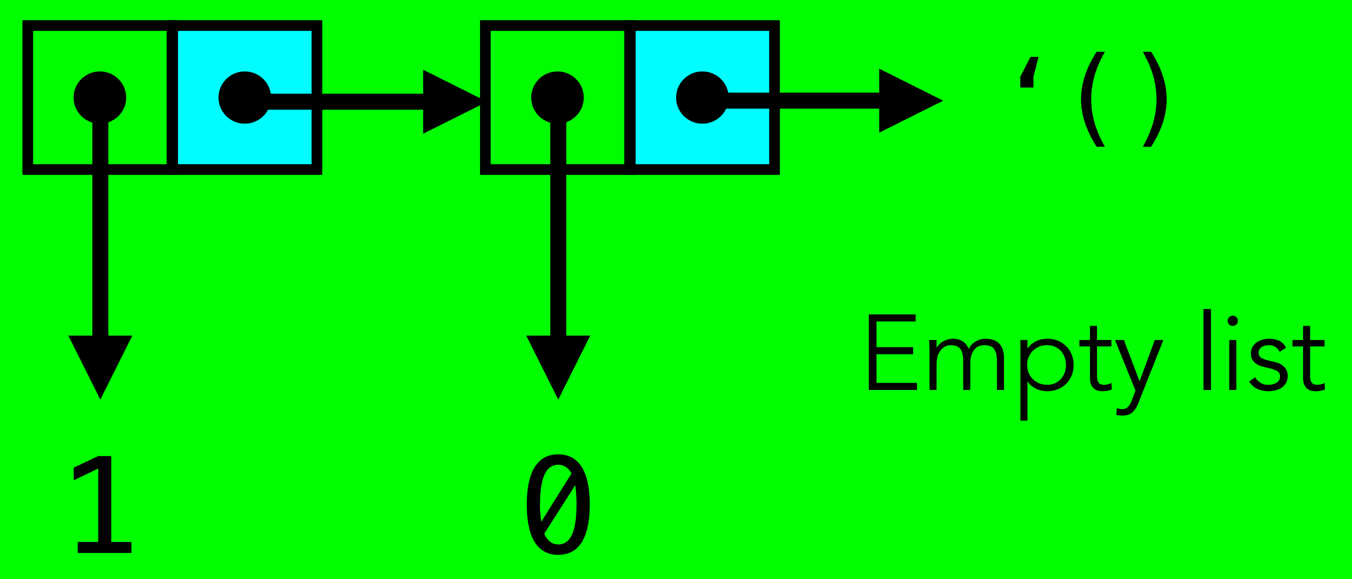
Unless we say otherwise, you should avoid using set!, any use will be at your own risk

Similarly, avoid vector-set!, hash-set!, …

Using `set!` will, in CIS352, lead to hard-to-debug code that will make it much harder for instructors to understand your code

Pairs enable us to build **linked lists** of data

```
(cons 1 (cons 0 '()))
```



This is how Racket represents lists in memory

Note that in Racket, the following are equivalent

```
(cons 2 (cons 1 (cons 0 ‘())))
‘(2 1 0)
```

But the following is called an **improper list**

```
(cons 2 (cons 1 0))
‘(2 1 . 0)
```

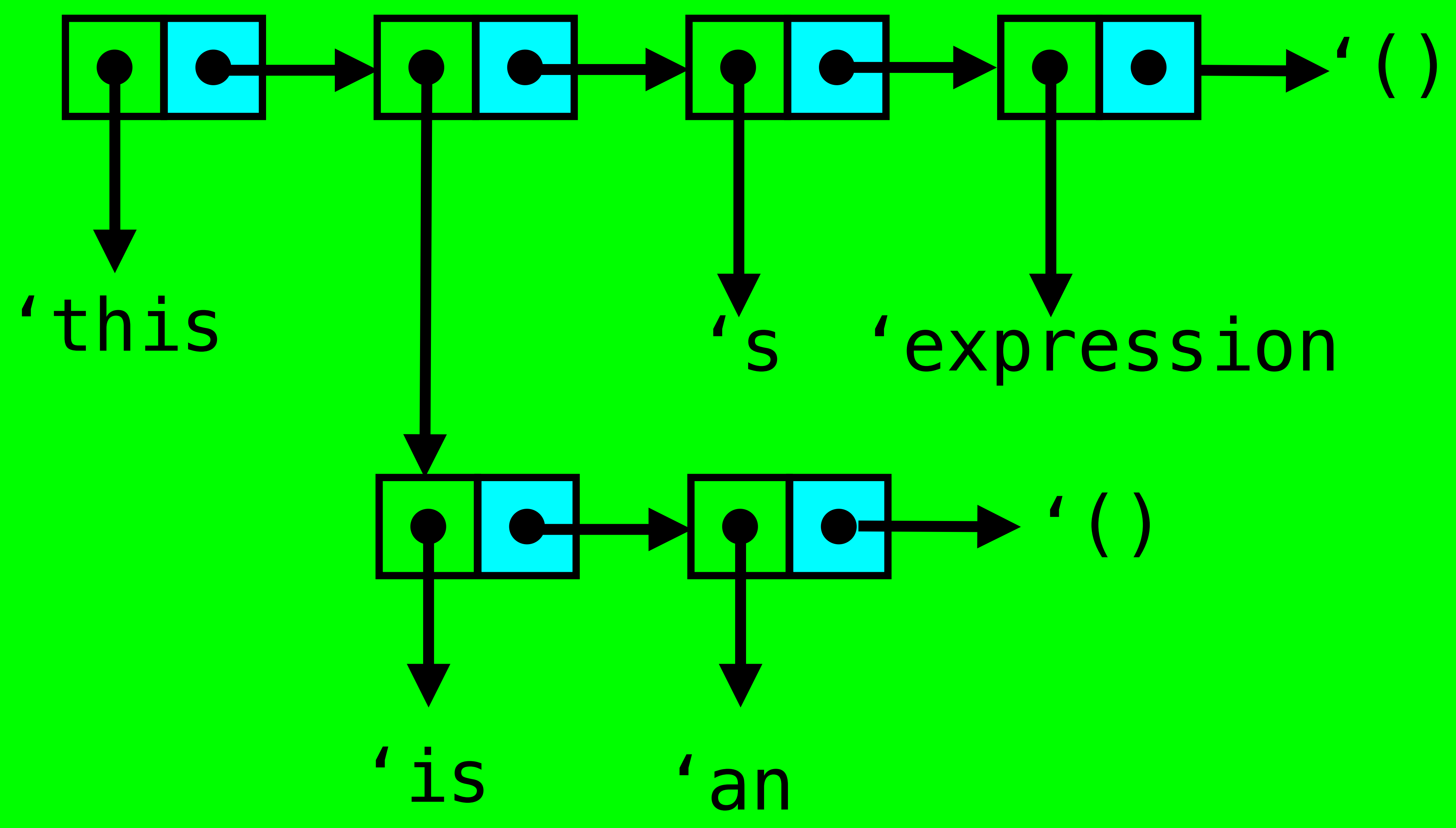
Dot indicates a cons cell of a left and right element

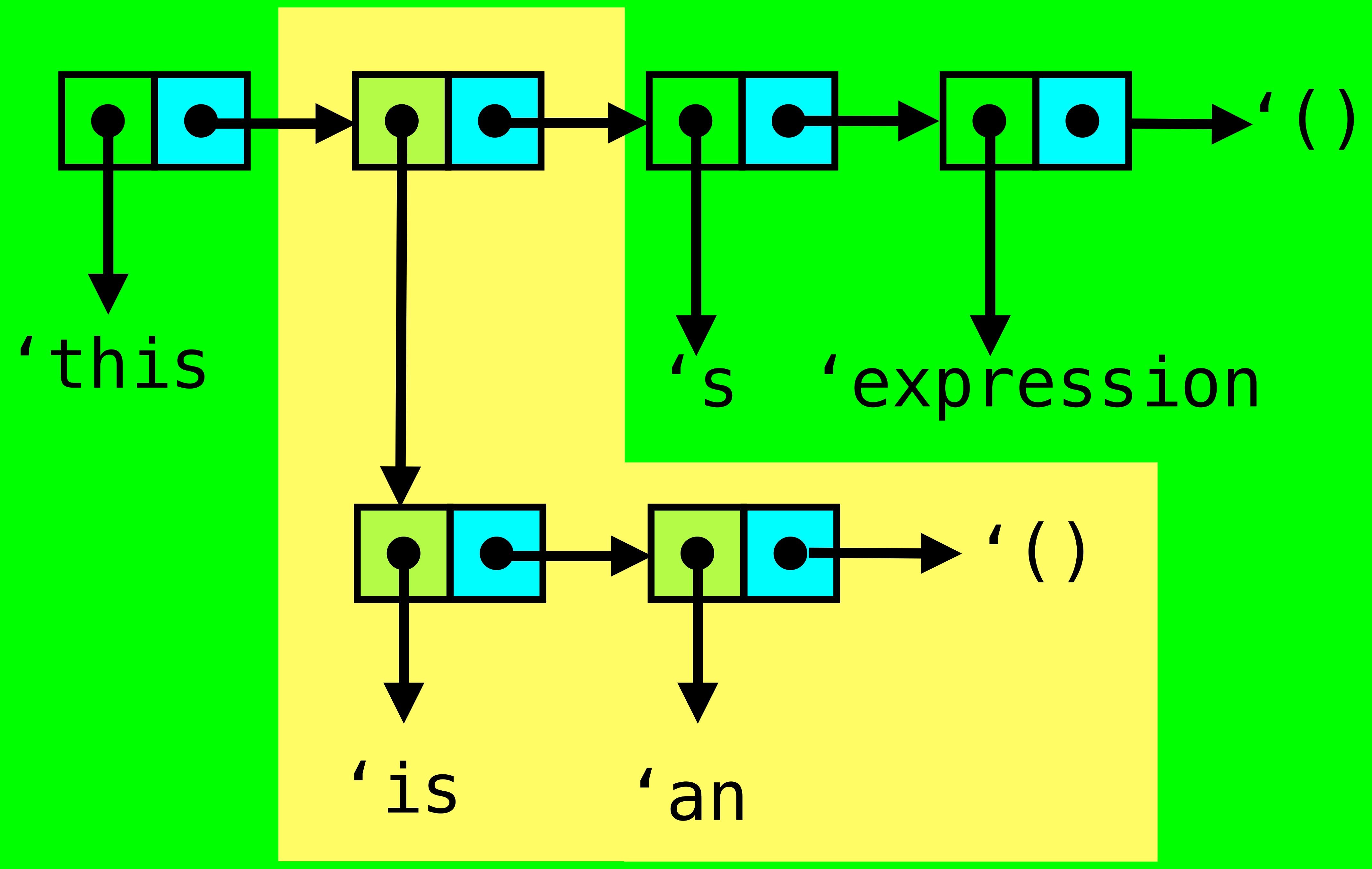
Also can build **compound** expressions

```
‘(this (is an) s expression)
```

Also can build **compound** expressions

`‘(this (is an) s expression)`

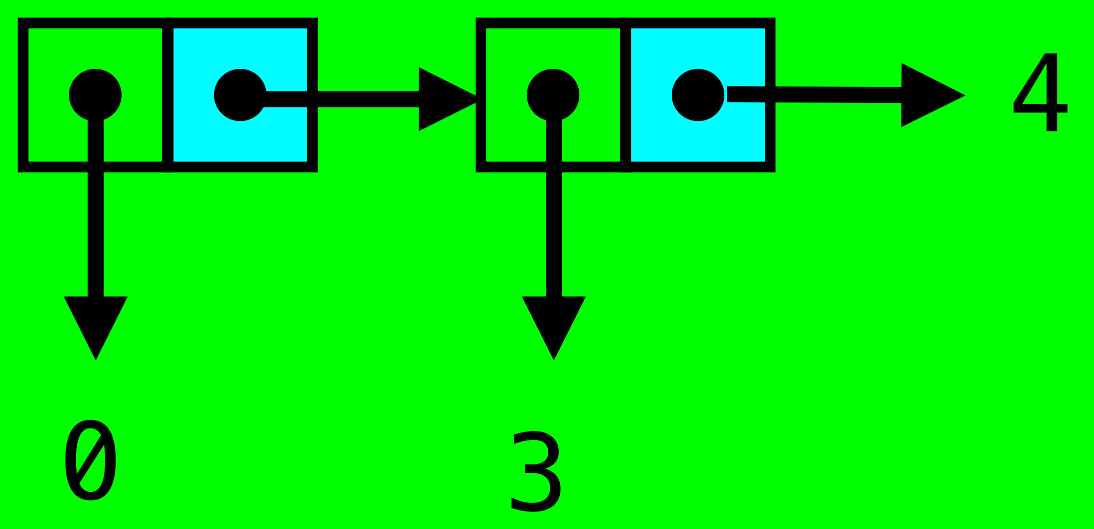



‘()
‘this
‘s
‘expression
‘()
‘is
‘an

Draw the cons diagram for…

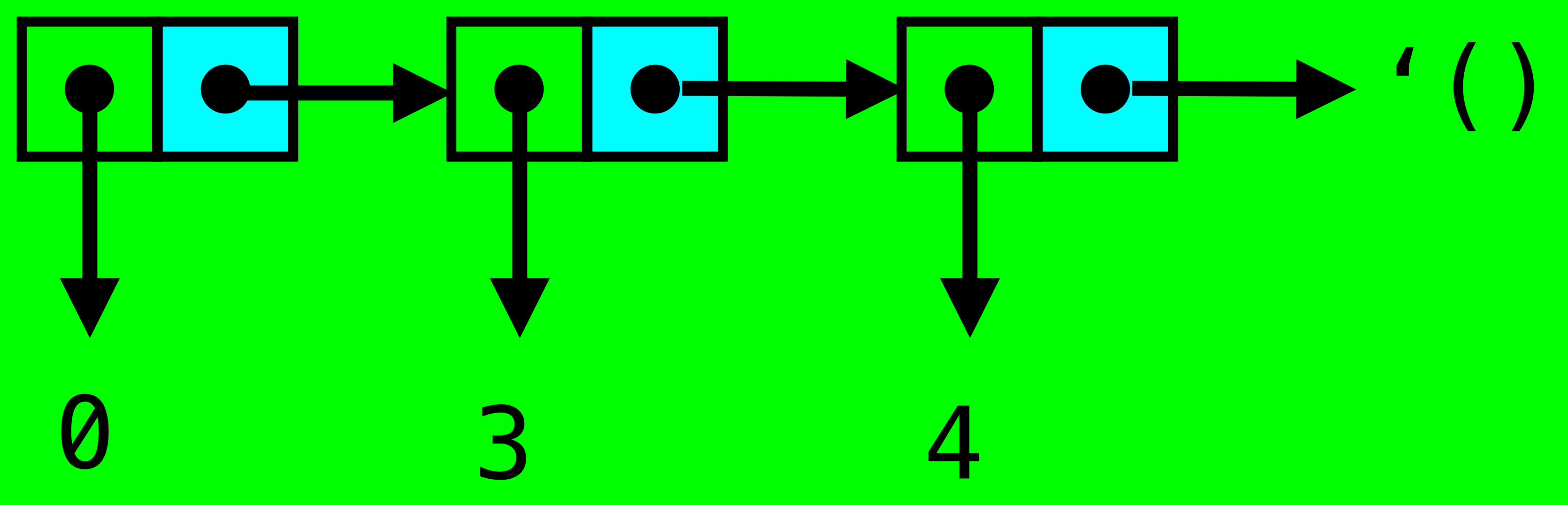
- (cons 0 (cons 3 4))
- Is this a list? If not, what is it?
- (cons 0 (cons 3 (cons 4 ‘())))
- Is this a list? If not, what is it?

(cons 0 (cons 3 4))



This is *not* a list (an improper list)

(cons 0 (cons 3 (cons 4 ‘())))



Mapping over Lists

CIS352 — Spring 2021

Kris Micinski

Project 0: Tic-Tac-Toe

CIS352 — Spring 2021

Kris Micinski

Quasiquoting and Pattern Matching

CIS352 — Spring 2021

Kris Micinski

- Racket **quasi-quotes** build S-expressions nicely
- `` `(,x y 3) `` is equivalent to `(list x ‘y 3)`
  - I.e., Racket splices in values that are unquoted via `,`
  - (quasiquote …), or `…, substitutes any sub-expr `,e` with the return value of `e` within the quoted s-expression

- Works multiple list “levels” deep:
  - `` `(square (point ,x0 ,y0) (point ,x1 ,y1)) ``
- Can unquote arbitrary expressions, not just references:
  - `` `(point ,(+ 1 x0) ,(- 1 y0)) ``

Exercise

Define mk-point and mk-square using Quasi-quotation:

```
(define (mk-point x y)
  (list ‘point x y))

(define (mk-square pt0 pt1)
  (list ‘square pt0 pt1))
```

Exercise

Define mk-point and mk-square using Quasi-quotation:

```
(define (mk-point x y)
  (list ‘point x y))

(define (mk-square pt0 pt1)
  (list ‘square pt0 pt1))
```

```
(define (mk-point x y)
  `(point ,x ,y))

(define (mk-square pt0 pt1)
  `(square ,pt0 ,pt1))
```

- Racket also has **pattern matching**
  - `(match e [pat₀ body₀] [pat₁ body₁]…)`
- Evaluates e and then checks each **pattern**, in order
- Pattern can bind variables, body can use pattern variables

- Many patterns (check docs to learn various useful forms)
- Patterns checked in order, first matching body is executed
    - Later bodies won't be executed, **even if they also match**!
        - **Students make frequent mistakes on this!**
- E.g.,

```
(match '(1 2 3)
       [`(,a ,b) b]
       [`(,a . ,b) b])  ; returns '(2 3)
```

Matching a literal

```
(match e
  [‘hello ‘goodbye]
  [(? number? n) (+ n 1)]
  [(? nonnegative-integer? n)
   (+ n 2)]
  [(cons x y) x]
  [`(,a0 ,a1 ,a2) (+ a1 a2)])
```

(binds n)

```
(match e
  [‘hello ‘goodbye]
  [(? number? n) (+ n 1)]
  [(? nonnegative-integer? n)
    (+ n 2)]
  [(cons x y) x]
  [`(,a0 ,a1 ,a2) (+ a1 a2)])
```

Matches when e evaluates to some number?

```
(match e
  [‘hello ‘goodbye]
  [(? number? n) (+ n 1)]
  [(? nonnegative-integer? n)
   (+ n 2)]
  [(cons x y) x]
  [`(,a0 ,a1 ,a2) (+ a1 a2)])
```

Never matches!

Subsumed by previous case!

```
(match e
  [‘hello ‘goodbye]
  [(? number? n) (+ n 1)]
  [(? nonnegative-integer? n)
    (+ n 2)]
  [(cons x y) x]
  [`(,a0 ,a1 ,a2) (+ a1 a2)])
```

Matches a cons cell, binds x and y

```
(match e
  ['hello 'goodbye]
  [(? number? n) (+ n 1)]
  [(? nonnegative-integer? n)
    (+ n 2)]
  [(cons x y) x]
  [`(,a0 ,a1 ,a2) (+ a1 a2)])
```

Matches a list of length three

Binds first element as `a0`, second as `a1`, etc…

Called a "quasi-pattern"

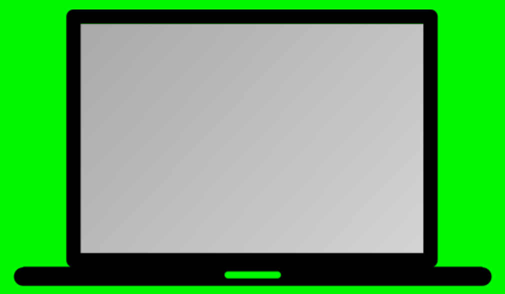
Can also test predicates on bound vars:

```
`(,(? nonnegative-integer? x) ,(? positive? y))
```

```
(match e
  ['hello 'goodbye]
  [(? number? n) (+ n 1)]
  [(? nonnegative-integer? n)
    (+ n 2)]
  [(cons x y) x]
  [`(,a0 ,a1 ,a2) (+ a1 a2)]
  [_ 23])
```

Can also have a **default case** written via **wildcard** _

Exercise



Define a function `foo` that returns:

- twice its argument, if its argument is a number?
- the first two elements of a list, if its argument is a list of length three, as a list
- the string “error” if it is anything else

```
(define (foo x)
  (match x
    [(? …) …]
    …))
```

Exercise

Define a function `foo` that returns:

- twice its argument, if its argument is a number?
- the first two elements of a list, if its argument is a list of length three, as a list
- the string "error" if it is anything else

Answer (one of many)

Observe how quasipatterns and quasiquotes interact

```
(define (foo x)
  (match x
    [(? number? n) (* n 2)]
    [`(,a ,b ,_) `(,a ,b)]
    [_ "error"]))
```

- Using pattern matching, we can build **type predicates**
  - Predicates that specify data formats
- We will **frequently** use these in-lieu of static typing

```
(define (tree? t)
  (match t
    ['empty #t]
    [`(leaf ,v) #t]
    [`(binary ,(? tree?) ,(? tree?)) #t]
    ;; don't forget this!
    [_ #f]))
```

- We can use **define/contract** to specify dynamically-checked **contracts** on functions

```
(define/contract (tree-min t0)
  (-> tree? any/c)
  (match t
    ['empty (error "no min of empty tree")]
    [`(leaf ,v) v]
    [`(binary ,t0 ,t1) (tree-min t0)]))

> (tree-min '(binary (leaf 2) empty))
2
```

```
> (tree-min '(binary 2 empty))
. . tree-min: contract violation
  expected: tree?
  given: '(binary 2 empty)
  in: the 1st argument of
      (-> tree? any/c)
  contract from: (function tree-min)
  blaming: anonymous-module
   (assuming the contract is correct)
```

Squaring every element of a list

```
(define (square-list-values lst)
  (if (null? lst)
      ‘()
      (cons (* (car lst) (car lst))
            (square-list-values (cdr lst)))))
```

Squaring every element of a list

```
(define (square-list-values lst)
  (if (null? lst)
      '()
      (cons (* (car lst) (car lst))
            (square-list-values (cdr lst)))))
```

Defines **base case**

Squaring every element of a list

```
(define (square-list-values lst)
  (if (null? lst)
      '()
      (cons (* (car lst) (car lst))
            (square-list-values (cdr lst)))))
```

Recursive case first computes the square of (car lst)

Squaring every element of a list

```
(define (square-list-values lst)
  (if (null? lst)
      ‘()
      (cons (* (car lst) (car lst))
            (square-list-values (cdr lst)))))
```

Recursive case next recurs on the list’s tail (cdr lst)

Squaring every element of a list

```
(define (square-list-values lst)
  (if (null? lst)
      '()
      (cons (* (car lst) (car lst))
            (square-list-values (cdr lst)))))
```

Recursive case finally extends the *new* tail list

Squaring every element of a list

```
(define (map f lst)
  (if (null? lst)
      ‘()
      (cons (f (car lst))
            (map f (cdr lst)))))

(define (square-list-values lst)
  (map (lambda (x) (* x x)) lst))
```

map takes a (unary) function and list

Squaring every element of a list

```
(define (map f lst)
  (if (null? lst)
      ‘()
      (cons (f (car lst))
            (map f (cdr lst)))))

(define (square-list-values lst)
  (map (lambda (x) (* x x)) lst))
```

```
(define (square-list-values lst)
  (if (null? lst)
      '()
      (cons (* (car lst) (car lst))
            (square-list-values (cdr lst)))))
```

```
(define (map f lst)
  (if (null? lst)
      '()
      (cons (f (car lst))
            (map f (cdr lst)))))
```

```
(define (square-list-values lst)
  (map (lambda (x) (* x x)) lst))
```

We can write the def of map in just one line!

```
(define (map f lst)
  (if (null? lst)
      ‘()
      (cons (f (car lst))
            (map f (cdr lst)))))

(define (square-list-values lst)
  (map (lambda (x) (* x x)) lst))
```

Exercise

Write an implementation of andmap, such that:

```
> (andmap list? ‘((1 2) () (3)))
#t
> (andmap list? ‘((1 . 2) ()))
#f
> (andmap list? ‘(1 2 3))
#f
```

Exercise

Double-check: does your implementation ***short-circuit***? What does your implementation give for:

```
> (andmap list? '())
```

Exercise

Double-check: does your implementation ***short-circuit***? What does your implementation give for:

```
> (andmap list? ‘())
```

```
(define andmap
  (lambda (p? lst)
    (if (null? lst)
        #t
        (and (p? (car lst))
             (andmap p? (cdr lst))))))
```

Tail Calls and Tail Recursion

CIS352 — Spring 2021

Kris Micinski

Practicing Tail Recursion

CIS352 — Spring 2021

Kris Micinski

((lambda (x) x) **((lambda (y) y) 5)**)

((lambda (x) x) 5)

5

Calculating factorial in Racket

```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (sub1 n)))))
```

Calculating factorial in Racket

```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (sub1 n)))))
```

Defines **base case**

Calculating factorial in Racket

```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (sub1 n)))))
```

and **inductive / recursive** case

Calculating factorial in Racket

```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (sub1 n)))))
```

We can think of recursion as “substitution”

```
> (factorial 2)
```

```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (sub1 n)))))
```

We can think of recursion as “substitution”

```
> (factorial 2)
= (if (= 2 0)
      1
      (* 2 (factorial (sub1 2))))
```

Copy defn, substitute for argument `n`

```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (sub1 n)))))
```

We can think of recursion as “substitution”

```
> (factorial 2)
= (if (= 2 0)
      1
      (* 2 (factorial (sub1 2))))
= (if #f 1 (* 2 (factorial (sub1 2))))
= (* 2 (factorial (sub1 2)))
= (* 2 (factorial 1))
= (* 2 (if …))
```

```
…
= (* 2 (if (= 2 0)
        1
        (* n (factorial (sub1 2)))))
= (* 2 (factorial 1))
= …
= (* 2 (* 1 1))
= (* 2 1)
= 2
```

Notice we're building a big stack of calls to *

Tail Calls

- Unlike calls in general, ***tail calls*** do not affect the stack:
  - Tail calls *do not grow* (or shrink) the stack.
    - They are more like a goto/jump than a normal call.

Tail Position

- A subexpression is in ***tail position*** if it's:
  - The last subexpression to run, whose return value is also the value for its parent expression
  - In `(let ([x rhs]) body)`; <u>body</u> is in *tail position…*
  - In `(if grd thn els)`; <u>thn</u> *&* <u>els</u> are in *tail position…*

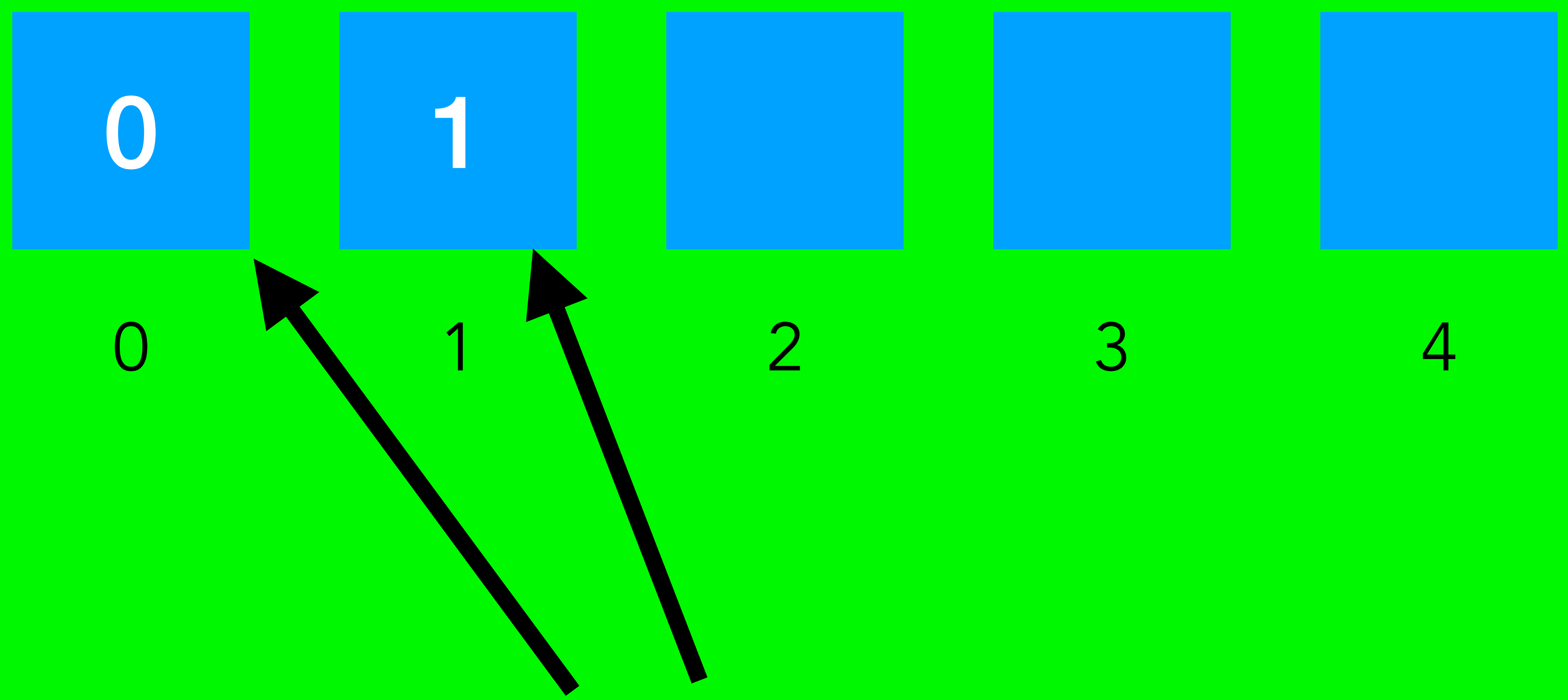
Tail Recursion

- A function is ***tail recursive*** if all recursive calls in tail position
- Tail-recursive functions are analogous to loops in imperative langs

Tail calls / tail recursion

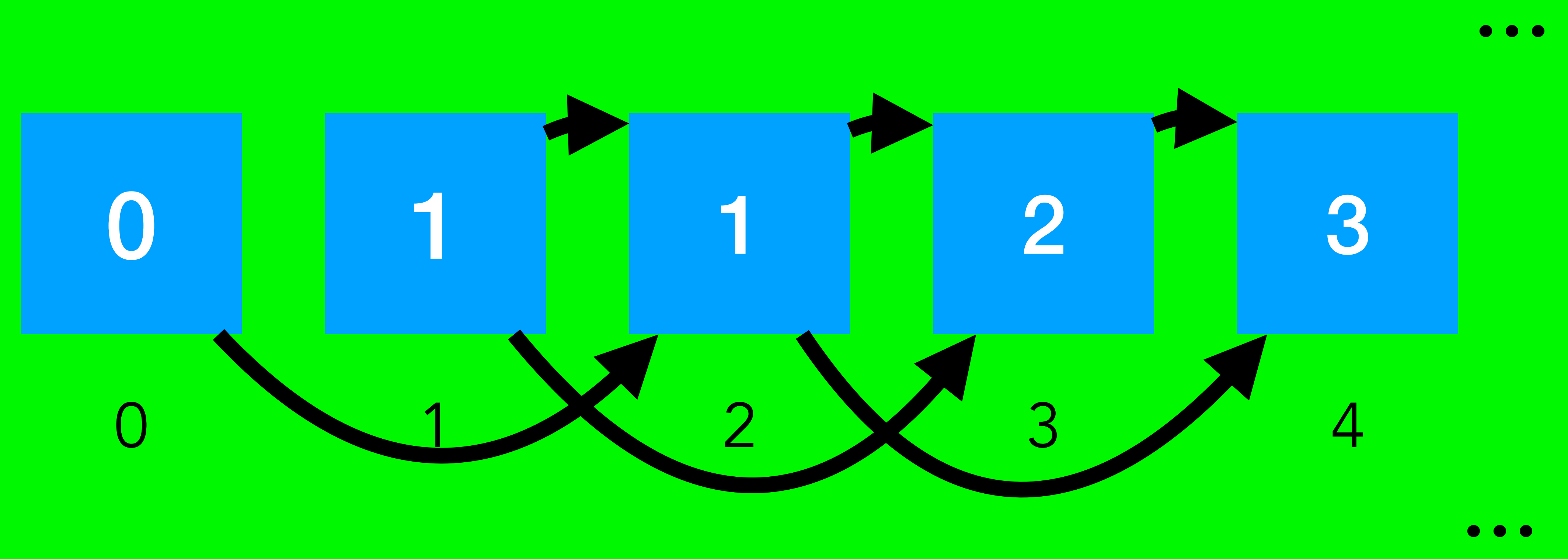
- Unlike calls in general, ***tail calls*** do not affect the stack:
  - Tail calls *do not grow* (or shrink) the stack.
    - They are more like a goto/jump than a normal call.
- A function is ***tail recursive*** if all recursive calls in tail position
- Tail-recursive functions are analogous to loops in imperative langs

Instead, use ***dynamic programming:***
design a recursive solution top-down, but implement
as a bottom-up algorithm!

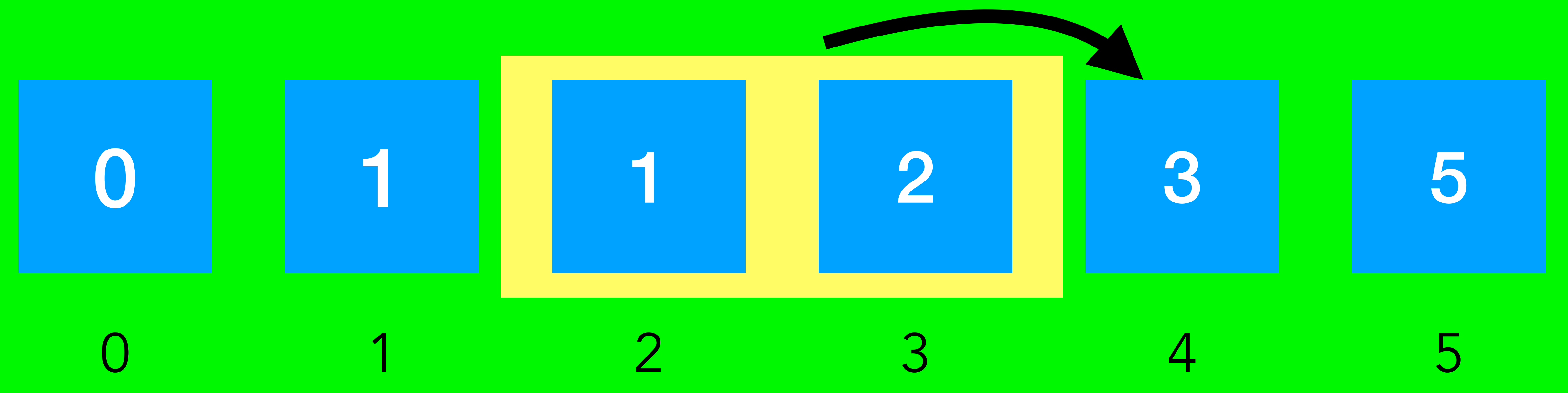


Start with first two, then build up

Instead, use ***dynamic programming:***
design a recursive solution top-down, but implement as a bottom-up algorithm!



Key idea: only need to look at **two most recent** numbers



Accumulate via arguments

```
(define (fib-h i n0 n1)
  (if (= i 0)
      n0
      (fib-h (- i 1) n1 (+ n0 n1))))

(define (fib n) (fib-h n 0 1))
```

Exercise

```
(define (fib-h i n0 n1)
  (if (= i 0)
      n0
      (fib-h (- i 1) n1 (+ n0 n1))))

(define (fib n) (fib-h n 0 1))
```

Question: what is the runtime complexity of `fib`?

Exercise

```
(define (fib-h i n0 n1)
  (if (= i 0)
      n0
      (fib-h (- i 1) n1 (+ n0 n1))))

(define (fib n) (fib-h n 0 1))
```

Answer: O(n), fib-helper runs from `n` to `0`

Consider how `fib-h` executes

```
(define (fib-h i n0 n1)
  (if (= i 0)
      n0
      (fib-h (- i 1) n1 (+ n0 n1))))

(define (fib n) (fib-h n 0 1))
```

```
(fib-helper 3 0 1)
= (if (= 3 0) 0 (fib-h (- 3 1) 1 (+ 0 1)))
= …
= (fib-h 2 1 1)
= (if (= 2 0) 1 (fib-h (- 2 1) 1 (+ 1 1)))
= …
= (fib-h 1 1 2)
```

Notice that we don't get the "stacking" behavior: recursive calls don't grow the stack

This is because `fib-h` is **tail recursive**

```
(define (fib-h i n0 n1)
  (if (= i 0)
      n0
      (fib-h (- i 1) n1 (+ n0 n1))))

(define (fib n) (fib-h n 0 1))
```

Intuitively: a callsite is in **tail-position** if it is the **last thing** a function will do before exiting

(We call these **tail calls**)

This is because `fib-h` is **tail recursive**

Both of these are tail calls

```
(define (fib-h i n0 n1)
  (if (= i 0)
      n0
      (fib-h (- i 1) n1 (+ n0 n1))))

(define (fib n) (fib-h n 0 1))
```

Intuitively: a callsite is in **tail-position** if it is the **last thing** a function will do before exiting

(We call these **tail calls**)

Tail calls / tail recursion

- Unlike calls in general, ***tail calls*** do not affect the stack:
  - Tail calls *do not grow* (or shrink) the stack.
    - They are more like a goto/jump than a normal call.
- A subexpression is in ***tail position*** if it's the last subexpression to run, whose return value is also the value for its parent expression:
  - In `(let ([x rhs]) body)`; <u>body</u> is in *tail position…*
  - In `(if grd thn els)`; <u>thn</u> & <u>els</u> are in *tail position…*
- A function is ***tail recursive*** if all recursive calls in tail position
- Tail-recursive functions are analogous to loops in imperative langs

Exercise

Which of the following is tail recursive?

```
(define (length-0 l)
  (if (null? l)
      0
      (+ 1 (length-0 (cdr l)))))
```

```
(define (length-1 l n)
  (if (null? l)
      n
      (length-1 (cdr l) (+ n 1))))
```

Exercise

Answer

```
(define (length-0 l)
  (if (null? l)
      0
      (+ 1 (length-0 (cdr l)))))
```

Not tail recursive

Adds (+ 1 _) operation to stack

```
(define (length-1 l n)
  (if (null? l)
      n
      (length-1 (cdr l) (+ n 1))))
```

Is tail recursive!

Call to length-1 in tail position

Folding over Lists

CIS352 — Spring 2021

Kris Micinski

Iterating over a list to accumulate a result is one of the most typical programming patterns

Iterating over a list to accumulate a result is one of the most typical programming patterns

```
(define (sum-list l)
  (match l
    ['() 0]
    [`(,hd . ,tl) (+ hd (sum-list tl))]))
```

Iterating over a list to accumulate a result is one of the most typical programming patterns

```
(define (list-product l)
  (match l
    ['() 1]
    [`(,hd . ,tl) (* hd (list-product tl))]))
```

Iterating over a list to accumulate a result is one of the most typical programming patterns

```
(define (filter f l)
  (match l
    ['() '()]
    [`(,hd . ,tl)
     (if (f hd)
         (cons hd (filter f tl))
         (filter f tl))]))
```

What do all these functions have in common?

```
(define (list-product l)
  (match l
    ['() 1]
    [`(,hd . ,tl) (* hd (list-product tl))]))
```

```
(define (sum-list l)
  (match l
    ['() 0]
    [`(,hd . ,tl) (+ hd (sum-list tl))]))
```

```
(define (filter f l)
  (match l
    ['() '()]
    [`(,hd . ,tl)
     (if (f hd) (cons hd (filter f tl)) (filter f tl))]))
```

Each matches on the list

```
(define (list-product l)
  (match l
    ['() 1]
    [`(,hd . ,tl) (* hd (list-product tl))]))
```

```
(define (sum-list l)
  (match l
    ['() 0]
    [`(,hd . ,tl) (+ hd (sum-list tl))]))
```

```
(define (filter f l)
  (match l
    ['() '()]
    [`(,hd . ,tl)
     (if (f hd) (cons hd (filter f tl)) (filter f tl))]))
```

Each returns an **initial** value

```
(define (list-product l)
  (match l
    ['() 1]
    [`(,hd . ,tl) (* hd (list-product tl))]))
```

```
(define (sum-list l)
  (match l
    ['() 0]
    [`(,hd . ,tl) (+ hd (sum-list tl))]))
```

```
(define (filter f l)
  (match l
    ['() '()]
    [`(,hd . ,tl)
     (if (f hd) (cons hd (filter f tl)) (filter f tl))]))
```

Each of them makes a recursive call and then **combines the result with hd**

```
(define (list-product l)
  (match l
    ['() 1]
    [`(,hd . ,tl) (* hd (list-product tl))]))
```

```
(define (sum-list l)
  (match l
    ['() 0]
    [`(,hd . ,tl) (+ hd (sum-list tl))]))
```

```
(define (filter f l)
  (match l
    ['() '()]
    [`(,hd . ,tl)
     (if (f hd) (cons hd (filter f tl)) (filter f tl))]))
```

Let’s think about how sum-list operates over lists…

```
(define (sum-list l)
  (match l
    ['() 0]
    [`(,hd . ,tl) (+ hd (sum-list tl))]))
```

```
(sum-list (cons 1 (cons 2 ‘())))
     … => (+    1 (+    2 0))
```

You can think of this as replacing cons with + and ‘() with 0

Now let's look at list-product

```
(define (list-product l)
  (match l
    ['() 1]
    [`(,hd . ,tl) (* hd (list-product tl))]))
```

```
(list-product (cons 1 (cons 2 ‘())))
         … => (*    1 (*    2 1))
```

You can think of ***this*** as replacing cons with * and ‘() with 1

```
(fold f i      (cons 1 (cons 2 ‘())))
         … => (f    1 (f    2 i))
```

Folds abstract this common pattern:

- Iterating over list to **accumulate** some result
- Some **default** or **initial** value to handle empty list
- Some two-argument **reducer** function
  - Combines first element w/ processed tail

```
(define (fold reducer init lst)
  (match lst
    ['() init]
    [`(,hd . ,tl)
     (reducer hd (fold reducer init tl))]))
```

Exercise

Use fold to write sum-list

```
(define (fold reducer init lst)
  (match lst
    ['() init]
    [`(,hd . ,tl)
     (reducer hd (fold reducer init tl))]))
```

Exercise

Use fold to write list-product

```
(define (fold reducer init lst)
  (match lst
    ['() init]
    [`(,hd . ,tl)
     (reducer hd (fold reducer init tl))]))
```

Exercise

Use fold to write filter-list

```
(define (fold reducer init lst)
  (match lst
    ['() init]
    [`(,hd . ,tl)
     (reducer hd (fold reducer init tl))]))
```

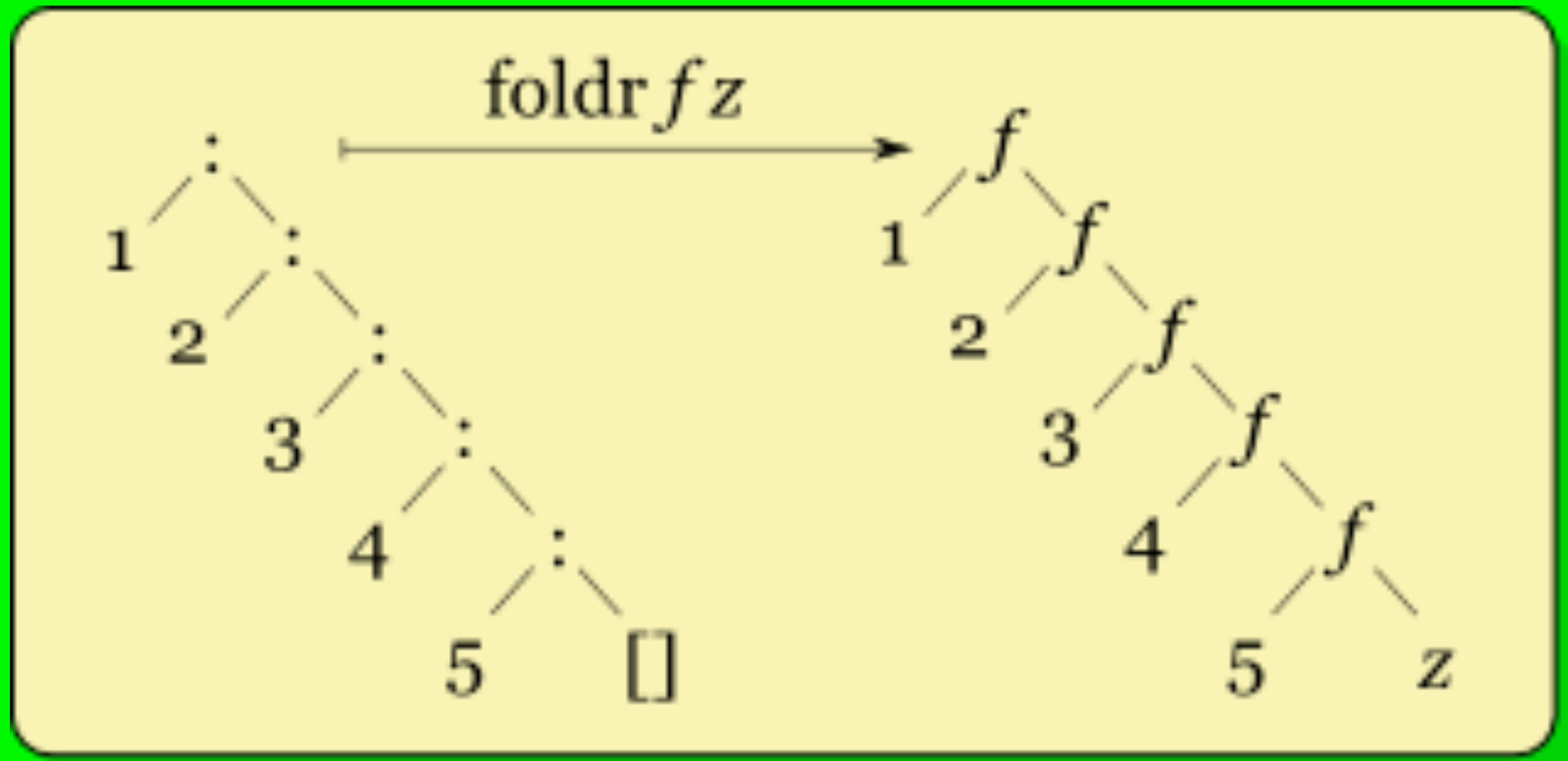
This version of fold is **direct-style**, meaning it will push stack frames

```
(define (foldr reducer init lst)
  (match lst
    ['() init]
    [`(,hd . ,tl)
     (reducer hd (fold reducer init tl))]))
```

This version of fold is **direct-style**, meaning it will push stack frames

```
(define (foldr reducer init lst)
  (match lst
    ['() init]
    [`(,hd . ,tl)
     (reducer hd (fold reducer init tl))]))
```

Traditionally this is called a “right” fold because it bottoms out at the end (right side) of the list, and reconstructs back up.

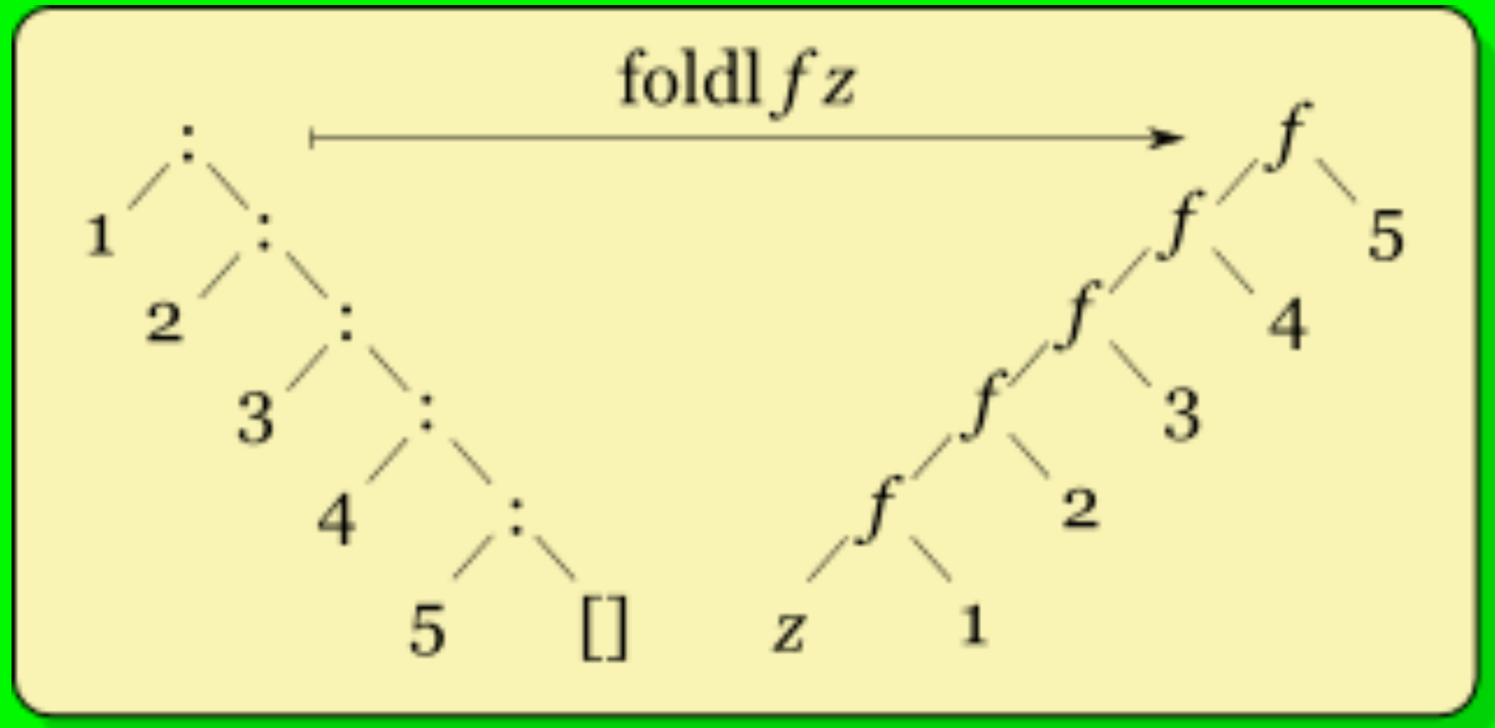


*** Diagram from the Haskell wiki**

We can also write a **tail-recursive** version of fold by swapping the argument order to reducer

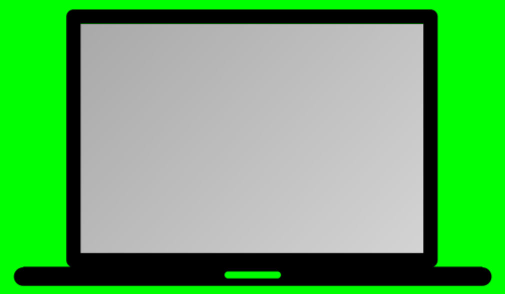
```
(define (foldl reducer acc lst)
  (match lst
    ['() acc]
    [`(,hd . ,tl)
     (fold reducer (reducer hd acc) tl)]))
```

This is called a **left fold** because it “starts” from the left (reducer will be called on first element w/ the “zero”)



* Diagram from the Haskell wiki

Exercise



Use foldl to write reverse

```
(define (foldl reducer acc lst)
  (match lst
    ['() acc]
    [`(,hd . ,tl)
     (fold reducer (reducer hd acc) tl)]))
```

Biggest takeaways for you:

- Consider using fold when possible
- Use Racket's foldl or foldr
  - Mostly the same, but process list differently
- You need a two argument **reducer** function
- You need an **initial value**

Interpreting IfArith

CIS352 — Spring 2021

Kris Micinski

Today, we're going to start building our **own** languages

We're going to do this by writing **interpreters**

To build a programming language, we need two things:

A **syntax** for the language (and the ability to **parse** it)

A **semantics** for the language. Typically either an **interpreter** or a **compiler**

For this class, all of our programs are going to be written as Racket datums

We specify syntax via a predicate that uses pattern matching

This means we can just write programs in our language just by building data in Racket

Here is the first language we will define:

```
(define (expr? e)
  (match e
    [(? integer? n) #t]
    [`(plus ,(? expr? e0) ,(? expr? e1)) #t]
    [`(div ,(? expr? e0) ,(? expr? e1)) #t]
    [`(not ,(? expr? e-guard)) #t]
    [`(if ,(? expr? e0) ,(? expr? e1) ,(? expr? e2)) #t]
    [_ #f]))
```

```
(define (expr? e)
  (match e
    [(? integer? n) #t]
    [`(plus ,(? expr? e0) ,(? expr? e1)) #t]
    [`(div ,(? expr? e0) ,(? expr? e1)) #t]
    [`(not ,(? expr? e-guard)) #t]
    [`(if ,(? expr? e0) ,(? expr? e1) ,(? expr? e2)) #t]
    [_ #f]))
```

"Any integer is a program in our language."

```
(define (expr? e)
  (match e
    [(? integer? n) #t]
    [`(plus ,(? expr? e0) ,(? expr? e1)) #t]
    [`(div ,(? expr? e0) ,(? expr? e1)) #t]
    [`(not ,(? expr? e-guard)) #t]
    [`(if ,(? expr? e0) ,(? expr? e1) ,(? expr? e2)) #t]
    [_ #f]))
```

"If e0 is an expression in our language, and e1 is an expression in our language, `(plus ,e0 ,e1) is, too."

```
(define (expr? e)
  (match e
    [(? integer? n) #t]
    [`(plus ,(? expr? e0) ,(? expr? e1)) #t]
    [`(div ,(? expr? e0) ,(? expr? e1)) #t]
    [`(not ,(? expr? e-guard)) #t]
    [`(if ,(? expr? e0) ,(? expr? e1) ,(? expr? e2)) #t]
    [_ #f]))
```

Here are some example expressions:

```
‘(plus 1 (div 2 3))
'(if 0 (plus 1 2) (div 2 2))
'(if 0 (plus 1 (div 2 3)) (if 1 (plus 2 3) 0))
```

IMPORTANT NOTE

We are defining a **new language** by **using** Racket. But our language is **not** Racket. In Racket, booleans are #t and #f. In **our** language, we will use 0 to represent false and non-0 to represent true (as in C).

Again, because this is confusing

When writing interpreters, always be careful to mentally separate the **language you are defining** and the language you are using to build the interpreter (Racket).

This can become confusing as the languages we build will "look like" Racket. Try to be mindful.

Key idea: write an **interp** function that takes in expressions as an argument, and returns **Racket** values

Key idea: write an **interp** function that takes in expressions as an argument, and returns **Racket** values

The "result" of programs will be a Racket integer:

```
(define value? integer?)
```

Key idea: write an **interp** function that takes in expressions as an argument, and returns **Racket** values

The “result” of programs will be a Racket integer:

```
(define value? integer?)

(define/contract (evaluate e)
  (-> expr? value?)
  ‘todo)
```

What should the following return…?
Remember, this is our own **new language we are defining, not necessarily Racket**

```
(evaluate '(plus 1 2))
=> 3
(evaluate '(if 0 (plus 1 2) (div 2 2)))
=> ‘todo
(evaluate '(if 1 (div 4 3) (plus 1 -1)))
=> ‘todo
```

What should the following return…?

Remember, this is our own **new language we are defining, not necessarily Racket**

```
(evaluate '(plus 1 2))
=> 3
(evaluate '(if 0 (plus 1 2) (div 2 2)))
=> 1
(evaluate '(if 1 (div 4 3) (plus 1 -1)))
=> 4/3
```

Now, let's build **evaluate** ourselves

In this lecture, we built a **metacircular** interpreter

Important Definition

A metacircular interpreter is an interpreter which uses features of a "host" language to define the semantics of a "target" language

Which features of Racket did we use to define our language…?

Important Definition

A metacircular interpreter is an interpreter which uses features of a "host" language to define the semantics of a "target" language

```
(define (evaluate e)
  (match e
    [(? integer? n) n]
    [`(plus ,(? expr? e0) ,(? expr? e1))
     (+ (evaluate e0) (evaluate e1))]
    …
```

Notice how we **inherit** the definition of + from Racket

John Reynolds introduced metacircular interpreters in 1978. One key idea: metacircular interpreters inherit properties of their host language!

Definitional Interpreters for Higher-Order Programming Languages

John C. Reynolds, Syracuse University

Reynolds/Definitional

72-1

1

Higher-order programming languages (i.e., languages in which procedures or labels can occur as values) are usually defined by interpreters which are themselves written in a programming language based on the lambda calculus (i.e., an applicative language such as pure LISP). Examples include McCarthy's definition of LISP, Landin's SECD machine, the Vienna definition of PL/I, Reynolds' definitions of GEDANKEN, and recent unpublished work by L. Morris and C. Wadsworth. Such definitions can be classified according to whether the interpreter contains higher-order functions, and whether the order of application (i.e., call-by-value versus call-by-name) in the defined language

INTRODUCTION

An important and frequently used method of defining a programming language is to give an interpreter for the language which is written in a second, hopefully better understood language. (We will call these two languages the *defined* and *defining* languages, respectively.) In this paper, we will describe and classify several varieties of such interpreters, and show how they may be derived from one another by informal but constructive methods. Although our approach to "constructive classification" is original, the paper is basically an attempt to review and systematize previous work in the field, and we have

Note: our interpreter is **direct-style**, it is **not** tail recursive

```
(define (evaluate e)
  (match e
    [(? integer? n) n]
    [`(plus ,(? expr? e0) ,(? expr? e1))
     (+ (evaluate e0) (evaluate e1))]
    …
```

This means we are relying on Racket's **stack** as well

We will later see how to eliminate the need for this

Natural Deduction for IfArith

CIS352 — Spring 2021

Kris Micinski

In this lecture, we'll introduce **natural deduction**

Natural deduction is a mathematical formalism that helps ground the ideas in metacircular interpreters

Natural deduction first used in mathematical logic, to specify **proofs** using inductive data

We will use natural deduction as a framework for specifying semantics of various languages throughout the course

Introduction Rules	Elimination Rules
$\dfrac{\dfrac{}{\vdash^N A}u \quad \cdots \quad \vdash^N B}{\vdash^N A \supset B}\supset\mathrm{I}^u$	$\dfrac{\vdash^N A \supset B \qquad \vdash^N A}{\vdash^N B}\supset\mathrm{E}$
$\dfrac{\dfrac{}{\vdash^N A}u \quad \cdots \quad \vdash^N p}{\vdash^N \neg A}\neg\mathrm{I}^{p,u}$	$\dfrac{\vdash^N \neg A \qquad \vdash^N A}{\vdash^N C}\neg\mathrm{E}$
$\dfrac{\vdash^N [a/x]A}{\quad}\forall\mathrm{I}^a$	$\dfrac{\vdash^N \forall x.\, A}{\quad}\forall\mathrm{E}$

When we specify the semantics of a language using natural deduction, we give its semantics via a set of **inference rules**

Rules read: if the thing on the **top** is true, then the thing on the **bottom** is also true.

This rule says: "if c is an integer (mathematically: c ∈ $\mathbb{Q}$), then c evaluates to c."

$$\textbf{Const}: \frac{c \in \mathbb{Q}}{c \Downarrow c}$$

Note: the notation e ⇓ v is read "e evaluates to v."

Some rules will have more than one **antecedent** (thing on the top).

You read these: "if the first thing, and second thing, and … are **all** true, then the thing on the bottom is true."

$$\textbf{Plus} : \frac{e_0 \Downarrow n_0 \quad e_1 \Downarrow n_1 \quad n' = n_0 + n_1}{(\text{plus } e_0\ e_1) \Downarrow n'}$$

"If $e_0 \Downarrow n_0$, and $e_1 \Downarrow n_1$, and n' = n_0 + n1, **then** I can say (plus e_0 e_1) $\Downarrow$ n'."

$$\textbf{Plus}: \frac{e_0 \Downarrow n_0 \quad e_1 \Downarrow n_1 \quad n' = n_0 + n_1}{(\mathsf{plus}\ e_0\ e_1) \Downarrow n'}$$

$$\textbf{Const} : \frac{c \in \mathbb{Q}}{c \Downarrow c} \qquad \textbf{Plus} : \frac{e_0 \Downarrow n_0 \quad e_1 \Downarrow n_1 \quad n' = n_0 + n_1}{(\mathsf{plus}\ e_0\ e_1) \Downarrow n'}$$

$$\textbf{Div} : \frac{e_0 \Downarrow n_0 \quad e_1 \Downarrow n_1 \quad n' = n_0 / n_1}{(\mathsf{div}\ e_0\ e_1) \Downarrow n'}$$

The natural deduction rule for **div** is similar

$$\textbf{Const} : \frac{c \in \mathbb{Q}}{c \Downarrow c} \qquad \textbf{Plus} : \frac{e_0 \Downarrow n_0 \quad e_1 \Downarrow n_1 \quad n' = n_0 + n_1}{(\mathsf{plus}\ e_0\ e_1) \Downarrow n'}$$

$$\textbf{Div} : \frac{e_0 \Downarrow n_0 \quad e_1 \Downarrow n_1 \quad n' = n_0 / n_1}{(\mathsf{div}\ e_0\ e_1) \Downarrow n'}$$

$$\textbf{Not}_\mathbf{0} : \frac{e \Downarrow 0}{(\mathsf{not}\ e) \Downarrow 1} \qquad \textbf{Not}_\mathbf{1} : \frac{e \Downarrow n \quad n \neq 0}{(\mathsf{not}\ e) \Downarrow 0}$$

We have **two** rules for not

Natural Deduction Rules for IfArith

$$\textbf{Const}: \frac{c \in \mathbb{Q}}{c \Downarrow c} \qquad \textbf{Plus}: \frac{e_0 \Downarrow n_0 \quad e_1 \Downarrow n_1 \quad n' = n_0 + n_1}{(\mathsf{plus}\ e_0\ e_1) \Downarrow n'}$$

$$\textbf{Div}: \frac{e_0 \Downarrow n_0 \quad e_1 \Downarrow n_1 \quad n' = n_0 / n_1}{(\mathsf{div}\ e_0\ e_1) \Downarrow n'}$$

$$\textbf{Not}_\mathbf{0}: \frac{e \Downarrow 0}{(\mathsf{not}\ e) \Downarrow 1} \qquad \textbf{Not}_\mathbf{1}: \frac{e \Downarrow n \quad n \neq 0}{(\mathsf{not}\ e) \Downarrow 0}$$

$$\textbf{If}_\mathbf{T}: \frac{e_0 \Downarrow 0 \quad e_1 \Downarrow n'}{(\mathsf{if}\ e_0\ e_1\ e_2) \Downarrow n'} \qquad \textbf{If}_\mathbf{F}: \frac{e_0 \Downarrow n \quad n = 0 \quad e_2 \Downarrow n'}{(\mathsf{if}\ e_0\ e_1\ e_2) \Downarrow n'}$$

Question: Now that we have the rules, what can we do with them?

Answer: Use them to **formally prove** that some program calculates some result

Let's say I want to prove that the following program evaluates to 4:

```
(if (plus 1 -1) 3 4)
```

What rule could go here..?

$$\frac{???}{\left(\text{if } (\text{plus } 1 \ \ -1) \ 3 \ 4\right) \Downarrow 4}$$

$$\mathbf{If_T} : \frac{e_0 \Downarrow n \quad n \neq 0 \quad e_1 \Downarrow n'}{(\mathsf{if}\ e_0\ e_1\ e_2) \Downarrow n'} \quad \mathbf{If_F} : \frac{e_0 \Downarrow 0 \quad e_2 \Downarrow n'}{(\mathsf{if}\ e_0\ e_1\ e_2) \Downarrow n'}$$

$$\frac{???}{\left(\mathsf{if}\ (\mathsf{plus}\ 1\ \ -1)\ 3\ 4\right) \Downarrow 4}$$

$$\textbf{If}_{\textbf{T}} : \frac{e_0 \Downarrow n \quad n \neq 0 \quad e_1 \Downarrow n'}{(\text{if } e_0\ e_1\ e_2) \Downarrow n'} \quad \textbf{If}_{\textbf{F}} : \frac{e_0 \Downarrow 0 \quad e_2 \Downarrow n'}{(\text{if } e_0\ e_1\ e_2) \Downarrow n'}$$

$$\frac{???}{\big(\text{if } (\text{plus } 1\ \ -1)\ 3\ 4\big) \Downarrow 4}$$

To apply a natural-deduction rule,
we must perform **unification**

There can be no variables in the resulting unification!

$$\mathbf{If_F} : \frac{e_0 \Downarrow 0 \quad e_2 \Downarrow n'}{(\text{if } e_0\ e_1\ e_2) \Downarrow n'}$$

$$\frac{(\text{plus } 1\ \ -1) \Downarrow 0 \qquad\qquad 4 \Downarrow 4}{\left(\text{if } (\text{plus } 1\ \ -1)\ 3\ 4\right) \Downarrow 4}$$

We perform unification:

e_0: (plus 1 -1), e_1: 3

e_2: 4, n′: 4

Not done yet, now we have to prove **these** things

$$\frac{(\text{plus } 1\ \ {-1}) \Downarrow 0 \qquad 4 \Downarrow 4}{\left(\text{if } (\text{plus } 1\ \ {-1})\ 3\ 4\right) \Downarrow 4}$$

Why can we say $4 \Downarrow 4$? Because of the **Const** rule

$$\frac{(\text{plus } 1 \ \ -1) \Downarrow 0 \qquad \dfrac{4 \in \mathbb{Q}}{4 \Downarrow 4}}{\big(\text{if } (\text{plus } 1 \ \ -1) \ 3 \ 4\big) \Downarrow 4}$$

We're not done yet, because **plus** requires an antecedent:

$$\textbf{Plus}: \frac{e_0 \Downarrow n_0 \quad e_1 \Downarrow n_1 \quad n' = n_0 + n_1}{(\text{plus } e_0\ e_1) \Downarrow n'}$$

$$\frac{(\text{plus } 1\ \ -1) \Downarrow 0 \qquad \dfrac{4 \in \mathbb{Q}}{4 \Downarrow 4}}{\left(\text{if } (\text{plus } 1\ \ -1)\ 3\ 4\right) \Downarrow 4}$$

But we're **still** not done, because we need to finish these three

$$\dfrac{\dfrac{1 \Downarrow 1 \quad -1 \Downarrow -1 \quad 1 + -1 = 0}{(\mathsf{plus}\ 1\ -1) \Downarrow 0} \qquad \dfrac{4 \in \mathbb{Q}}{4 \Downarrow 4}}{\left(\mathsf{if}\ (\mathsf{plus}\ 1\ -1)\ 3\ 4\right) \Downarrow 4}$$

Things that are simply true from algebra require no antecedents, we take them as "axioms."

$$
\dfrac{\dfrac{\dfrac{1 \in \mathbb{Q}}{1 \Downarrow 1} \quad \dfrac{-1 \in \mathbb{Q}}{-1 \Downarrow -1} \quad \dfrac{}{1 + -1 = 0}}{(\mathsf{plus}\ 1\ -1) \Downarrow 0} \qquad \dfrac{4 \in \mathbb{Q}}{4 \Downarrow 4}}{\left(\mathsf{if}\ (\mathsf{plus}\ 1\ -1)\ 3\ 4\right) \Downarrow 4}
$$

This is a complete proof that the program computes 4

$$\dfrac{\dfrac{\dfrac{1 \in \mathbb{Q}}{1 \Downarrow 1} \quad \dfrac{-1 \in \mathbb{Q}}{-1 \Downarrow -1} \quad \dfrac{}{1 + -1 = 0}}{(\mathsf{plus}\ 1\ -1) \Downarrow 0} \qquad \dfrac{4 \in \mathbb{Q}}{4 \Downarrow 4}}{\left(\mathsf{if}\ (\mathsf{plus}\ 1\ -1)\ 3\ 4\right) \Downarrow 4}$$

Question: could you write this proof..? What would happen if you tried…?

$$\frac{???}{\left(\text{if (plus } 1 \;\; -1)\ 3\ 4 \Downarrow 3\right)}$$

$$\mathbf{If_T} : \frac{e_0 \Downarrow n \quad n \neq 0 \quad e_1 \Downarrow n'}{(\text{if } e_0\ e_1\ e_2) \Downarrow n'} \quad \mathbf{If_F} : \frac{e_0 \Downarrow 0 \quad e_2 \Downarrow n'}{(\text{if } e_0\ e_1\ e_2) \Downarrow n'}$$

$$\frac{:(}{\left(\text{if } (\text{plus } 1\ \ -1)\ 3\ 4\right) \Downarrow 3}$$

Answer: you **can't** write this proof, because IfT will only let you evaluate e1 when e0 is non-0!

$$\frac{???}{\big(\textsf{plus}\ (\textsf{plus}\ 0\ 1)\ 2\big) \Downarrow 3} \qquad \frac{???}{(\textsf{if}\ 1\ (\textsf{div}\ 1\ 1)\ 2) \Downarrow 1}$$

$$\textbf{Const} : \frac{c \in \mathbb{Q}}{c \Downarrow c} \qquad \textbf{Plus} : \frac{e_0 \Downarrow n_0 \quad e_1 \Downarrow n_1 \quad n' = n_0 + n_1}{(\textsf{plus}\ e_0\ e_1) \Downarrow n'}$$

$$\textbf{Div} : \frac{e_0 \Downarrow n_0 \quad e_1 \Downarrow n_1 \quad n' = n_0 / n_1}{(\textsf{div}\ e_0\ e_1) \Downarrow n'}$$

$$\textbf{Not}_\mathbf{0} : \frac{e \Downarrow 0}{(\textsf{not}\ e) \Downarrow 1} \qquad \textbf{Not}_\mathbf{1} : \frac{e \Downarrow n \quad n \neq 0}{(\textsf{not}\ e) \Downarrow 0}$$

$$\textbf{If}_\mathbf{T} : \frac{e_0 \Downarrow n \quad n \neq 0 \quad e_1 \Downarrow n'}{(\textsf{if}\ e_0\ e_1\ e_2) \Downarrow n'} \quad \textbf{If}_\mathbf{F} : \frac{e_0 \Downarrow n \quad n = 0 \quad e_2 \Downarrow n'}{(\textsf{if}\ e_0\ e_1\ e_2) \Downarrow n'}$$

Small-Step Semantics of IfArith

CIS352 — Spring 2021

Kris Micinski

Code in the description!

Last Week: Defined **Big-Step** semantics for IfArith

Last Week: Defined **Big-Step** semantics for IfArith

Two different, but similar, formulations:

- Metacircular Interpreter in Racket
- Natural Deduction

The metacircular interpreter is our "implementation" of natural deduction

```
(define (evaluate e)
  (match e
    [(? integer? n) n]
    [`(plus ,(? expr? e0) ,(? expr? e1))
     (+ (evaluate e0) (evaluate e1))]
    [`(div ,(? expr? e0) ,(? expr? e1))
     (/ (evaluate e0) (evaluate e1))]
    [`(not ,(? expr? e-guard))
    (if (= (evaluate e-guard) 0) 1 0)]
    [`(if ,(? expr? e0) ,(? expr? e1) ,(? expr? e2))
     (if (equal? 0 (evaluate e0)) (evaluate e2) (evaluate e1))]
    [_ "unexpected input"]))
```

$$\textbf{Const} : \frac{c \in \mathbb{Q}}{c \Downarrow c} \qquad \textbf{Plus} : \frac{e_0 \Downarrow n_0 \quad e_1 \Downarrow n_1 \quad n' = n_0 + n_1}{(\text{plus } e_0\ e_1) \Downarrow n'}$$

$$\textbf{Div} : \frac{e_0 \Downarrow n_0 \quad e_1 \Downarrow n_1 \quad n' = n_0 / n_1}{(\text{div } e_0\ e_1) \Downarrow n'}$$

$$\textbf{Not}_\mathbf{0} : \frac{e \Downarrow 0}{(\text{not } e) \Downarrow 1} \qquad \textbf{Not}_\mathbf{1} : \frac{e \Downarrow n \quad n \neq 0}{(\text{not } e) \Downarrow 0}$$

$$\textbf{If}_\mathbf{T} : \frac{e_0 \Downarrow 0 \quad e_1 \Downarrow n'}{(\text{if } e_0\ e_1\ e_2) \Downarrow n'} \qquad \textbf{If}_\mathbf{F} : \frac{e_0 \Downarrow n \quad n = 0 \quad e_2 \Downarrow n'}{(\text{if } e_0\ e_1\ e_2) \Downarrow n'}$$

```
(define (evaluate e)
  (match e
    [(? integer? n) n]
    [`(plus ,(? expr? e0) ,(? expr? e1))
     (+ (evaluate e0) (evaluate e1))]
    [`(div ,(? expr? e0) ,(? expr? e1))
     (/ (evaluate e0) (evaluate e1))]
    [`(not ,(? expr? e-guard))
    (if (= (evaluate e-guard) 0) 1 0)]
    [`(if ,(? expr? e0) ,(? expr? e1) ,(? expr? e2))
     (if (equal? 0 (evaluate e0)) (evaluate e2) (evaluate e1))]
    [_ "unexpected input"]))
```

$$\textbf{Const} : \frac{c \in \mathbb{Q}}{c \Downarrow c} \qquad \textbf{Plus} : \frac{e_0 \Downarrow n_0 \quad e_1 \Downarrow n_1 \quad n' = n_0 + n_1}{(\text{plus } e_0\ e_1) \Downarrow n'}$$

$$\textbf{Div} : \frac{e_0 \Downarrow n_0 \quad e_1 \Downarrow n_1 \quad n' = n_0 / n_1}{(\text{div } e_0\ e_1) \Downarrow n'}$$

$$\textbf{Not}_0 : \frac{e \Downarrow 0}{(\text{not } e) \Downarrow 1} \qquad \textbf{Not}_1 : \frac{e \Downarrow n \quad n \neq 0}{(\text{not } e) \Downarrow 0}$$

$$\textbf{If}_\textbf{T} : \frac{e_0 \Downarrow 0 \quad e_1 \Downarrow n'}{(\text{if } e_0\ e_1\ e_2) \Downarrow n'} \qquad \textbf{If}_\textbf{F} : \frac{e_0 \Downarrow n \quad n = 0 \quad e_2 \Downarrow n'}{(\text{if } e_0\ e_1\ e_2) \Downarrow n'}$$

```
(define (evaluate e)
  (match e
    [(? integer? n) n]
    [`(plus ,(? expr? e0) ,(? expr? e1))
     (+ (evaluate e0) (evaluate e1))]
    [`(div ,(? expr? e0) ,(? expr? e1))
     (/ (evaluate e0) (evaluate e1))]
    [`(not ,(? expr? e-guard))
    (if (= (evaluate e-guard) 0) 1 0)]
    [`(if ,(? expr? e0) ,(? expr? e1) ,(? expr? e2))
     (if (equal? 0 (evaluate e0)) (evaluate e2) (evaluate e1))]
    [_ "unexpected input"]))
```

$$\textbf{Const} : \frac{c \in \mathbb{Q}}{c \Downarrow c} \qquad \textbf{Plus} : \frac{e_0 \Downarrow n_0 \quad e_1 \Downarrow n_1 \quad n' = n_0 + n_1}{(\text{plus } e_0\ e_1) \Downarrow n'}$$

$$\textbf{Div} : \frac{e_0 \Downarrow n_0 \quad e_1 \Downarrow n_1 \quad n' = n_0 / n_1}{(\text{div } e_0\ e_1) \Downarrow n'}$$

$$\textbf{Not}_0 : \frac{e \Downarrow 0}{(\text{not } e) \Downarrow 1} \qquad \textbf{Not}_1 : \frac{e \Downarrow n \quad n \neq 0}{(\text{not } e) \Downarrow 0}$$

$$\textbf{If}_\textbf{T} : \frac{e_0 \Downarrow 0 \quad e_1 \Downarrow n'}{(\text{if } e_0\ e_1\ e_2) \Downarrow n'} \qquad \textbf{If}_\textbf{F} : \frac{e_0 \Downarrow n \quad n = 0 \quad e_2 \Downarrow n'}{(\text{if } e_0\ e_1\ e_2) \Downarrow n'}$$

This week we'll be looking at **small-step** interpreters

Implement and formalize **textual reduction**

Small-step interpreters specify execution as a sequence of **steps**, where each step makes only a small, local computation

```
  (div (plus 2 2) (plus 3 -1))
→ (div 4 (plus 3 -1))
→ (div 4 2)
→ 2
```

We will define the rules precisely in a few slides…

This allows us to reason about, and implement, control over execution in a fine-grained way at each step.

```
  (div (plus 2 2) (plus 3 -1))
→ (div 4 (plus 3 -1))
→ (div 4 2)
→ 2
```

Allows us to reason about traces of the program more easily. Useful for things like…

- Reasoning about finite prefix of infinitely-looping programs (servers)
- Temporal properties of the program (data-race freedom, etc…)

Our job is to define this step function / operator, written mathematically as $e_0 \rightarrow e_1$

```
  (div (plus 2 2) (plus 3 -1))
→ (div 4 (plus 3 -1))
→ (div 4 2)
→ 2
```

First observation: can only take a step when both arguments to plus / div are **values**

```
  (div (plus 2 2) (plus 3 -1))
→ (div 4 (plus 3 -1))
→ (div 4 2)
→ 2
```

We can immediately evaluate `(plus 2 2)` to 4, and then to step the whole expression, we substitute 4 in place of `(plus 2 2)`

```
  (div (plus 2 2) (plus 3 -1))
→ (div 4 (plus 3 -1))
→ (div 4 2)
→ 2
```

We first identify a **redex** (“reducible expression”)

Now two rules (so far)

- Immediately reduce plus/div when args are values
- When e_0 or e_1 is **not** a value, reduce one of them and replace it

```
  (div (plus 2 2) (plus 3 -1))
→ (div 4 (plus 3 -1))
→ (div 4 2)
→ 2
```

- Immediately reduce plus/div when args are values

Let's translate this into the natural deduction style..

By the way, in this lecture we are defining a **new set of rules** for the small-step semantics, which I will call **SmallIfArith**

These rules are separate from the rules for **IfArith**

“Immediately reduce plus/div when args are values”

“Immediately reduce plus/div when args are values”

$$\textbf{StepPlus}\ \frac{n_0 \in \mathbb{Q} \quad n_1 \in \mathbb{Q} \quad n' = n_0 + n_1}{(\mathsf{plus}\ n_0\ n_1) \to n'}$$

"When e_0 or e_1 is **not** a value, reduce one of them and replace it"

$$\textbf{PlusLeft}\ \frac{e_0 \to e'}{(\text{plus } e_0\ e_1) \to (\text{plus } e'\ e_1)}$$

$$\textbf{PlusRight}\ \frac{e_1 \to e'}{(\text{plus } n\ e_1) \to (\text{plus } n\ e')}$$

The n here is a bit crucial: it adds determinism to our semantics!

"When e_0 or e_1 is **not** a value, reduce one of them and replace it"

$$\textbf{StepPlus}\ \frac{n_0 \in \mathbb{Q} \quad n_1 \in \mathbb{Q} \quad n' = n_0 + n_1}{(\text{plus } n_0\ n_1) \to n'}$$

$$\textbf{PlusRight}\ \frac{n \in \mathbb{Q} \quad e_1 \to e'}{(\text{plus } n\ e_1) \to (\text{plus } n\ e')}$$

$$\textbf{PlusLeft}\ \frac{e_0 \to e'}{(\text{plus } e_0\ e_1) \to (\text{plus } e'\ e_1)}$$

"To process (plus e_0 e_1), first check if is a value. If it is, then check if e_1 is a value. If both are, perform the addition."

"When e_0 or e_1 is **not** a value, reduce one of them and replace it"

$$\textbf{StepPlus}\ \frac{n_0 \in \mathbb{Q} \quad n_1 \in \mathbb{Q} \quad n' = n_0 + n_1}{(\text{plus } n_0\ n_1) \to n'}$$

$$\textbf{PlusRight}\ \frac{n \in \mathbb{Q} \quad e_1 \to e'}{(\text{plus } n\ e_1) \to (\text{plus } n\ e')}$$

$$\textbf{PlusLeft}\ \frac{e_0 \to e'}{(\text{plus } e_0\ e_1) \to (\text{plus } e'\ e_1)}$$

These are the three cases you need to consider for +

Very similar operation for division…

$$\textbf{StepDiv}\ \frac{n_0 \in \mathbb{Q} \quad n_1 \in \mathbb{Q} \quad n' = n_0/n_1}{(\mathsf{div}\ n_0\ n_1) \to n'}$$

$$\textbf{DivRight}\ \frac{n \in \mathbb{Q} \quad e_1 \to e'}{(\mathsf{div}\ n\ e_1) \to (\mathsf{div}\ n\ e')}$$

$$\textbf{DivLeft}\ \frac{e_0 \to e'}{(\mathsf{div}\ e_0\ e_1) \to (\mathsf{div}\ e'\ e_1)}$$

$$\textbf{PlusLeft}\ \frac{e_0 \rightarrow e'}{(\text{plus}\ e_0\ e_1) \rightarrow (\text{plus}\ e'\ e_1)}$$

$$\textbf{PlusRight}\ \frac{e_1 \rightarrow e'}{(\text{plus}\ e_0\ e_1) \rightarrow (\text{plus}\ e_0\ e')}$$

What would happen if we did this instead…?

Semantics would be **nondeterministic**

```
((plus 1 2) (plus 2 2)) -> (plus (plus 1 2) 4)
((plus 1 2) (plus 2 2)) -> (plus 3 (plus 2 2))
```

$$\textbf{PlusLeft}\ \frac{e_0 \to e'}{(\text{plus } e_0\ e_1) \to (\text{plus } e'\ e_1)}$$

$$\textbf{PlusRight}\ \frac{e_1 \to e'}{(\text{plus } e_0\ e_1) \to (\text{plus } e_0\ e')}$$

This will manifest by complicating our definition of step

```
(define/contract (step e)
  (expr? -> expr?)
  …)
```

We would need instead…

```
(define/contract (step e)
  (expr? -> (listof expr?))
  …)
```

What about not..?

$$\textbf{StepNot}_0\ \frac{n \neq 0}{(\mathsf{not}\ n) \to 0}$$

$$\textbf{StepNot}_1\ \frac{n = 0}{(\mathsf{not}\ n) \to 1}$$

$$\textbf{StepNot}\ \frac{e \to e'}{(\mathsf{not}\ e) \to (\mathsf{not}\ e')}$$

Finally, if…

$$\textbf{If}_\textbf{T}\ \frac{n \neq 0}{(\mathsf{if}\ n\ e_1\ e_2) \to e_1}$$

$$\textbf{If}_\textbf{F}\ \frac{n = 0}{(\mathsf{if}\ n\ e_1\ e_2) \to e_2}$$

$$\textbf{If}\ \frac{e_0 \to e'}{(\mathsf{if}\ e_0\ e_1\ e_2) \to (\mathsf{if}\ e'\ e_1\ e_2)}$$

So many rules! Rules are overly complicated: next lecture we will refactor them to be more attractive...

$$\textbf{StepPlus}\ \frac{n_0 \in \mathbb{Q} \quad n_1 \in \mathbb{Q} \quad n' = n_0 + n_1}{(\text{plus } n_0\ n_1) \to n'}$$

$$\textbf{StepDiv}\ \frac{n_0 \in \mathbb{Q} \quad n_1 \in \mathbb{Q} \quad n' = n_0 / n_1}{(\text{div } n_0\ n_1) \to n'}$$

$$\textbf{PlusRight}\ \frac{n \in \mathbb{Q} \quad e_1 \to e'}{(\text{plus } n\ e_1) \to (\text{plus } n\ e')}$$

$$\textbf{DivRight}\ \frac{n \in \mathbb{Q} \quad e_1 \to e'}{(\text{div } n\ e_1) \to (\text{div } n\ e')}$$

$$\textbf{PlusLeft}\ \frac{e_0 \to e'}{(\text{plus } e_0\ e_1) \to (\text{plus } e'\ e_1)}$$

$$\textbf{DivLeft}\ \frac{e_0 \to e'}{(\text{div } e_0\ e_1) \to (\text{div } e'\ e_1)}$$

$$\textbf{StepNot}_0\ \frac{n \neq 0}{(\text{not } n) \to 0}$$

$$\textbf{StepNot}_1\ \frac{n = 0}{(\text{not } n) \to 1}$$

$$\textbf{StepNot}\ \frac{e \to e'}{(\text{not } e) \to (\text{not } e')}$$

$$\textbf{If}_\textbf{T}\ \frac{n \neq 0}{(\text{if } n\ e_1\ e_2) \to e_1} \qquad \textbf{If}_\textbf{F}\ \frac{n = 0}{(\text{if } n\ e_1\ e_2) \to e_2}$$

$$\textbf{If}\ \frac{e_0 \to e'}{(\text{if } e_0\ e_1\ e_2) \to (\text{if } e'\ e_1\ e_2)}$$

One very important omission: there is **no defined step** for values!

These rules only tell us how to step expressions. We need to keep doing that (in a loop) until we reach a value.

Now that we have the rules, let's code them up as a small-step interpreter

```
(define/contract (step e)
  (-> (lambda (x) (and (expr? x) (not (value? x)))) expr?)
  'todo)
```

Context and Redex Semantics

CIS352 — Spring 2021

Kris Micinski

Last lecture: **so many rules! How could you ever remember all of these!?**

$$\textbf{StepPlus}\ \frac{n_0 \in \mathbb{Q} \quad n_1 \in \mathbb{Q} \quad n' = n_0 + n_1}{(\text{plus } n_0\ n_1) \to n'}$$

$$\textbf{StepDiv}\ \frac{n_0 \in \mathbb{Q} \quad n_1 \in \mathbb{Q} \quad n' = n_0 / n_1}{(\text{div } n_0\ n_1) \to n'}$$

$$\textbf{PlusRight}\ \frac{n \in \mathbb{Q} \quad e_1 \to e'}{(\text{plus } n\ e_1) \to (\text{plus } n\ e')}$$

$$\textbf{DivRight}\ \frac{n \in \mathbb{Q} \quad e_1 \to e'}{(\text{div } n\ e_1) \to (\text{div } n\ e')}$$

$$\textbf{PlusLeft}\ \frac{e_0 \to e'}{(\text{plus } e_0\ e_1) \to (\text{plus } e'\ e_1)}$$

$$\textbf{DivLeft}\ \frac{e_0 \to e'}{(\text{div } e_0\ e_1) \to (\text{div } e'\ e_1)}$$

$$\textbf{StepNot}_0\ \frac{n \neq 0}{(\text{not } n) \to 0}$$

$$\textbf{If}_\textbf{T}\ \frac{n \neq 0}{(\text{if } n\ e_1\ e_2) \to e_1} \qquad \textbf{If}_\textbf{F}\ \frac{n = 0}{(\text{if } n\ e_1\ e_2) \to e_2}$$

$$\textbf{StepNot}_1\ \frac{n = 0}{(\text{not } n) \to 1}$$

$$\textbf{StepNot}\ \frac{e \to e'}{(\text{not } e) \to (\text{not } e')}$$

$$\textbf{If}\ \frac{e_0 \to e'}{(\text{if } e_0\ e_1\ e_2) \to (\text{if } e'\ e_1\ e_2)}$$

In this case, it was much easier to write the interpreter!

$$\textbf{StepPlus}\ \frac{n_0 \in \mathbb{Q} \quad n_1 \in \mathbb{Q} \quad n' = n_0 + n_1}{(\mathsf{plus}\ n_0\ n_1) \rightarrow n'}$$

$$\textbf{StepDiv}\ \frac{n_0 \in \mathbb{Q} \quad n_1 \in \mathbb{Q} \quad n' = n_0 / n_1}{(\mathsf{div}\ n_0\ n_1) \rightarrow n'}$$

$$\textbf{PlusRight}\ \frac{n \in \mathbb{Q} \quad e_1 \rightarrow e'}{(\mathsf{plus}\ n\ e_1) \rightarrow (\mathsf{plus}\ n\ e')}$$

$$\textbf{DivRight}\ \frac{n \in \mathbb{Q} \quad e_1 \rightarrow e'}{(\mathsf{div}\ n\ e_1) \rightarrow (\mathsf{div}\ n\ e')}$$

$$\textbf{PlusLeft}\ \frac{e_0 \rightarrow e'}{(\mathsf{plus}\ e_0\ e_1) \rightarrow (\mathsf{plus}\ e'\ e_1)}$$

$$\textbf{DivLeft}\ \frac{e_0 \rightarrow e'}{(\mathsf{div}\ e_0\ e_1) \rightarrow (\mathsf{div}\ e'\ e_1)}$$

$$\textbf{StepNot}_0\ \frac{n \neq 0}{(\mathsf{not}\ n) \rightarrow 0}$$

$$\textbf{StepNot}_1\ \frac{n = 0}{(\mathsf{not}\ n) \rightarrow 1}$$

$$\textbf{StepNot}\ \frac{e \rightarrow e'}{(\mathsf{not}\ e) \rightarrow (\mathsf{not}\ e')}$$

$$\textbf{If}_\textbf{T}\ \frac{n \neq 0}{(\mathsf{if}\ n\ e_1\ e_2) \rightarrow e_1}$$

$$\textbf{If}_\textbf{F}\ \frac{n = 0}{(\mathsf{if}\ n\ e_1\ e_2) \rightarrow e_2}$$

$$\textbf{If}\ \frac{e_0 \rightarrow e'}{(\mathsf{if}\ e_0\ e_1\ e_2) \rightarrow (\mathsf{if}\ e'\ e_1\ e_2)}$$

Also, our small-step rules violate a basic principle: We might prefer that each step of the semantics to have a maximum (bounded) runtime.

Our small-step semantics needs to “dig down” arbitrarily far into the term before it makes progress.

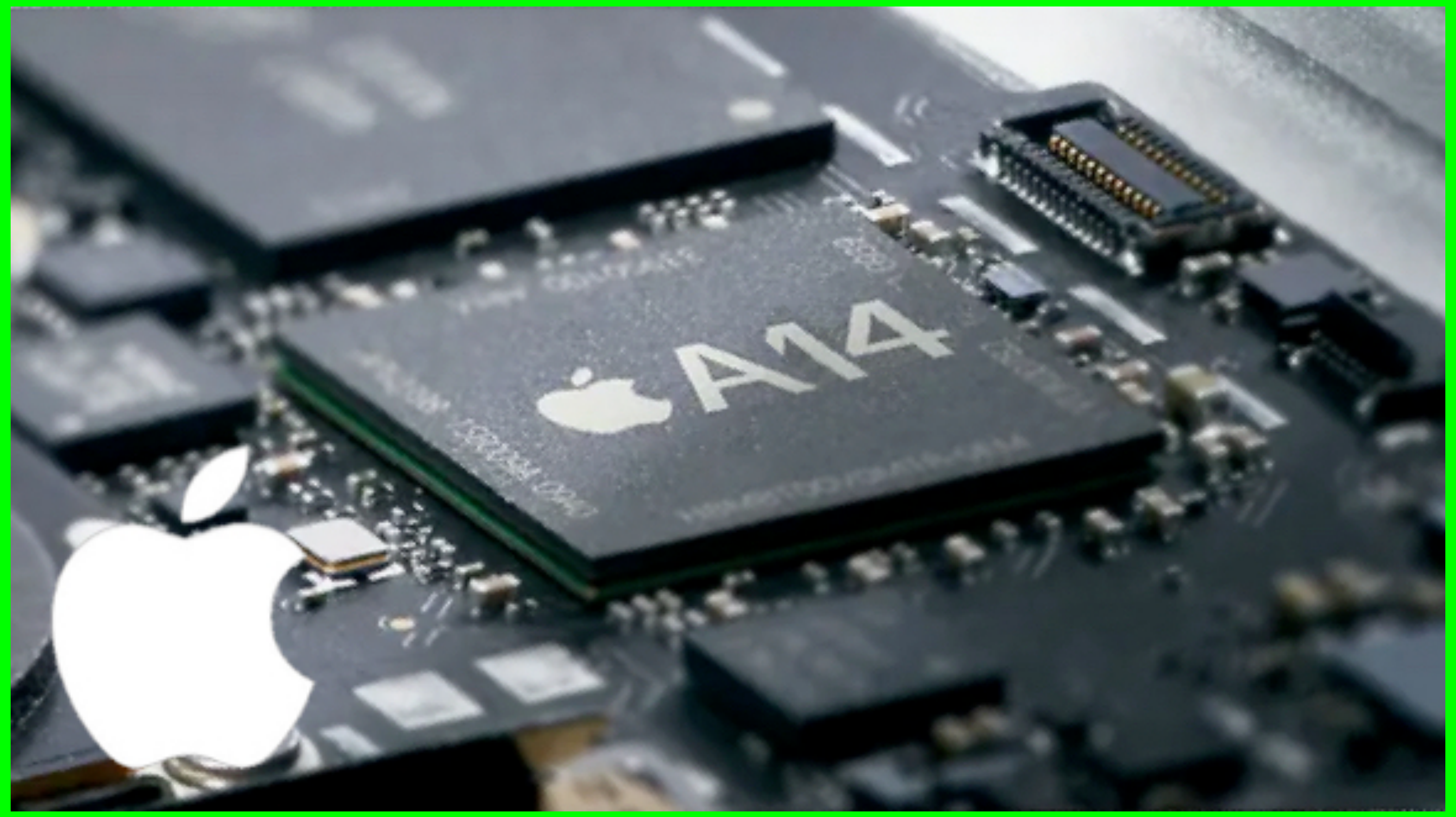
$$\cfrac{\cfrac{\cfrac{1 \in \mathbb{Q}}{(\mathsf{not}\ 1) \to 0}}{(\mathsf{not}\ (\mathsf{not}\ 1)) \to 1}}{\cfrac{\vdots}{(\mathsf{not}\ (\ldots(\mathsf{not}\ (\mathsf{not}\ 1))\ldots) \to 0}}$$

In our last interpreter, the step function is **not** tail-recursive, instead **step** is direct-style recursive and then called in a tail-recursive loop by evaluate!

```
(define (step e)
  (match e
    …
    [`(not 0) 1]
    [`(not ,n) #:when (not (equal? n 0)) 0]
    …))
```

This is not necessarily a problem, but it is often desirable for our step function to be finite. For example, assembly languages **must** operate in finite time because instructions are executed

```
MOV     r0, #10
MOV     r1, #3
ADD     r0, r0, r1
```



Also: our semantics is very **wasteful** with respect to work. Again: for large terms it "digs down" to find the correct redex (reducible expression)…

```
(plus (plus (plus 1 1) 2) 3)
```

Then "rebuilds" the term, only to then "dig down" again during the next step…

Lots of wasted effort digging, rebuilding, and digging again…

In this lecture, we're going to talk about **context and redex** semantics, which is anoptimization of the small-step semantics we saw last lecture.

```
(define (step e)
  (match e
    …
    [`(not 0) 1]
    [`(not ,n) #:when (not (equal? n 0)) 0]
    …))
```

In this lecture, we're going to talk about **context and redex** semantics, which is anoptimization of the small-step semantics we saw last lecture.

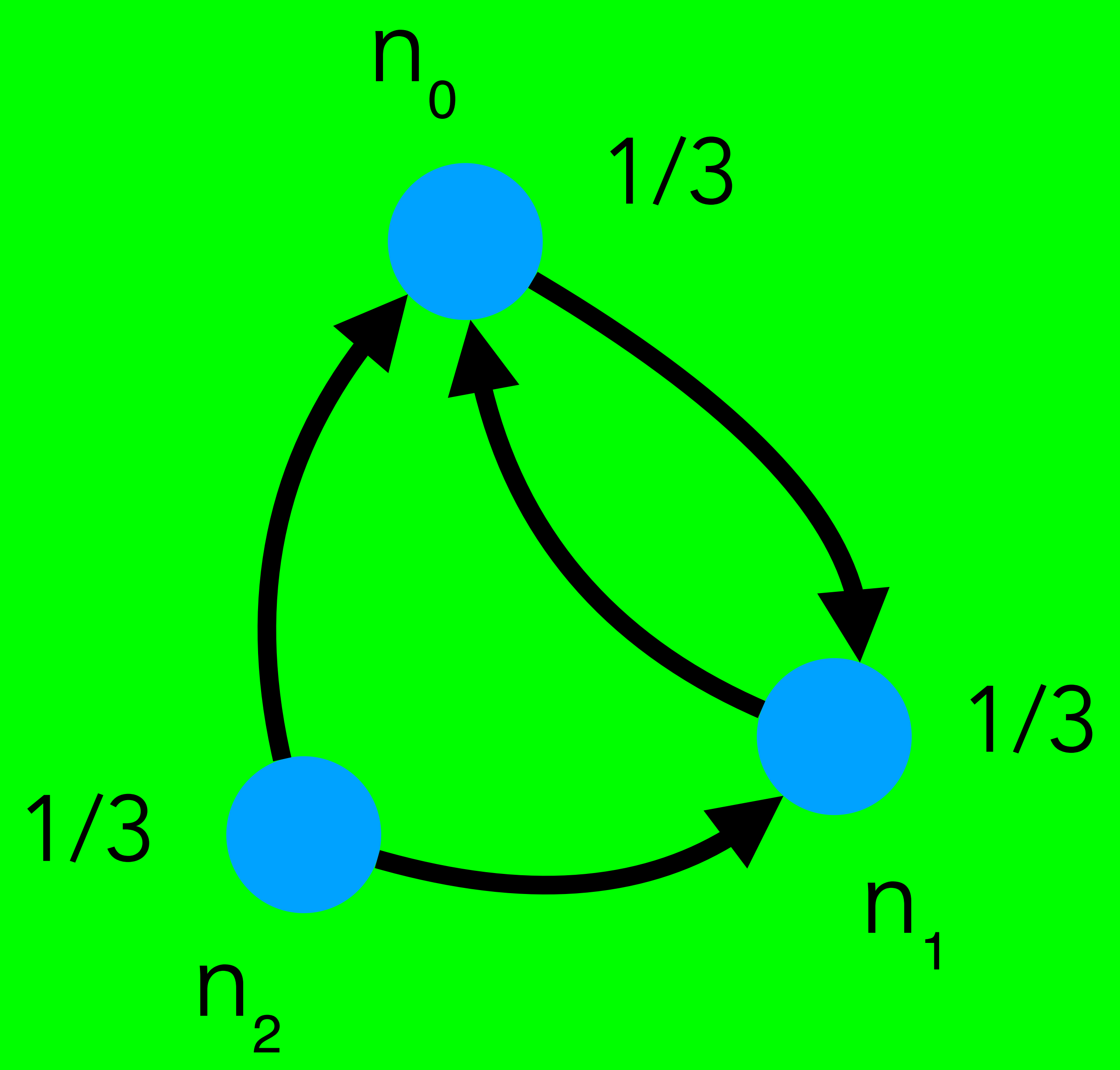
```
(define (step e)
  (match e
    …
    [`(not 0) 1]
    [`(not ,n) #:when (not (equal? n 0)) 0]
    …))
```

P1: PageRank

CIS352 — Spring 2021

Kris Micinski

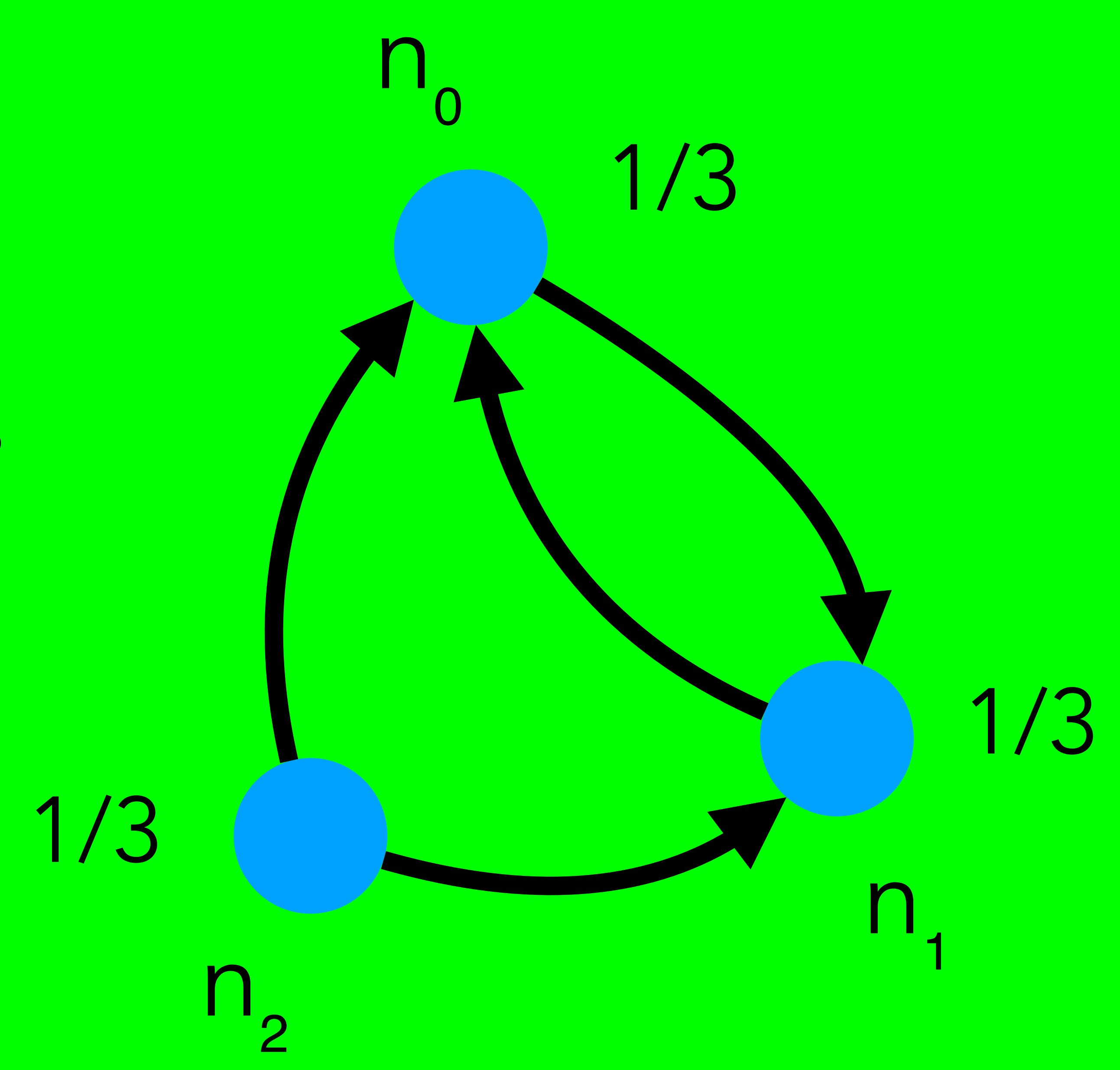
Graphs



- A graph is a pair ⟨N,E⟩ of
  - A set of **nodes**, N
  - A set of **edges**, E, of the form
    - $(n_0,n_1) \mid n_0,n_1 \in N$
- Can equivalent represent in several ways:
  - Adjacency list (list of edges)
  - Graphs can be be composed of either **undirected** or **directed** edges

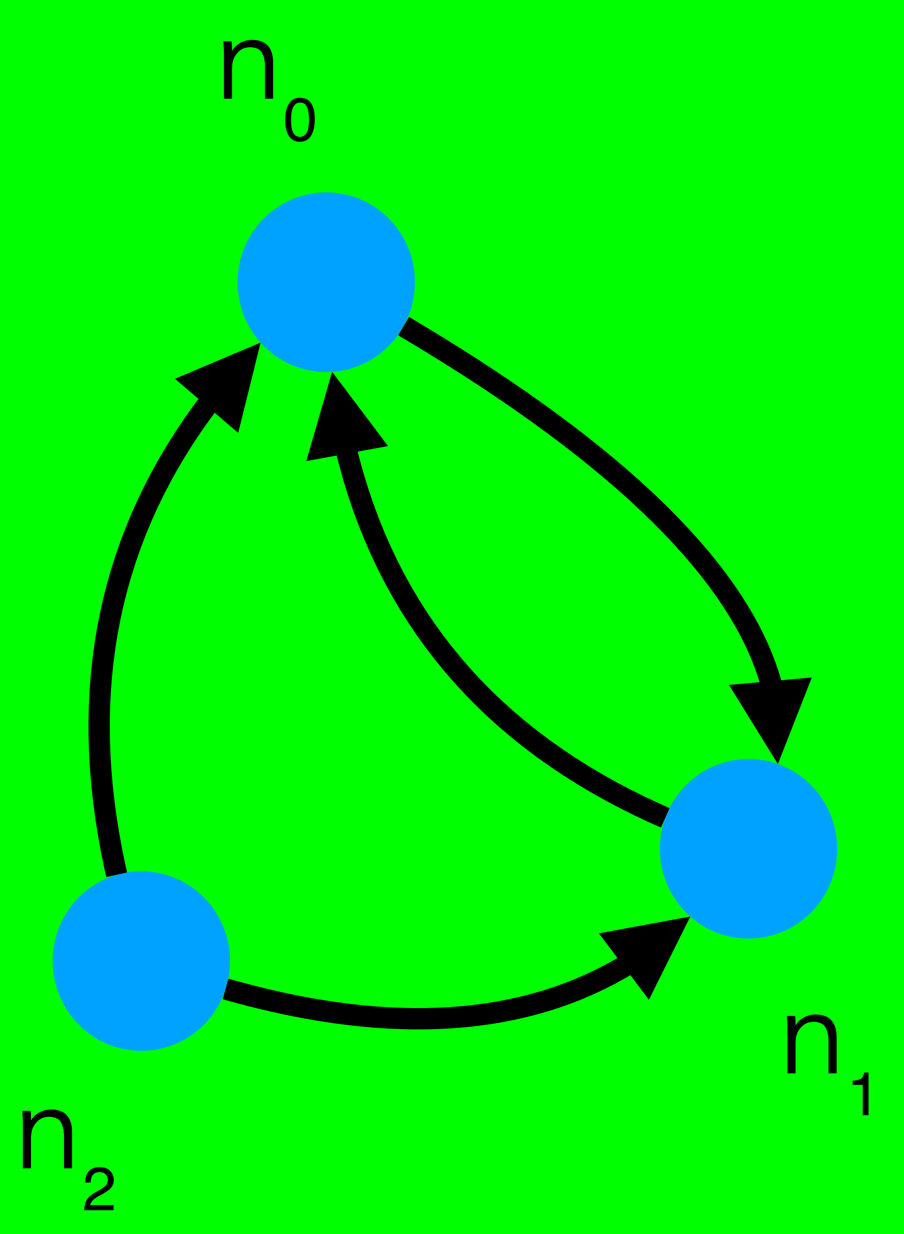
PageRank

- Algorithm that originally powered Google
- Calculates a probability distribution on a graph
  - I.e., assigns a number in [0,1] to each node
  - This number is the page's "rank."
- Forms a **probability distribution**
  - Page ranks sum to 1 across all pages
  - $f \in N \rightarrow [0,1]$
  - $\sum_i f(i) = 1$ over $i \in dom(f)$



- For this assignment we will use **list of edges**
- Can use this to calculate:
  - Neighbors
  - Num of nodes in graph (total)
  - As input to PageRank

```
(define x '((n0 n1)
            (n1 n0)
            (n2 n0)
            (n2 n1)))
```



Write a function that calculates the pages to which a given page links

```
(define x '((n0 n1)
            (n1 n0)
            (n2 n0)
            (n2 n1)))
```

Write a function that calculates the pages to which a given page links

```
(define (links-of graph node)
  (define (loop graph l)
    (match graph
      [`() l]
      [`((,p0 ,p1) . ,rst)
       (if (equal? p0 node)
           (loop rst (cons p1 l))
           (loop rst l))]))
  (loop graph '()))
```

Representing PageRanks

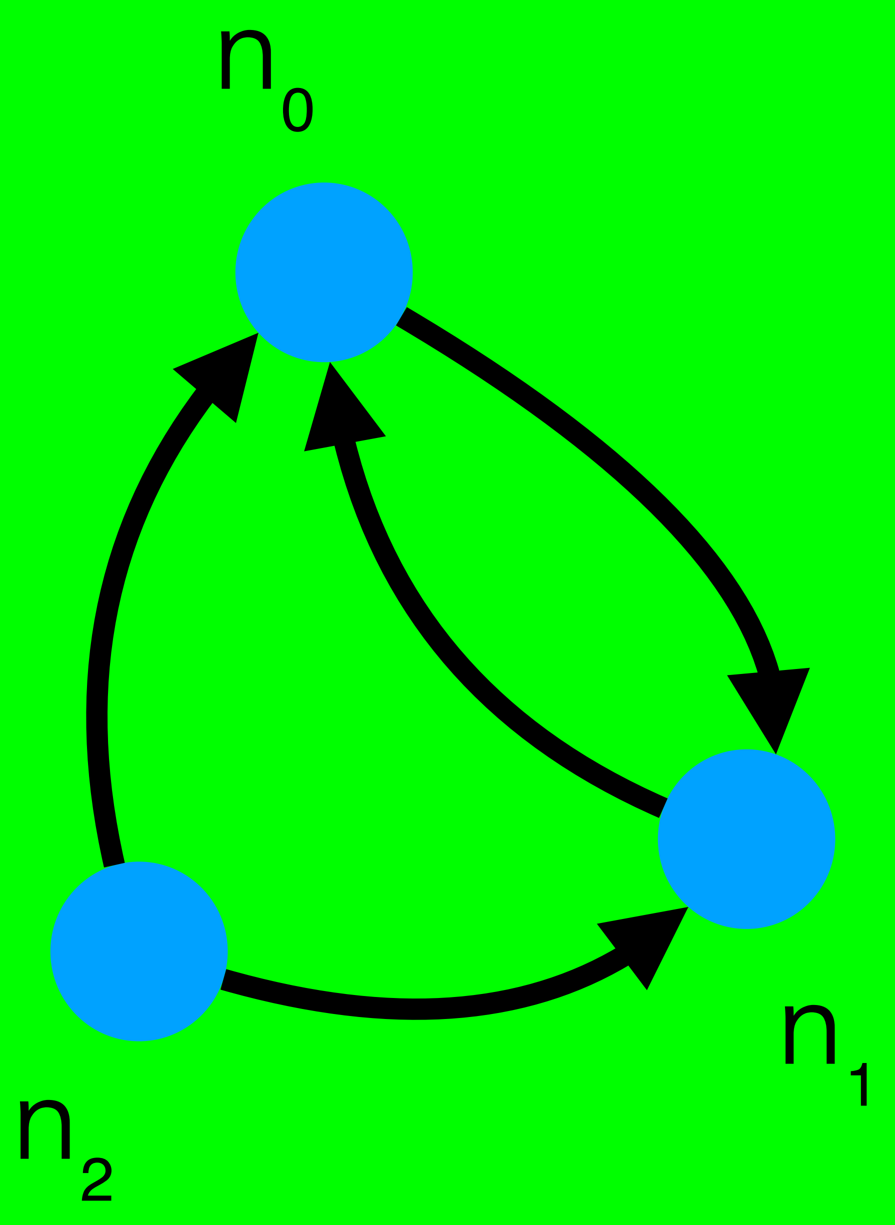
- PageRanks are represented using Racket **hashes**
  - Key/value maps (similar to hash tables)
  - Immutable w/ O(1) runtime for lookup/insert
    - Based on Hash Array-Mapped Tries (HAMT)

- `(hash 'a 0 1 2 "hello" 'c)` — creates hashes, note keys can be heterogeneous type
- `(hash-ref x 'a)` — Looks up value for key `'a`
- `(hash-set x 'a 2)` — Returns a **new** hash with updated key for 'a
- `(hash-keys x)` and `(hash-values x)`— Return list of keys / values (useful for iterating)

PageRank algorithm

- Begins by constructing **initial** PageRank
  - Each page has rank 1/N (for N nodes)
- Then, performs an **iteration** step some number of times
  - You decide how long you want to do this
  - Usually until change is smaller than some delta

```
(hash ‘n0 1/4
      ‘n1 1/4
      ‘n2 1/4
      ‘N3 1/4)
```



PageRank Iteration Step

PageRank is like a **vote**. Each page has a certain share of votes (its PR), each step it votes for each page to which it links, but it divides its vote equally across links.

Intuitively the next PageRank for page i is the sum of:

- A **random chance** that a surfer will jump to i
  - (1-d)/N applies random chance to all pages
- The PageRanks of the pages that link to it, weighted by the number of links those pages have

$$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)}$$

- At each step, the next PR for page i is calculated as:

$$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)}$$

- Where:
  - $M(p_i)$: set of pages that link to i
  - $PR(p_j)$ and $PR(p_i)$: PageRanks of i and j
  - $L(p_j)$ is the number of links from j to any other page
  - d is a “dampening factor” (typically .85)

- At each step, the next PR for page i is calculated as:

$$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)}$$

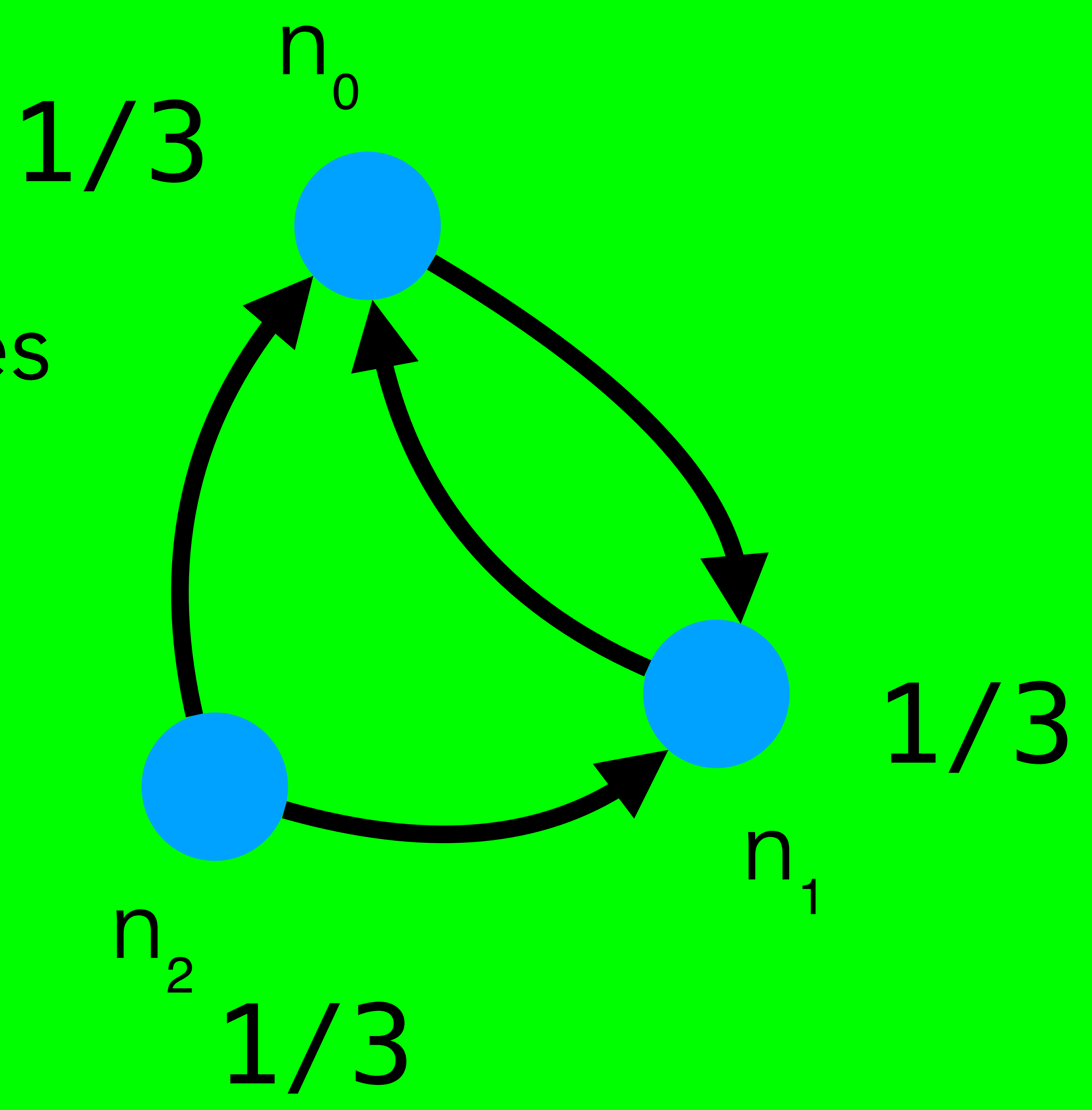
- Where:
  - $M(p_i)$: set of pages that link to i
  - $PR(p_j)$ and $PR(p_i)$: PageRanks of i and j
  - $L(p_j)$ is the number of links from j to any other page
  - d is a “dampening factor” (typically .85)

$$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)}$$

Let's calculate the next values of n_0, n_1, and n_2 (assume d=85/100)

For n_0. Sum of…

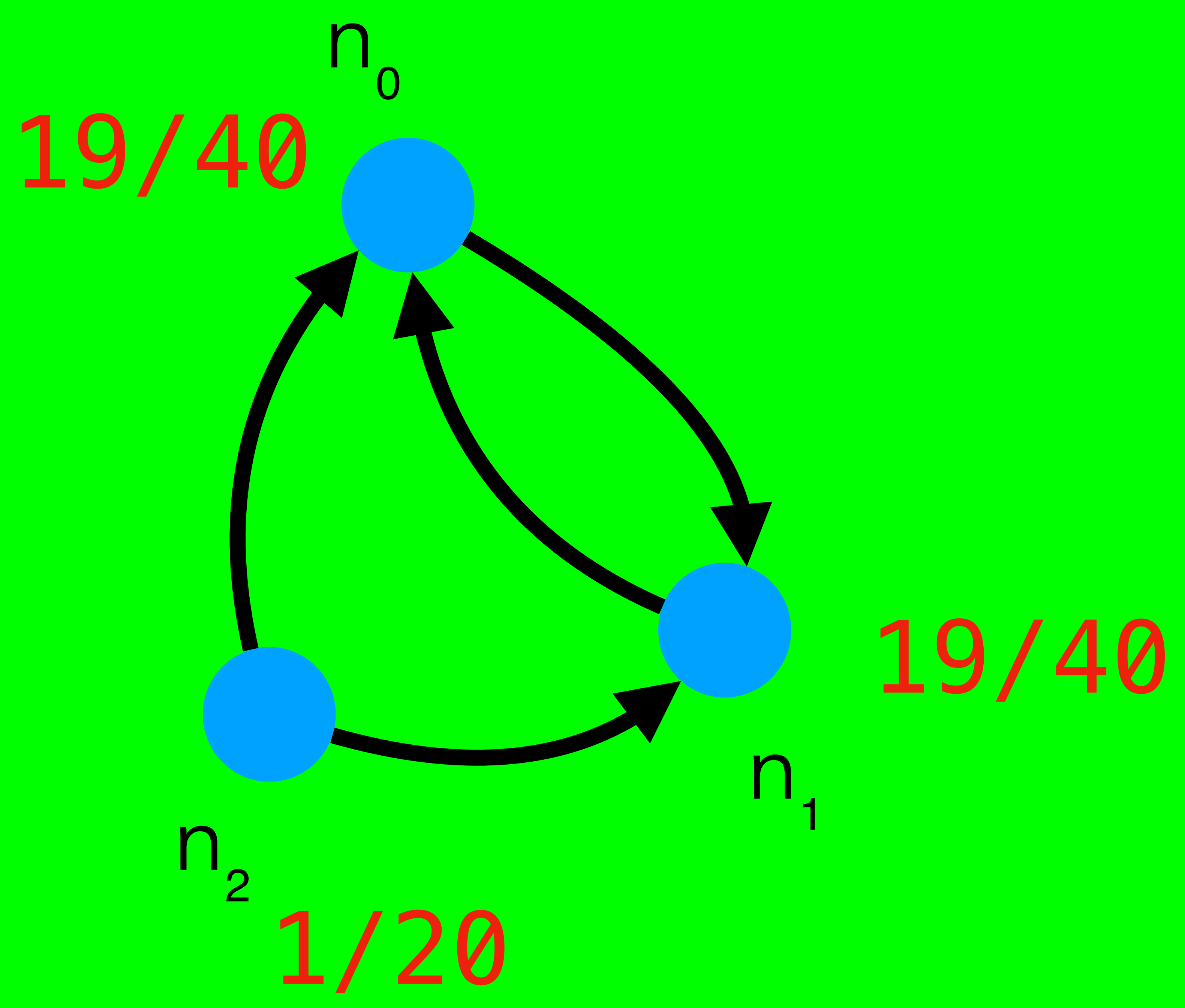
- (1-85/100)/3, since 3 nodes
- For n_2…
  - 85/100 * 1/3 / 2
- For n_1…
  - 85/100 * 1/3 / 1
- = 19/40



$$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)}$$

So next PR should be…

```
(hash ‘n0 19/40
      ‘n1 19/40
      ‘n2 1/20)
```



PageRank Assumptions

- Several simplifying assumptions for input graphs
- No "self-links:" remove links from a page to itself
- All nodes link to at least one other node
  - Can fix this manually: link to **every other node**
- These steps necessary to make math work out (i.e., so that iteration forms a probability distribution)
- **All test input graphs have this form**

Hints

- Read Racket docs for lists, sets, and hashes
- **Start sooner rather than later**
  - Will require much more time than a0
- num-pages, num-links, and num-backlinks are all easier
  - Should be able to mostly do now
- mk-initial-pagerank, step-pagerank and iterate-pagerank-until are a little harder

Lambda Calculus Introduction

CIS352 — Spring 2021

Kris Micinski

The Lambda Calculus (1930s)

- Variables
- Function application
- Lambda abstraction

Just these three elements form a *complete* computational system

Original Syntax

$$
\begin{array}{lll}
e & ::= x & \text{Variables} \\
  & \mid \lambda x\,.\, e & \text{Lambdas} \\
  & \mid e_0\ e_1 & \text{Applications}
\end{array}
$$

Scheme Syntax

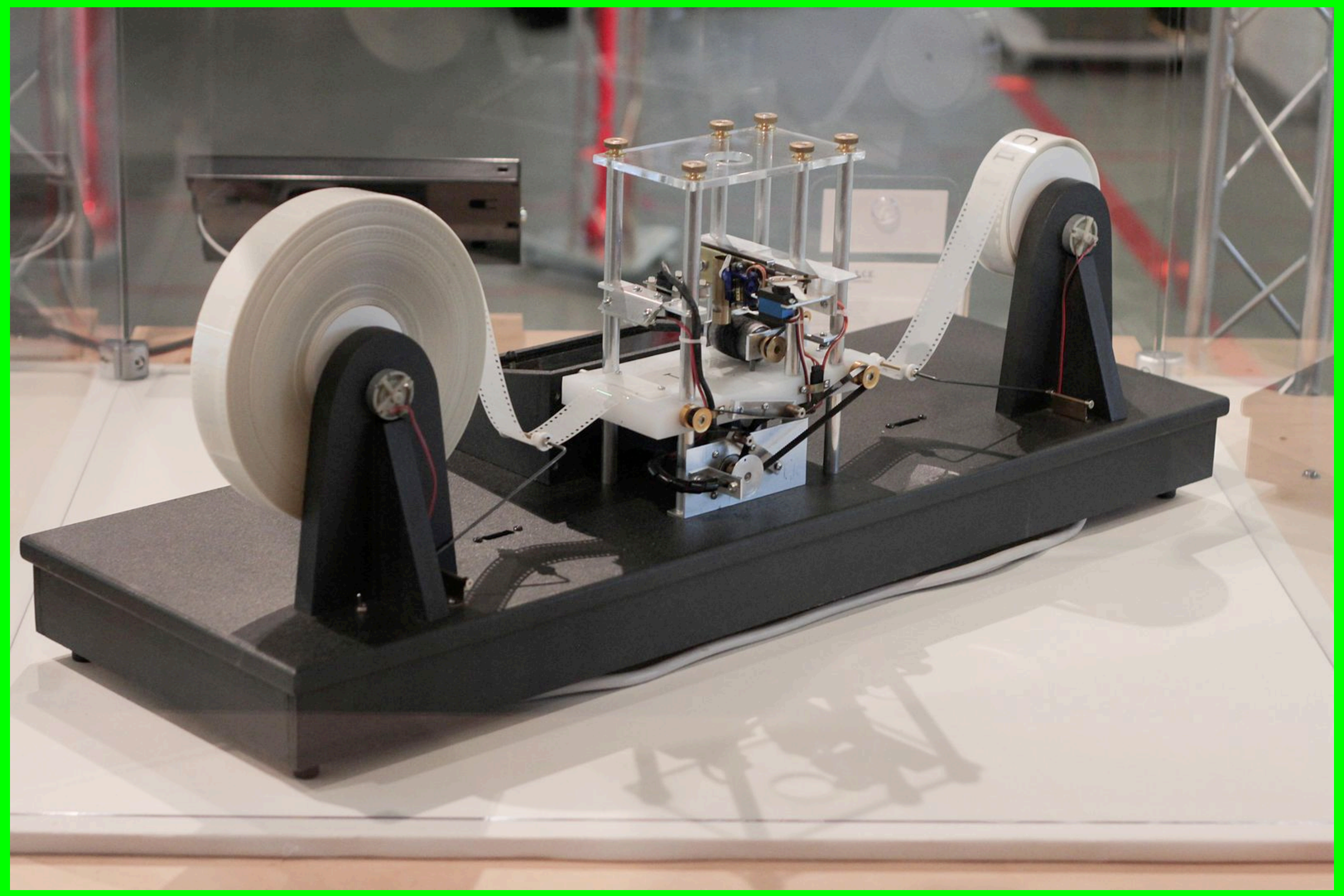
e	$::=$	x	Variables
	$\mid$	$(\lambda\ (x)\ e)$	Lambdas
	$\mid$	$(e_0\ e_1)$	Applications

```
(define (expr? e)
  (match e
    [(? symbol? x) #t]
    [`(lambda (,(? symbol? x)) ,(? expr? e-body)) #t]
    [`(,(? expr? e0) ,(? expr? e1)) #t]
    [_ #f]))
```

Lambda Calculus vs. Turing machines

Lambda Calculus equivalent (in expressivity) to Turing machines.

The **Church-Turing Thesis** states that turing machines / lambda calculus can encode any computable function.



In fact, it is possible to encode (most of) any Scheme program as a lambda calculus expression via a **Church/Boehm encoding**.

Now let’s look at the three lambda calculus forms in detail…

An expression, *abstracted* over all possible values
for a formal parameter, in this case, x.

$$(\lambda\ (x)\ e)$$

Formal parameter

Function body

An expression, *abstracted* over all possible values for a formal parameter, in this case, x.

$$(\lambda\ (x)\ e)$$

Formal parameter

Function body

In fact, you can read lambdas *mathematically* as "**for all**." This observation forms the basis for universal quantification in higher-order logics implemented using typed lambda calculus variants!

Next we have **applications**

$$(e \;\; e)$$

Expression in
function position

Expression in
argument position

Variables are only defined/assigned when a function is applied and its parameter bound to an argument.

$$x$$

Variable reference

How do we compute with the lambda calculus..?

Answer: via **reductions**, which define equivalent / transformed terms.

The **most important** reduction is β, which applies a function by substituting arguments

```
((λ (f) (f (f (λ (x) x)))) (λ (x) x))
```

The **most important** reduction is β, which applies a function by substituting arguments

```
((λ (f) (f (f (λ (x) x)))) (λ (x) x))
```

$\Big\downarrow$ β

```
((λ (x) x) ((λ (x) x) (λ (x) x)))
```

The **most important** reduction is β, which applies a function by substituting arguments

```
((λ (f) (f (f (λ (x) x)))) (λ (x) x))
```

$\downarrow \beta$

```
((λ (x) x) ((λ (x) x) (λ (x) x)))
```

$\downarrow \beta$

```
((λ (x) x) (λ (x) x))
```

The **most important** reduction is β, which applies a function by substituting arguments

```
((λ (f) (f (f (λ (x) x)))) (λ (x) x))
```

$\Big\downarrow \beta$

```
((λ (x) x) ((λ (x) x) (λ (x) x)))
```

$\Big\downarrow \beta$

```
((λ (x) x) (λ (x) x))
```

$\Big\downarrow \beta$

```
(λ (x) x)
```

Textual substitution. This says:
replace every x in E_0 with E_1.

$$\underbrace{\texttt{((}\lambda\ \texttt{(x)}\ E_0\texttt{)}\ E_1\texttt{)}}_{\text{redex}} \quad \rightarrow_\beta \quad E_0[x \leftarrow E_1]$$

(**red**ucible **ex**pression)

Next lecture: carefully defining substitution!

$$\begin{array}{c} ((\lambda\ (x)\ x)\ (\lambda\ (x)\ x)) \\ \Big\downarrow \beta \\ x[x \leftarrow (\lambda\ (x)\ x)] \end{array}$$

$$
\begin{array}{c}
((\lambda\ (x)\ x)\ (\lambda\ (x)\ x)) \\
\big\downarrow{\scriptstyle \beta} \\
(\lambda\ (x)\ x)
\end{array}
$$

Can you beta-reduce the following term more than once…?

```
((λ (x) (x x)) (λ (x) (x x)))
```

```
((λ (x) (x x)) (λ (x) (x x)))
```

β reduction may continue indefinitely (i.e., in non-terminating programs)

$\downarrow \beta$

```
((λ (x) (x x)) (λ (x) (x x)))
```

$\downarrow \beta$

```
((λ (x) (x x)) (λ (x) (x x)))
```

$\downarrow \beta$

```
((λ (x) (x x)) (λ (x) (x x)))
```

$\downarrow \beta$

```
(λ (x) x x) (λ (x) x x)
```

$\downarrow \beta$

```
((λ (x) (x x)) (λ (x) (x x)))
```

$\downarrow \beta$

```
((λ (x) (x x)) (λ (x) (x x)))
```

$\downarrow \beta$

This specific program is known as Ω (Omega)

```
((λ (x) (x x)) (λ (x) (x x)))
```

$\downarrow \beta$

```
((λ (x) (x x)) (λ (x) (x x)))
```

$\downarrow \beta$

```
(λ (x) x x) (λ (x) x x)
```

$\downarrow \beta$

Ω is the smallest non-terminating program!

Note how it reduces to itself in a single step!

$$((\lambda\ (x)\ (x\ x))\ (\lambda\ (x)\ (x\ x)))$$

$$\Big\downarrow \beta$$

$$((\lambda\ (x)\ (x\ x))\ (\lambda\ (x)\ (x\ x)))$$

Lambda Calculus: Reduction / Substitution

CIS352 — Spring 2021

Kris Micinski

Last lecture: β−reduction, informally

$$\underbrace{\texttt{((}\lambda\ \texttt{(x)}\ \texttt{E}_0\texttt{)}\ \texttt{E}_1\texttt{)}}_{\text{redex}} \rightarrow_\beta \texttt{E}_0\texttt{[x} \leftarrow \texttt{E}_1\texttt{]}$$

redex

replace every x in E_0 with E_1.

(**red**ucible **ex**pression)

If you watch the **history of the lambda calculus discussion by Dana Scott**, I will award two participation points (min 5-30):

https://www.youtube.com/watch?v=uS9InrmPIoc

How can we define beta reduction as a
Racket function…?

```
(define (beta-reduce e)
  (match e
    [`((lambda (,x) ,e-body) ,e-arg) (subst x e-arg e-body)]
    [_ (error "beta-reduction cannot apply...")]))
```

Today: how do we define the **subst** function?

Variables are **challenging**

Semantics of the Lambda Calculus

Typical presentations of the lambda calculus define a **textual-reduction semantics**.

You can envision a “machine” where the machine’s **state** is the *text* of the program as it evolves

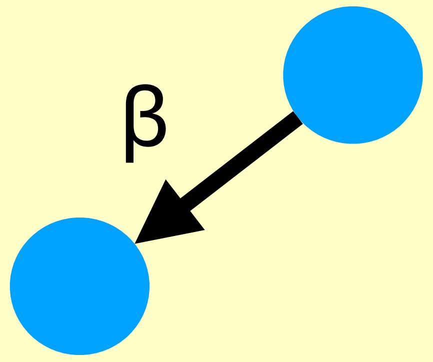
```
((lambda (x) x) ((lambda (z) z) y))
```

Semantics of the Lambda Calculus

Typical presentations of the lambda calculus define a **textual-reduction semantics**.

You can envision a “machine” where the machine’s **state** is the *text* of the program as it evolves

```
((lambda (x) x) ((lambda (z) z) y))
```

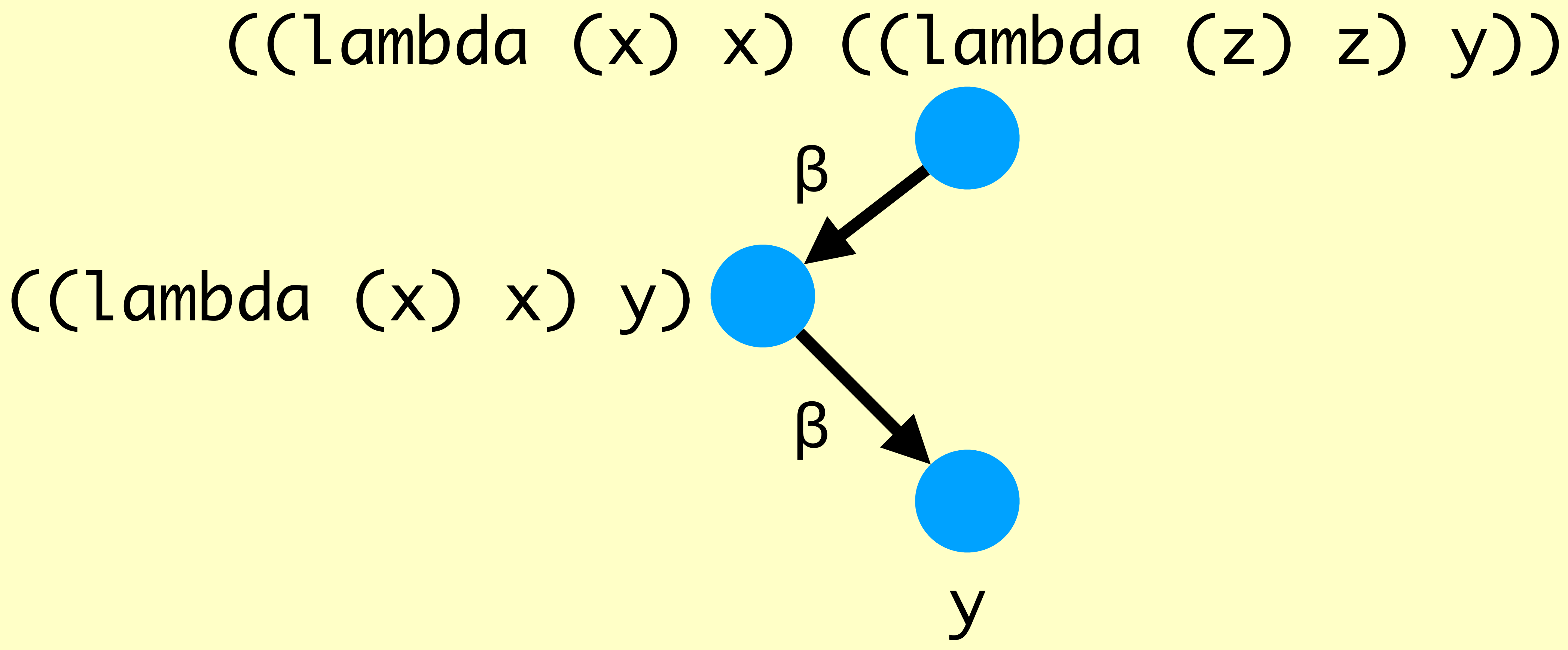



```
((lambda (x) x) y)
```

Semantics of the Lambda Calculus

Typical presentations of the lambda calculus define a **textual-reduction semantics**.

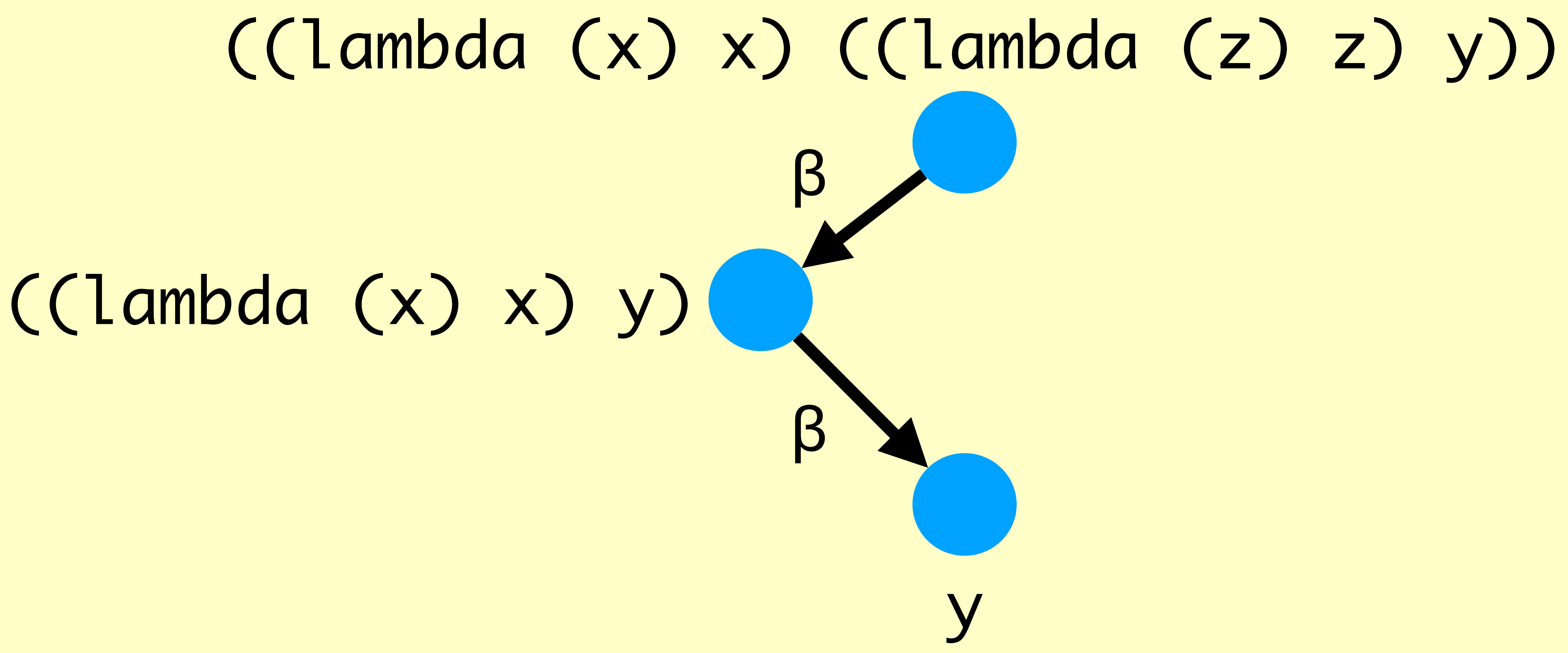
You can envision a “machine” where the machine’s **state** is the *text* of the program as it evolves



Semantics of the Lambda Calculus

Observe! B-Reduction is nondeterministic

In general, a term may have **multiple** β redexes, and thus multiple β reductions

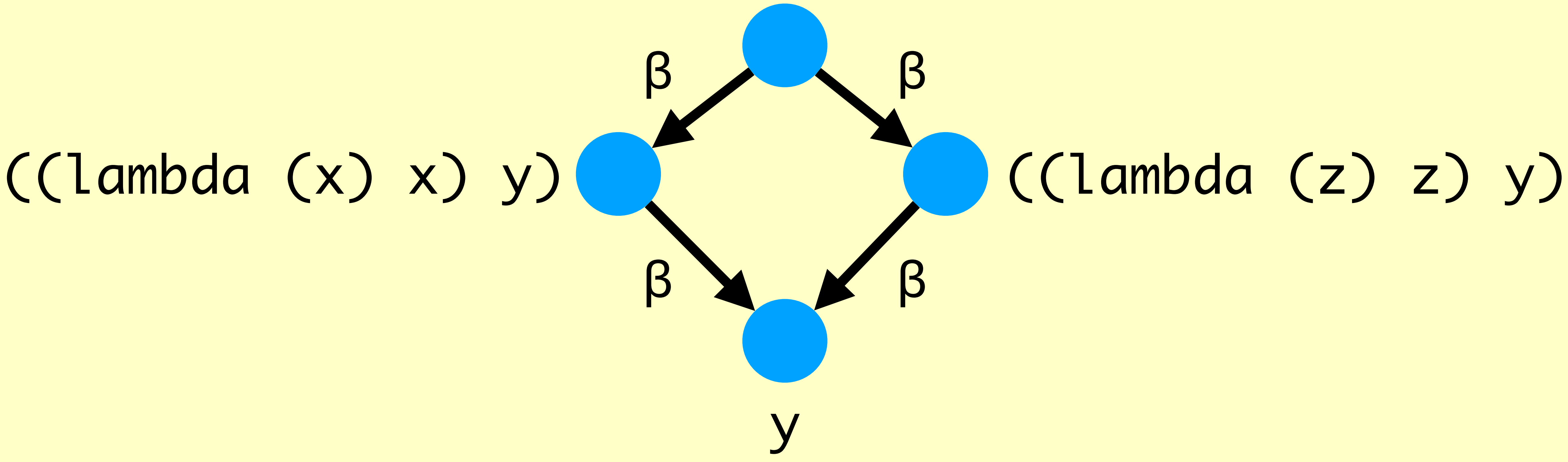


Semantics of the Lambda Calculus

This term has **two** beta redexes!

The outer one in **red**
The inner one in **blue**

`((lambda (x) x) ((lambda (z) z) y))`



The two challenges for this lecture:

- How do we implement substitution
- How do we deal with nondeterminism in the semantics

Substitution seems conceptually simple, but it is surprisingly tricky. But consider this: substitution is fundamentally **where computation happens**!

```
(define (beta-reduce e)
  (match e
    [`((lambda (,x) ,e-body) ,e-arg) (subst x e-arg e-body)]
    [_ (error "beta-reduction cannot apply...")]))
```

If we have **subst**, we can easily define **beta-reduce**.

Free Variables

We define the free variables of a lambda expression via the function FV:

$$\mathbf{FV} : \mathbf{Exp} \to \mathscr{P}(\mathbf{Var})$$

$$\mathbf{FV}(x) \triangleq \{x\}$$

$$\mathbf{FV}((\lambda\ (x)\ e_b)) \triangleq \mathbf{FV}(e_b) \setminus \{x\}$$

$$\mathbf{FV}(e_f\ e_a)) \triangleq \mathbf{FV}(e_f)\ \cup\ \mathbf{FV}(e_a)$$

$$\mathbf{FV}((x\ \ y)) = \{x, y\}$$

$$\mathbf{FV}(((\lambda\ \ (x)\ \ x)\ \ y)) = \{y\}$$

$$\mathbf{FV}(((\lambda\ \ (x)\ \ x)\ \ x)) = \{x\}$$

$$\mathbf{FV}(((\lambda\ \ (y)\ \ ((\lambda\ \ (x)\ \ (z\ \ x))\ \ x))) = \{z, x\}$$

$$\mathbf{FV}((x\ \ y)) = \{x, y\}$$

$$\mathbf{FV}(((\lambda\ \ (x)\ \ x)\ \ y)) = \{y\}$$

$$\mathbf{FV}(((\lambda\ \ (x)\ \ x)\ \ x)) = \{x\}$$

$$\mathbf{FV}(((\lambda\ \ (y)\ \ ((\lambda\ \ (x)\ \ (z\ \ x))\ \ x))) = \{z, x\}$$

What are the free variables of each of the following terms?

```
((λ (x) x) y)

((λ (x) (x x)) (λ (x) (x x)))

((λ (x) (z y)) x)
```

What are the free variables of each of the following terms?

```
((λ (x) x) y)
```

{y}

```
((λ (x) (x x)) (λ (x) (x x)))
```

{}

```
((λ (x) (z y)) x)
```

{x, y, z}

Closed Terms

A term is **closed** when it has no free variables:

- ((lambda (x) x) (lambda (y) y))
- (lambda (z) (lambda (x) (z (lambda (z) z)))

Sometimes we call these (closed terms) **combinators**

Some **open** terms…

- (lambda (x) ((lambda (z) z) **z**))
- ((lambda (x) x) (lambda (z) **x**))

Alpha-Renaming

α-renaming allows us to rename variables:

$$\frac{y \notin FV(e)}{(\lambda\,(x)\;\, e) \overset{\alpha}{\rightarrow} (\lambda\,(y)\;\, e[x \mapsto y])}$$

Still need to define substitution…

Important consequence: terms are unique **up to α equivalence**

$$e_0 \xleftrightarrow{\alpha} e_1 \xleftrightarrow{\alpha} e_2 \xleftrightarrow{\alpha} e_3 \xleftrightarrow{\alpha} e_4 \xleftrightarrow{\alpha} e_5$$

`(lambda (😋) 😋)` `(lambda (x) x)`

Every term has infinitely-many terms to which it is α equivalent

What breaks if the antecedent isn't enforced..?

$$\frac{y \notin FV(e)}{(\lambda\,(x)\;\; e) \xrightarrow{\alpha} (\lambda\,(y)\;\; e[x \mapsto y])}$$

Meaning of term changes! Someone might have an intention to **use** that free variable y

`(lambda (x) y)` very different from `(lambda (x) x)`

Can we define lambda calculi without explicit variables? (**Yes**!)

- De-Bruin Indices (variables are numbers indicating to which binder they belong)
- Combinatory logic uses bases of fully-closed terms. Always possible to rewrite any LC term to use only several closed combinators

We won't study either of these

$$\lambda\ (\lambda\ 1\ (\lambda\ 1))\ (\lambda\ 2\ 1)$$

We define **capture-avoiding substitution**, in which we are careful to avoid places where variables would become **captured** by a substitution.

The problem with (naive) textual substitution

```
((λ (a) (λ (a) a)) (λ (b) b))
```

$$\Big\downarrow \beta$$

```
(λ (a) a)[a ← (λ (b) b)]
```

The problem with (naive) textual substitution

```
((λ (a) (λ (a) a)) (λ (b) b))
```

$\downarrow \beta$

```
(λ (a) (λ (b) b))
```

Capture-avoiding substitution

$E_0[x \leftarrow E_1]$

$$x[x \leftarrow E] = E$$

$$y[x \leftarrow E] = y \quad \text{where} \quad y \neq x$$

$$x[x \leftarrow E] = E$$

$$y[x \leftarrow E] = y \quad \text{where} \quad y \neq x$$

$$(E_0 \;\; E_1)[x \leftarrow E] = (E_0[x \leftarrow E] \;\; E_1[x \leftarrow E])$$

$$x[x \leftarrow E] = E$$

$$y[x \leftarrow E] = y \quad \text{where} \quad y \neq x$$

$$(E_0 \;\; E_1)[x \leftarrow E] = (E_0[x \leftarrow E] \;\; E_1[x \leftarrow E])$$

$$(\lambda \;\; (x) \;\; E_0)[x \leftarrow E] = (\lambda \;\; (x) \;\; E_0)$$

$$x[x \leftarrow E] = E$$

$$y[x \leftarrow E] = y \quad \text{where} \quad y \neq x$$

$$(E_0 \;\; E_1)[x \leftarrow E] = (E_0[x \leftarrow E] \;\; E_1[x \leftarrow E])$$

$$(\lambda \;\; (x) \;\; E_0)[x \leftarrow E] = (\lambda \;\; (x) \;\; E_0)$$

$$(\lambda \;\; (y) \;\; E_0)[x \leftarrow E] = (\lambda \;\; (y) \;\; E_0[x \leftarrow E])$$

$$\text{where} \quad y \neq x \quad \text{and} \quad y \notin FV(E)$$

β-reduction cannot occur when $y \in FV(E)$

How can you beta-reduce the following expression using capture-avoiding substitution?

```
((λ (y)
    ((λ (z) (λ (y) (z y))) y))
 (λ (x) x))
```

How can you beta-reduce the following expression using capture-avoiding substitution?

```
((λ (y)
    ((λ (z) (λ (y) (z y))) y))
 (λ (x) x))
```

$$\Big\downarrow \beta$$

```
((λ (z) (λ (y) (z y))) (λ (x) x))
```

How can you beta-reduce the following expression using capture-avoiding substitution?

```
(λ (y) ((λ (z) (λ (y) z)) (λ (x) y)))
```

How can you beta-reduce the following expression using capture-avoiding substitution?

```
(λ (y) ((λ (z) (λ (y) z)) (λ (x) y)))
```

You **cannot!** This redex would require:

```
(λ (y) z)[z ← (λ (x) y)]
```

(y is free here, so it would be captured)

How can you beta-reduce the following expression using capture-avoiding substitution?

```
    (λ (y) ((λ (z) (λ (y) z)) (λ (x) y)))

→α  (λ (y) ((λ (z) (λ (w) z)) (λ (x) y)))

→β  (λ (y) (λ (w) (λ (x) y)))
```

Instead we alpha-convert first.

To formally define the semantics of the lambda calculus via reduction, we also need rules that will let us apply reductions **inside of** rules:

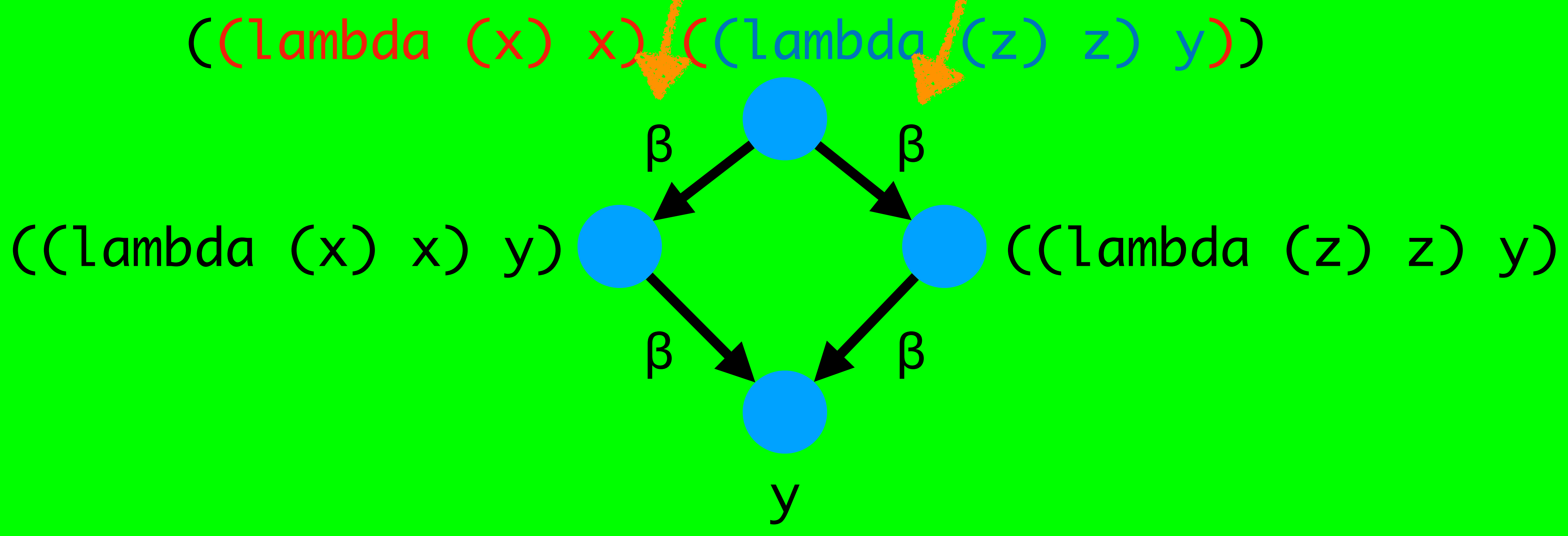
$$\alpha\ \frac{y \notin FV(e)}{(\lambda\,(x)\ e) \overset{\alpha}{\to} (\lambda\,(y)\ e[x \mapsto y])} \quad \beta\ \frac{e' = e_b\,[x \mapsto e_1]}{\big((\lambda\,(x)\ e_b)\ e_1\big) \overset{\beta}{\to} e'}$$

$$\beta_{\mathbf{0}}\ \frac{e_0 \overset{\beta\alpha}{\to} e'}{(e_0\ e_1) \to (e'\ e_1)} \qquad \beta_{\mathbf{1}}\ \frac{e_1 \overset{\beta\alpha}{\to} e'}{(e_0\ e_1) \to (e_0\ e')}$$

$$\alpha\ \frac{y \notin FV(e)}{(\lambda\,(x)\ e) \xrightarrow{\alpha} (\lambda\,(y)\ e[x \mapsto y])} \qquad \beta\ \frac{e' = e_b\,[x \mapsto e_1]}{\big((\lambda\,(x)\ e_b)\ e_1\big) \xrightarrow{\beta} e'}$$

$$\beta_{\mathbf{0}}\ \frac{e_0 \xrightarrow{\beta\alpha} e'}{(e_0\ e_1) \to (e'\ e_1)} \qquad \beta_{\mathbf{1}}\ \frac{e_1 \xrightarrow{\beta\alpha} e'}{(e_0\ e_1) \to (e_0\ e')}$$

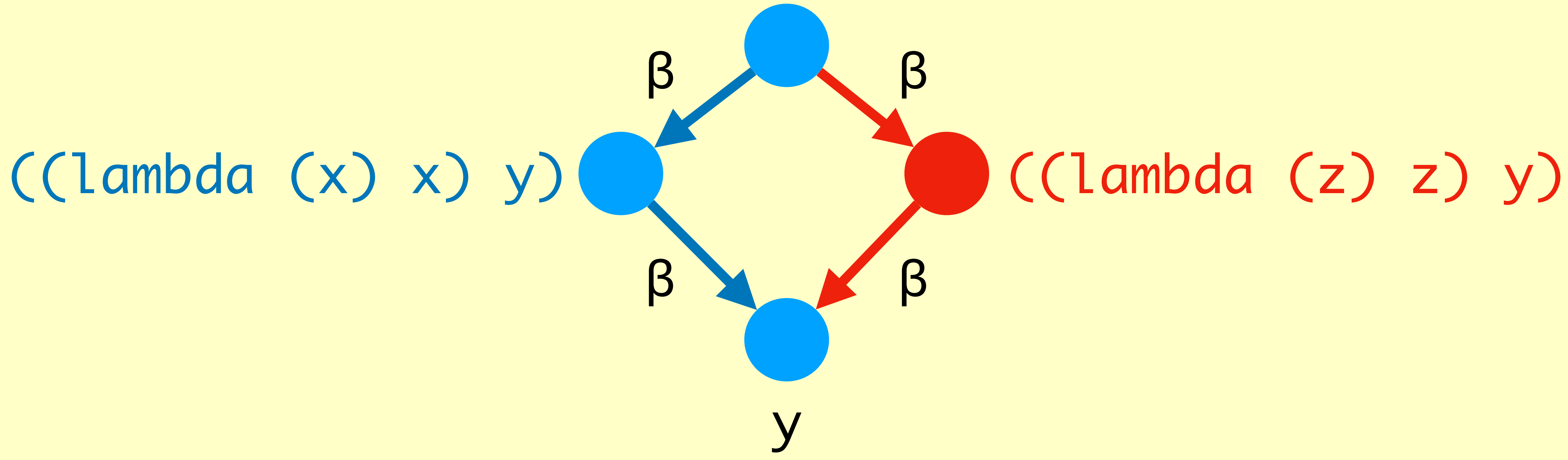
Recall: a term may have **multiple redexes**!



Because β and α reduction are inherently nondeterministic, we use a **reduction strategy**, which is system that tells us which reduction to apply:

- Normal Order — Leftmost (outermost) application
- Applicative Order — Innermost application

`((lambda (x) x) ((lambda (z) z) y))`



We’ll talk more about these **next time**. They relate to the computational notions of **call-by-name (normal)** and **call-by-value (applicative)** η-reduction / expansion capture a property akin to extensionality

$$(\lambda\ (x)\ (E_0\ x)) \rightarrow_\eta E_0 \quad \text{where } x \notin FV(E_0)$$

$$E_0 \rightarrow_\eta (\lambda\ (x)\ (E_0\ x)) \quad \text{where } x \notin FV(E_0)$$

We do not use η-reduction/expansion in computation (unlike β), but it helps us establish certain equalities in lambda theories

When unambiguous, we refer to **reduction** in the lambda calculus as the application of a beta, alpha, or eta reduction:

$$(\rightarrow) \;=\; (\rightarrow_\beta) \cup (\rightarrow_\alpha) \cup (\rightarrow_\eta)$$

$$(\rightarrow^*)$$

(When necessary for exams, we will clarify…)

It is often helpful to think of applying a sequence of reductions to arrive at some final "result."

In the lambda calculus, we call these results / values "normal forms."

A **normal form** is a form that has no more possible applications of some kind of reduction…

$$\begin{array}{c} E_0 \\ {*}\big\downarrow \\ E_8 \\ {*}\big\downarrow \\ ? \end{array}$$

$$E_0 \xrightarrow{*}$$

```
(λ (x) … (λ (z) ((a …) …)))
```

In **beta *normal form***, no function position can be a lambda;
this is to say: *there are no unreduced redexes left*!

We covered a lot of material!

- Free variables
- Alpha renaming
- Beta reduction
- Eta reduction / expansion
- Capture-avoiding substitution
- Applicative / normal order

Next time: **reduction strategies and more normal forms…**

Lambda Calculus Reduction Strategies

CIS352 — Spring 2021

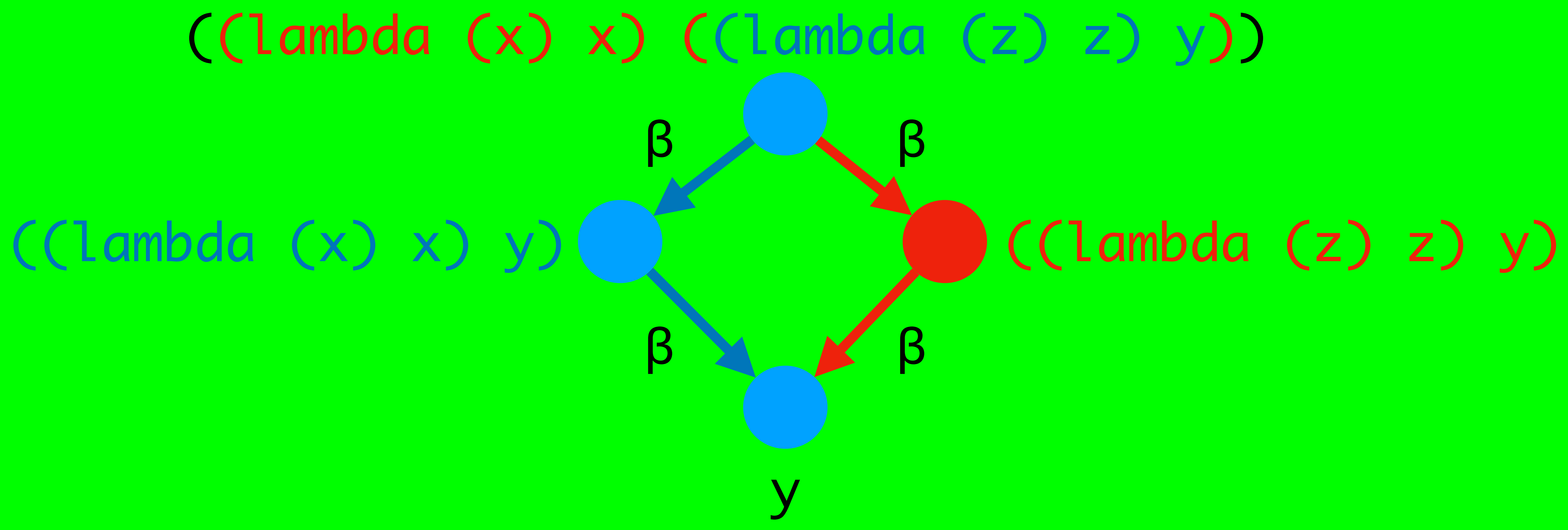
Kris Micinski

Last lecture: reduction **rules** for the lambda calculus

This lecture: reduction **strategies**

As a computer scientist, we can view nondeterminism in the rules as a challenge—it is easier to implement deterministic machines.

As a computer scientist, we can view nondeterminism in the rules as a challenge—it is easier to implement deterministic machines.



We will assume a few basic, but **important**, choices:

- Evaluation of a term will occur **top-down**

We will assume a few basic, but **important**, choices:

- Evaluation of a term will occur **top-down**
- We will never reduce **under a lambda**

We will assume a few basic, but **important**, choices:

- Evaluation of a term will occur **top-down**
- We will never reduce **under a lambda**

```
(lambda (x) ((lambda (y) (y y)) (lambda (y) (y y))))
```

We say that lambda expressions are in **Weak Head Normal Form (WHNF)**

Even though a potential redex exists under the lambda, we will not evaluate it (until application)

Two popular strategies:

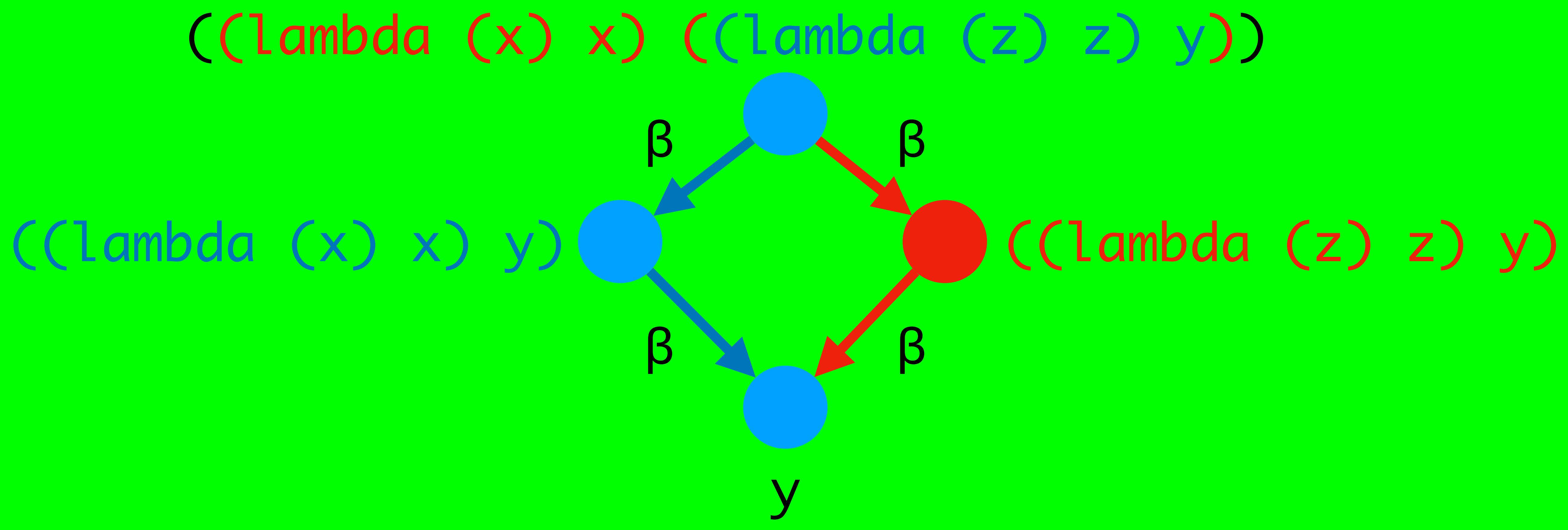
- Call by value, reduce arguments **early** as possible
- Call by name, reduce arguments **late** as possible

Two popular strategies:

- Call by value, reduce arguments **early** as possible
  - Applicative order (innermost), but **not under lambdas**
- Call by name, reduce arguments **late** as possible
  - Normal order, but **not under lambdas**

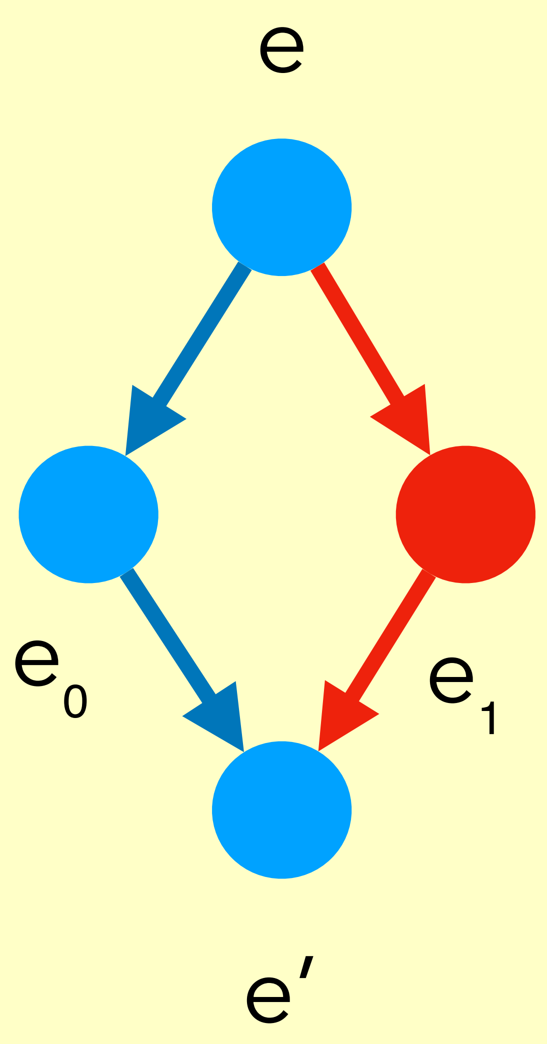
Whenever you get to an application of a lambda,
you have a choice:

- Attempt to evaluate argument?
- Perform application immediately



Church-Rosser Theorem

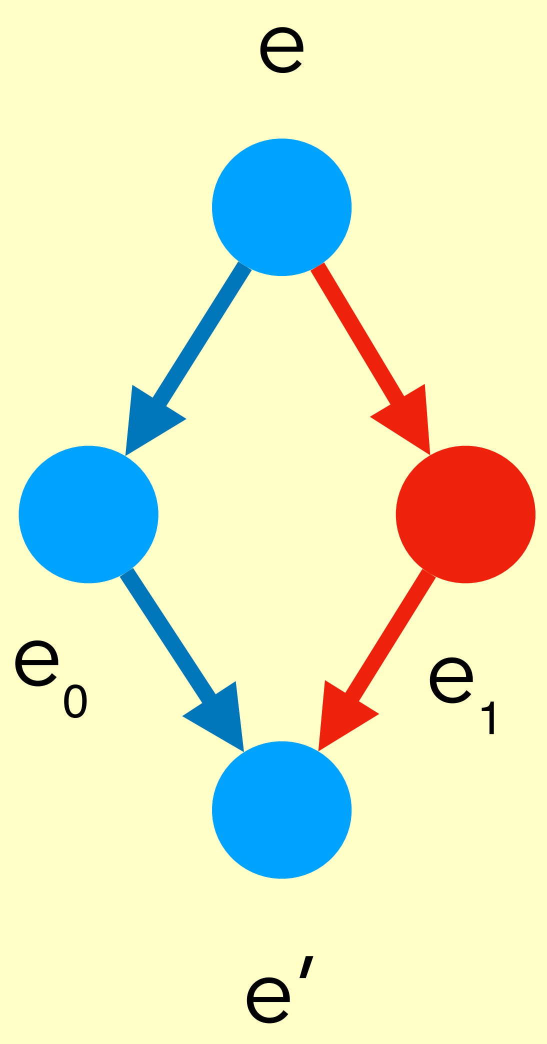
For any expression e,
If e $\to^*$ e_0 **and** e $\to^*$ e_1
Then, both e_0 and e_1 step to
some **common** term e′



Church-Rosser Theorem

For any expression e,
If e $\to^*$ e_0 **and** e $\to^*$ e_1
Then, both e_0 and e_1 step to some **common** term e′

Corollary: all terminating paths result in same normal form!



Give the **reduction sequences** using…

- Call-by-Name
- Call-by-Value

```
((lambda (x) x) ((lambda (y) y) (lambda (y) y)))
```

Give the **reduction sequences** using…

- Call-by-Name
- Call-by-Value

```
((lambda (x) x) ((lambda (y) y) (lambda (y) y)))
```

CBN

```
((lambda (y) y) (lambda (y) y))
```

```
(lambda (y) y)
```

CBV

```
((lambda (x) x) (lambda (y) y))
```

```
(lambda (y) y)
```

Up to alpha equivalence, evaluate this term using:

- Call-by-Name
- Call-by-Value

```
((lambda (x) (lambda (y) y))
 ((lambda (x) (x x)) (lambda (x) (x x))))
```

Up to alpha equivalence, evaluate this term using:

- Call-by-Name
- Call-by-Value

```
((lambda (x) (lambda (y) y))
 ((lambda (x) (x x)) (lambda (x) (x x))))
```

$\downarrow$

```
(lambda (y) y)
```

CBN

Up to alpha equivalence, evaluate this term using:

- Call-by-Name
- Call-by-Value

```
((lambda (x) (lambda (y) y))
 ((lambda (x) (x x)) (lambda (x) (x x))))
```

CBN

```
(lambda (y) y)
```

CBV

```
((lambda (x) (lambda (y) y))
 ((lambda (x) (x x)) (lambda (x) (x x))))
```

```
((lambda (x) (lambda (y) y))
 ((lambda (x) (x x)) (lambda (x) (x x))))
```

```
((lambda (x) (lambda (y) y))
 ((lambda (x) (x x)) (lambda (x) (x x))))
```

Standardization theorem

If an expression can be evaluated to WHNF (i.e., it doesn’t loop), then it has a normal-order reduction sequence.

In other words: the lazy semantics is most permissive, in terms of termination.

Church Numerals

CIS352 — Spring 2021

Kris Micinski

This week in class we're going to talk about **Church Encoding**, a technique to express arbitrary Racket code using **only** the lambda calculus.

We will (by hand) compile Racket forms to just LC

Why do this? Answer: illustrate theoretical expressivity of LC

Our goal this lecture: translate simple arithmetic operations over constants to the lambda calculus

2 + 1 * 2 = 4

We want to express **this** with the lambda calculus

I think this is one of the trickiest things to understand in the course. I first learned this by working out the beta-reductions on paper, and I recommend that approach.

One key problem: how do we represent numbers as lambdas?

Observation 1

On simplifying assumption: focus only on the naturals

Can write any natural number n as:

$$\underbrace{1 + \ldots + 0}_{\text{n times}}$$

$$0 = 0$$

$$1 = 1 + 0$$

$$2 = 1 + 1 + 0$$

$$3 = 1 + 1 + 1 + 0$$

Observation 2: represent the number **n** as a **function** that accepts **another** function g and returns a function that performs g n times.

$$0 = \big(\lambda\,(f)\;(\lambda\,(x)\;x)\big)$$

$$1 = \Big(\lambda\,(f)\;\big(\lambda\,(x)\;(f\;x)\big)\Big)$$

$$2 = \Big(\lambda\,(f)\;\big(\lambda\,(x)\;(f\;(f\;x))\big)\Big)$$

$$\ldots$$

This is where it starts getting confusing, if you are lost here, stop to think this through for a few minutes…

Observation 2: represent the number **n** as a **function** that accepts **another** function g and returns a function that performs g n times.

```
(define zero (lambda (f) (lambda (x) x)))
(define one  (lambda (f) (lambda (x) (f x))))
(define two  (lambda (f) (lambda (x) (f (f x)))))
```

By the way, how do we translate a Church-encoded number to a **Racket** number?

```
;; do add1 n times, starting from 0
;; (add1 (add1 … (add1 0) …))
(define (church->nat n)
  ((n add1) 0)
```

Observation 3: when we use this encoding, any two expressions that are alpha-equivalent to n **is** n

```
(((lambda (y) (y y)) (lambda (x) x))
 (lambda (z) (lambda (x) (z (z x)))))
```

Observation 3: when we use this encoding, any two expressions that are alpha-equivalent to n **is** n

```
(((lambda (y) (y y)) (lambda (x) x))
 (lambda (z) (lambda (x) (z (z x)))))
```

```
(((lambda (x) x) (lambda (x) x))
 (lambda (z) (lambda (x) (z (z x)))))
```

Observation 3: when we use this encoding, any two expressions that are alpha-equivalent to n **is** n

```
(((lambda (y) (y y)) (lambda (x) x))
 (lambda (z) (lambda (x) (z (z x)))))
```

```
(((lambda (x) x) (lambda (x) x))
 (lambda (z) (lambda (x) (z (z x)))))
```

```
((lambda (x) x)
 (lambda (z) (lambda (x) (z (z x)))))
```

Observation 3: when we use this encoding, any two expressions that are alpha-equivalent to n **is** n

```
(((lambda (y) (y y)) (lambda (x) x))
 (lambda (z) (lambda (x) (z (z x)))))
```

```
(((lambda (x) x) (lambda (x) x))
 (lambda (z) (lambda (x) (z (z x)))))
```

```
((lambda (x) x)
 (lambda (z) (lambda (x) (z (z x)))))
```

```
(lambda (z) (lambda (x) (z (z x)))) ;; 2
```

Question:

Say I give you a number n. You know its normal-form (when it is fully-reduced) must be **something** like

```
n = (lambda (f) (lambda (x) (f (f (f … (f x) …)))))
```

How can you generate n + 1?

Question:

Say I give you a number n. You know its normal-form (when it is fully-reduced) must be **something** like

```
n = (lambda (f) (f (f … (f x) …)))
```

How can you generate n + 1?

```
n+1 = (lambda (f) (f (f (f … (f x) …))))
```

Question:

Say I give you a number n. You know its normal-form (when it is fully-reduced) must be **something** like

```
n = (lambda (f) (f (f … (f x) …)))
```

Now, how could I wrote a function, **succ**, which computes n+1 using **only the lambda calculus**?

Question:

Say I give you a number n. You know its normal-form (when it is fully-reduced) must be **something** like

```
n = (lambda (f) (f (f … (f x) …)))
```

Now, how could I wrote a function, **succ**, which computes n+1 using **only the lambda calculus**?

```
;; the *argument*
(lambda (n)
  ;; the thing we're *returning* should do f "n+1 times"
  ;; ((n f) x) "applies f n times" and returns a result
  ;;
  (lambda (f) (lambda (x) (f ((n f) x)))))
```

```
(define succ
 (lambda (n) (lambda (f) (lambda (x) (f ((n f) x))))))
```

```
;; (succ 1) should equal 2
((lambda (n)
   (lambda (f) (lambda (x) (f ((n f) x))))))
 (lambda (f) (lambda (x) (f x))))
```

```
;; (succ 1) should equal 2
(lambda (f)
  (lambda (x) (f (((lambda (f) (lambda (x) (f x))) f) x)))))))
```

```
;; note here: we're reducing under lambda!
(lambda (f)
  (lambda (x) (f ((lambda (x) (f x)) x)))))))
```

```
(lambda (f)
  (lambda (x) (f (f x))))))) ;; this is 2!
```

Question:

Now how do you do addition…? Observation: need **two** arguments. We will use a trick named **currying**.

```
plus = (lambda (n) (lambda (k) …))
one =  (lambda (f) (lambda (x) (f x))
```

We can call this like:

```
((plus one) one) ;; compute 2
```

Question:

Now how do you do addition…? Observation: need **two** arguments. We will use a trick named **currying**.

```
plus = (lambda (n) (lambda (k) …))
one =  (lambda (f) (lambda (x) (f x))
```

We can call this like:

```
((plus one) one)
```

Observe the key idea: plus returns a function that **takes another function** (the second `one`) to complete the work!

```
((n f) x) ;; applies f to x n times
((k f) x) ;; applies f to x k times
```

```
plus =
(lambda (n) (lambda (k)
  (lambda (f) (lambda (x) ((k f) ((n f) x))))))
```

```
((n f) x) ;; applies f to x n times
((k f) x) ;; applies f to x k times
```

```
plus =
(lambda (n) (lambda (k)
  (lambda (f) (lambda (x) ((k f) ((n f) x))))))
```

Homework:

Reduce (to beta-normal-form, i.e., doing all possible reductions) the following (encoding plus, 0, 1, and 2 correctly):

```
(plus 0 1) ;; (lambda (f) (lambda (x) (f x))
(plus 1 1) ;; (lambda (f) (lambda (x) (f (f x)))
(plus 2 0) ;; (lambda (f) (lambda (x) (f (f x)))
```

Alright, now how do you do multiplication..?
Well, do "n **k times**!"

```
((n f) x) ;; applies f to x n times
((k f) x) ;; applies f to x k times
```

```
(lambda (n)
    (lambda (k)
      (lambda (f) (lambda (x) (((n k) f) x)))))
```

```
((n f) x) ;; applies f to x n times
((k f) x) ;; applies f to x k times
```

```
(lambda (n)
    (lambda (k)
      (lambda (f) (lambda (x) (((n k) f) x)))))
```

Homework:

Reduce (to beta-normal-form, i.e., doing all possible reductions) the following (encoding plus, 0, 1, and 2 correctly):

```
(mult 1 1) ;; (lambda (f) (lambda (x) (f x))
(mult 2 1) ;; (lambda (f) (lambda (x) (f (f x)))
(mult 2 0) ;; (lambda (f) (lambda (x) x))
```

P2: Church Encoding

CIS352 — Spring 2021

Kris Micinski

Last lecture: Church numerals and operations over arithmetic.

After last lecture, you should be able to use Church encoding to express things like this:

$$2 + 3 * (4 + 1)$$

In this project, we’ll translate Scheme programs to the lambda calculus.

This project: how do we translate the **rest** of Scheme?

```
e ::= (letrec ([x (lambda (x …) e)]))
    | (let ([x e] …) e)
    | (lambda (x …) e)
    | (e e …)
    | x
    | (if e e e)
    | (prim e e) | (prim e)
    | d
d ::= ℕ | #t | #f | ‘()
x ::= <vars>
prim ::= + | - | * | not | cons | …
```

(Language used in project p2)

Output language

```
e ::= (lambda (x) e)
    | (e e)
    | x

x ::= <vars>
```

Let’s go through the forms one by one and eliminate them :-)

Currying is a trick where you translate multi-arg lambdas into **sequences** of lambdas

```
(λ (x y z) e)  ⟶  (λ (x)
                     (λ (y)
                       (λ (z) e)))

(λ (x) e)      ⟶  (λ (x) e)

(λ () e)       ⟶  (λ (_) e)
```

Of course, you **also** need to fix up **callsites**

```
(f a b c d)   ⟶   ((((f a) b) c) d)
(f a)         ⟶   (f a)
(f)           ⟶   (f (λ (x) x))
```

Alright, so we started with this…

```
e ::= (letrec ([x (lambda (x …) e)]))
    | (let ([x e] …) e)
    | (lambda (x …) e)
    | (e e …)
    | x
    | (if e e e)
    | (prim e e) | (prim e)
    | d
d ::= ℕ | #t | #f | ‘()
x ::= <vars>
prim ::= + | - | * | not | cons | …
```

Now we have…

```
e ::= (letrec ([x (lambda (x …) e)]))
    ;; let is encoded…
    | (lambda (x) e) ;; single x
    | (e e)          ;; single arg
    | x
    | (if e e e)
    | ((prim e) e) | (prim e)
    | d
d ::= ℕ | #t | #f | ‘()
x ::= <vars>
prim ::= + | - | * | not | cons | …
```

Now let's encode if

$\texttt{(if \#t } e_T\ e_F\texttt{)} \longrightarrow e_T$

$\texttt{(if \#f } e_T\ e_F\texttt{)} \longrightarrow e_F$

We need an encoding that does this…

Let's say we encode true as `(λ (t f) t)`

(if #t e_T e_F)   (if #f e_T e_F)

((λ (t f) t) v_T v_F)   ((λ (t f) f) v_T v_F)

v_T v_F

This is critically broken!

Because if we did that, then the encoding of

```
(if #t 0 Ω) ;; Ω = ((lambda (x) (x x)) (lambda (x) (x x)))
```

```
((λ (t f) t) 0 Ω)
```

$\downarrow$

....

Not right! We want it to be just 0!

Note: already explained how to encode 0-arg lambda…

```
((λ (t f) (t)) (λ () eT) (λ () Ω))
```

$\downarrow$

```
((λ () eT))
```

$\downarrow$

e_T

$\downarrow$

v_T

So our true encoding for if/true/false is…

Note: already explained how to encode 0-arg lambda…

$$((\lambda\ (t\ f)\ (t))\ (\lambda\ ()\ e_T)\ (\lambda\ ()\ \Omega))$$

$$\downarrow$$

$$((\lambda\ ()\ e_T))$$

$$\downarrow$$

$$e_T$$

$$\downarrow$$

$$v_T$$

Now we're just down to…

```
e ::= (letrec ([x (lambda (x) e)]))
    | (lambda (x) e)
    | (e e)
    | x
    | ((+ e) e) | ((* e) e)
    | ((cons e) e) | (car e)
    | (cdr e) | (null? e)
    | d
d ::= ℕ | '()
x ::= <vars>
```

We taught you how to do these in the last video!

```
e ::= (letrec ([x (lambda (x) e)]))
    | (lambda (x) e)
    | (e e)
    | x
    | ((+ e) e) | ((* e) e)
    | ((cons e) e) | (car e)
    | (cdr e)
    | d
d ::= ℕ | ‘()
x ::= <vars>
```

So now all we need to do is this…

```
e ::= (letrec ([x (lambda (x) e)]))
    | (lambda (x) e)
    | (e e)
    | x
    | ((+ e) e) | ((* e) e)
    | ((cons e) e) | (car e)
    | (cdr e) | (null? e)
    | d
d ::= ℕ | ‘()
x ::= <vars>
```

```
‘() = (λ (when-cons) (λ (when-null)
        (when-null)))

(cons a b) = (λ (when-cons) (λ (when-null)
                (when-cons a b)))
```

Using this definition, can you define car, cdr, and null?

```
church:null? = (λ (lst)
                 (lst (λ (a b) #f) ;; when cons
                    (λ () #t)))    ;; when null
```

Now all we have is…

```
e ::= (letrec ([x (lambda (x) e)]))
    | (lambda (x) e)
    | (e e)
    | x
x ::= <vars>
```

To implement letrec, we use a **fixed-point combinator** (such as the Y combinator…). This is a bit tricky, so we'll explain it next week in class.

Fixed Points

CIS352 — Spring 2021

Kris Micinski

Last lecture: encoding Scheme in the lambda calculus

```
e ::= (letrec ([x (lambda (x …) e)]))
    | (let ([x e] …) e)
    | (lambda (x …) e)
    | (e e …)
    | x
    | (if e e e)
    | (prim e e) | (prim e)
    | d
d ::= ℕ | #t | #f | ‘()
x ::= <vars>
prim ::= + | - | * | not | cons | …
```

Right now: clone the corresponding autograder exercise for this lecture so you can get participation points…

Last lecture: encoding Scheme in the lambda calculus

But didn't do letrec

```
e ::= (letrec ([x (lambda (x …) e)]))
    | (let ([x e] …) e)
    | (lambda (x …) e)
    | (e e …)
    | x
    | (if e e e)
    | (prim e e) | (prim e)
    | d
d ::= ℕ | #t | #f | ‘()
x ::= <vars>
prim ::= + | - | * | not | cons | …
```

`letrec` lets us define recursive loops

```
(letrec ([f (lambda (x)
              (if (= x 0)
                  1
                  (* x (f (sub1 x)))))])
  (f 20))
```

`letrec` lets us define recursive loops

```
(letrec ([f (lambda (x)
              (if (= x 0)
                  1
                  (* x (f (sub1 x))))])
  (f 20))
```

Unlike **let**, letrec allows referring to f **within** its definition

Unlike **let**, letrec allows referring to f **within** its definition

```
(define (fib-using-letrec x)
  (letrec ([fib (lambda (x)
                  ;; Your answer:
                  'todo)])
    (fib x)))
```

Today, we will discuss a magic term, **Y**, that allows us to write…

```
(letrec ([f (lambda (x)
              (if (= x 0)
                  1
                  (* x (f (sub1 x)))))])
  (f 20))
```

```
(let ([f
       (Y (lambda (f)
            (lambda (x)
              (if (= x 0)
                  1
                  (* x (f (- x 1)))))))])
  (f 20))
```

This magic term, named Y, allows us to construct recursive functions.

```
(define Y (λ (g) ((λ (f) (g (λ (x) ((f f) x))))
                  (λ (f) (g (λ (x) ((f f) x)))))))
```

First, the U combinator

```
(define U (lambda (x) (x x)))
```

The U combinator lets us do something very crucial: pass a copy of a function to itself.

Let’s say I didn’t have letrec, what could I do…?

First observation: pass f to **itself**

```
(let ([f (lambda (mk-f)
           (lambda (x)
             (if (= x 0)
                 1
                 (* x ((mk-f mk-f) x)))))])
  ((f f) 20))
```

`mk-f` is pronounced “make f”

```
(let ([f (lambda (mk-f)
           (lambda (x)
             (if (= x 0)
                 1
                 (* x ((mk-f mk-f) (sub1 x))))))])
  ((f f) 20))
```

Let’s see why this works!

```
(let ([f (lambda (mk-f)
           (lambda (x)
             (if (= x 0)
                 1
                 (* x ((mk-f mk-f) (sub1 x))))))])
  ((f f) 20))
```

Let's see why this works!

1: First, apply f to itself. First lambda goes away, returns (lambda (x) …) with mk-f bound to mk-f

This initial call "makes the next copy"

```
(let ([f (lambda (mk-f)
           (lambda (x) ;; x = 20
             (if (= x 0)
                 1
                 (* x ((mk-f mk-f) (sub1 x)))))))])
  ((f f) 20))
```

Let's see why this works!

1: First, apply f to itself. First lambda goes away, returns (lambda (x) …) with mk-f bound to mk-f

2: Second, apply that (lambda (x) …) to 20, take false branch

```
(let ([f (lambda (mk-f)
           (lambda (x)
             (if (= x 0)
                 1
                 (* x ((mk-f mk-f) (sub1 x))))))])
  ((f f) 20))
```

Let’s see why this works!

1: First, apply f to itself. First lambda goes away, returns (lambda (x) …) with mk-f bound to mk-f

2: Next, apply that (lambda (x) …) to 20, take false branch

3: Next, compute (mk-f mk-f), which **gives us another copy of (lambda (x) …)**

```
(let ([f (lambda (mk-f)
           (lambda (x)
             (if (= x 0)
                 1
                 (* x ((mk-f mk-f) (sub1 x))))))])
  ((f f) 20))
```

Let’s see why this works!

1: First, apply f to itself. First lambda goes away, returns (lambda (x) …) with mk-f bound to mk-f

2: Next, apply that (lambda (x) …) to 20, take false branch

3: Next, compute (mk-f mk-f), which **gives us another copy of (lambda (x) …)**

4: Apply that same function again (until base case)!

The U combinator recipe for recursion…

```
(letrec ([f (lambda (x) e-body)])
  letrec-body)
```

Systematically translate any letrec by:

- Wrapping `(lambda (x) e-body)` in `(lambda (f) …)`
- Changing occurrences of `f` (in e-body) to `(f f)`
- Apply U combinator / apply function to itself
- Changing `letrec` to `let`

Think carefully why this works..!

The U combinator recipe for recursion…

```
(letrec ([f (lambda (x) e-body)])
  letrec-body)
```

Systematically translate any letrec by:

- Wrapping `(lambda (x) e-body)` in `(lambda (f) …)`
- Changing occurrences of `f` (in e-body) to `(f f)`
- Apply U combinator / apply function to itself
- Changing `letrec` to `let`

```
(let ([f (U (lambda (f)
              ;; replace f w/ (f f)
              (lambda (x) e-body))])
  letrec-body)
```

Let's do an example…

```
(define (length-using-letrec lst)
  (letrec ([len (lambda (x)
                  (if (null? x)
                      0
                      (add1 (len (rest x)))))])
    (len lst)))
```

Your job…

```
(define (length-using-u lst)
  (let ([len (U (lambda (f)
                 (lambda (x)
                   'todo)))])
    (len lst)))
```

Now another example…

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

Translate **this** one to use U

```
(define (fib-using-U n)
  (letrec ([fib (U 'todo)])
    (fib n)))
```

```
(let ([f (lambda (mk-f)
           (lambda (x)
             (if (= x 0)
                 1
                 (* x ((mk-f mk-f) (sub1 x))))))])
  ((U f) 20))
```

One pesky thing: need to rewrite function so that calls to mk-f need to first “get another copy” by doing (mk-f mk-f)

By contrast, the **Y** combinator will allow us to write **this**

```
(let ([f (lambda (f)
           (lambda (x)
             (if (= x 0)
                 1
                 (* x (f (sub1 x))))))])
  ((Y f) 20))
```

```
(let ([f (Y (lambda (f)
              ;; no change to e-body
              (lambda (x) e-body))])
  letrec-body)
```

Let's ask ourselves: what does f need to **be** when Y plugs it in…?

$$(Y\ f) = f\ (Y\ f)$$

Deriving Y

```
(Y f) = (f (Y f))
```

```
Y = (λ (f) (f (Y f)))
```

1. Treat as definition

```
mY = (λ (mY)
       (λ (f)
         (f ((mY mY) f))))
```

2. Lift to mY, use self-application

```
mY = (λ (mY)
       (λ (f)
         (f (λ (x) (((mY mY) f) x)))))
```

3. Eta-expand

U-combinator: `(U U)` **is Omega**

```
Y  =  (U (λ (y) (λ (f)
          (f (λ (x) (((y y) f) x))))))
```

```
mY = (λ (mY)
       (λ (f)
         (f (λ (x) (((mY mY) f) x)))))
```

$$(Y\,f) = f\,(Y\,f)$$

By contrast, the **Y** combinator will allow us to write **this**

```
(let ([f (lambda (f)
           (lambda (x)
             (if (= x 0)
                 1
                 (* x (f (sub1 x))))))])
  ((Y f) 20))
```

Closing words of advice:

- Understand how to write recursive functions w/ U / Y
- Do not need to remember precisely why Y works
  - But do need to remember how to use it!
- If you want to understand: just think carefully about what U / Y are doing (with examples)

Continuations

CIS352 — Spring 2021

Kris Micinski

Often speak of evaluating programs in a sequence of steps:

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

E.g., textual reduction. We defined textual reduction for IfArith and for lambda calculus (beta, …)

Textual Reduction Review

Key idea: at each step, we just decided which expression to reduce (using reduction strategy)

```
((lambda (x) ((lambda (y) x) z))
 (lambda (z) (lambda (…) …))
```

In a real implementation, this would be slow (would have to traverse term at each step)

Another way to conceptualize this would be to think of an **explicit stack**

The rule here is: once we “finish” the current expression, we “fill in” the stack

```
(+ (* 2 1) 3)    stack = □ (empty stack)
```

Another way to conceptualize this would be to think of an **explicit stack**

The rule here is: once we "finish" the current expression, we "fill in" the stack

```
   (+ (* 2 1) 3)    stack = □ (empty stack)
-> (* 2 1)          stack = (+ □ 3)
```

Another way to conceptualize this would be to think of an **explicit stack**

The rule here is: once we “finish” the current expression, we “fill in” the stack

```
   (+ (* 2 1) 3)    stack = □ (empty stack)
-> (* 2 1)          stack = (+ □ 3)
-> 2                stack = (+ □ 3)
```

Another way to conceptualize this would be to think of an **explicit stack**

The rule here is: once we “finish” the current expression, we “fill in” the stack

```
   (+ (* 2 1) 3)    stack = □ (empty stack)
-> (* 2 1)          stack = (+ □ 3)
-> 2                stack = (+ □ 3)
-> 3                stack = (+ 2 □)
```

Another way to conceptualize this would be to think of an **explicit stack**

The rule here is: once we “finish” the current expression, we “fill in” the stack

```
   (+ (* 2 1) 3)    stack = □ (empty stack)
-> (* 2 1)          stack = (+ □ 3)
-> 2                stack = (+ □ 3)
-> 3                stack = (+ 2 □)
-> (+ 2 3)          stack = □
```

Another way to conceptualize this would be to think of an **explicit stack**

The rule here is: once we “finish” the current expression, we “fill in” the stack

```
   (+ (* 2 1) 3)    stack = □ (empty stack)
-> (* 2 1)          stack = (+ □ 3)
-> 2                stack = (+ □ 3)
-> 3                stack = (+ 2 □)
-> (+ 2 3)          stack = □
-> 5                stack = □ (done!)
```

These stacks have another appeal: the fact that they make only local changes makes them fast (compared to identifying redex each time).

However, we won’t focus a lot on the efficiencies of this style. If you want to see that, consider taking the compilers course here at SU.

Instead, we will observe that this style offers an additional flexibility: we can always conceptualize the return point as a function!

We call this function the “continuation,” since it lets us “continue” the computation.

```
   (+ (* 2 1) 3) ;; (lambda (rtn) rtn)
-> (* 2 1)       ;; (lambda (x) (+ x 3))
-> 2             ;; (lambda (x) (+ x 3))
-> 3             ;; (lambda (x) (+ 2 x))
-> (+ 2 3)       ;; (lambda (x) x)
-> 5             ;; (lambda (x) x)
```

If you're used to programming in Java/C++, you can think of a continuation as a "callback we invoke to return from a function."

```
   (+ (* 2 1) 3) ;; (lambda (x) x)
-> (* 2 1)       ;; (lambda (x) (+ x 3))
-> 2             ;; (lambda (x) (+ x 3))
-> 3             ;; (lambda (x) (+ 2 x))
-> (+ 2 3)       ;; (lambda (x) x)
-> 5             ;; (lambda (x) x)
```

The call/cc form allows us to **bind** this continuation to a **function**

```
(+ 4 (call/cc (lambda (k) (k 3))))
```

When control reaches call/cc, the program binds the **current continuation** to k

In this case, the current continuation is…

```
(+ 4 (call/cc (lambda (k) (k 3))))
;; (lambda (x) (+ 4 x))
```

How could we write the continuation at the **underlined point**?

```
(let* ([x (+ (* 2 3) 4)]
       [y (add1 x)])
  y)

(lambda (z)
 (let* ([x (+ z 4)] [y (add1 x])) y))
```

How could we write the continuation at the **underlined point**?

```
(let* ([x (+ (* 2 3) 4)]
       [y (add1 x)])
  y)
```

```
(lambda (result)
  (let* ([x (+ result 4)]
         [y (add1 x)])
  y)
```

DANGER

Continuations are normal functions in most ways. One crucial difference: when you invoke a continuation, it **abandons** the current stack and **reinstates** the continuation!

Again: invoking a continuation is **different** than invoking a *normal* (non-continuation) function.

Students **frequently** find this confusing!

When execution reaches **this point**, k is bound as the continuation

```
(+ 4 (call/cc (lambda (k) (k 3))))
```

Then, when we **invoke** the continuation, we **abandon** the *current* continuation and **reinstate** the *saved* continuation

```
(+ 4 (call/cc (lambda (k) (k 3))))
```

Then, when we **invoke** the continuation, we **abandon** the *current* continuation and **reinstate** the *saved* continuation

```
(+ 4 (call/cc (lambda (k) (k 3))))
```

But in this example, the saved continuation is <u>equivalent</u> to the current continuation, so we observe no difference!

The program never returns from call `(k 3)` because
undelimited continuations run until the program exits.

`call/cc` gives us undelimited (a.k.a. full) continuations.

```
(+ 1 (call/cc (lambda (k) (k 3) (print 0))))
;; =>  3       (print 0) is never reached
```

The program never returns from call `(k 2)` because
undelimited continuations run until the program exits.

`call/cc` gives us undelimited (a.k.a. full) continuations.

```
(+ 1 (call/cc (lambda (k) (k 2) (print 0))))
;; =>  3      (print 0) is never reached
```

Pause the video and type this one into Dr. Racket!

Do you understand why `(print 0)` is never reached?

```
(+ 1 (call/cc (lambda (k) (k 2))))
;; => 3
```

This `call/cc`'s behavior is *roughly* the same as the application:

```
((lambda (k) (k 2))
 (lambda (n) (exit (print (+ 1 n)))))
;; => 3
```

Where the high-lit continuation `(lambda (n) …)` takes a return value for the `(call/cc …)` expression and finishes the program.

When execution reaches **this point**, k is bound as the continuation

```
(+ 4 (call/cc (lambda (k) (+ 5 (k 3)))))

  k = <continuation> (lambda (x) (+ 4 x))
```

When control **reaches** this point, the current continuation is…

```
(lambda (x) (+ 4 (+ 5 x)))

(+ 4 (call/cc (lambda (k) (+ 5 (k 3)))))
```

```
(+ 4 (call/cc (lambda (k) (+ 5 (k 3)))))
```

And, **by invoking k**, then we abandon it to *reinstate* k

```
(lambda (x) (+ 4 x))
```

Try an example. What do each of these 3 examples return?
(Hint: Racket evaluates argument expressions left to right.)

```
(call/cc (lambda (k0)
           (+ 1 (call/cc (lambda (k1)
                           (+ 1 (k0 3)))))))
```

```
(call/cc (lambda (k0)
           (+ 1 (call/cc (lambda (k1)
                           (+ 1 (k0 (k1 3))))))))
```

```
(call/cc (lambda (k0)
           (+ 1
              (call/cc (lambda (k1)
                         (+ 1 (k1 3))))
              (k0 1))))
```

Try an example. What do each of these 3 examples return?
(Hint: Racket evaluates argument expressions left to right.)

```
(call/cc (lambda (k0)
           (+ 1 (call/cc (lambda (k1)
                           (+ 1 (k0 3)))))))
```

3

```
(call/cc (lambda (k0)
           (+ 1 (call/cc (lambda (k1)
                           (+ 1 (k0 (k1 3))))))))
```

4

```
(call/cc (lambda (k0)
           (+ 1
              (call/cc (lambda (k1)
                         (+ 1 (k1 3))))
              (k0 1))))
```

1

Lecture Summary

- Continuations allow us to capture the stack in a first-class way
- call/cc (call-with-current-continuation)
  - Let's us bind special **continuation** functions
- When invoked, continuations **reset the stack**
- As we will soon see, this enables building non-local control constructs (loops, exceptions, etc…)

Closures

CIS352 — Spring 2021

Kris Micinski

```
(let ([cc (call/cc (lambda (k) k))])
  ...)
```

A common idiom for `call/cc` is to let-bind the current continuation.

```
(let ([cc (call/cc (lambda (k) k))])
    ...)
```

Note that applying call/cc on the identity function is exactly the same as applying it on the u-combinator!

```
(let ([cc (call/cc (lambda (k) (k k)))])
    ...)
```

Why is this the case?

```
(let ([cc (call/cc (lambda (k) k))])
  ...)
```

This return point …is the same as this one…

```
(let ([cc (call/cc (lambda (k) (k k)))])
  ...)
```

…and calling k on itself, returns k to itself!

Returning value v is the same as *calling* that saved return point *on* v.