# Church Numerals

CIS352 — Fall 2024

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**
(Encoding works on naturals—adaptable to ints, etc..)

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.*"

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

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

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

$$\cdots$$

**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 (λ (f) (λ (x) x)))
(define one  (λ (f) (λ (x) (f x))))
(define two  (λ (f) (λ (x) (f (f x)))))
```

**Exercise 1:** Write the church encoding of **3**

**Exercise 2:** Write two α-equivalent versions of **0**

Let's say we have a **church-encoded** number, that is a term like

```
(λ (f) (f (f … (f x))…))
```

We can turn it **back** into a Racket number by calling it in "curried" style

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

**Exercise 3:** translate the following Church-encoded numbers to Racket natural numbers

```
(λ (g) (λ (x) (g x)))
(λ (h) (λ (y) (h (h (h y)))))
```

**Observation 3**: when we use this encoding, *any expression α/β/η-equivalent* to n **is** n

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

CBV β

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

CBV β

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

CBV β

```
(λ (z) (λ (x) (z (z x))))
```

This is **2**

**Exercise 4**: Write a derivation sequence to a normal form and obtain the answer for the below term. Note: you **will** have to reduce under lambdas!

```
((λ (z) z)
 (λ (g) ((λ (x) (x x)) (λ (x) x))))
```

**Exercise 4**: Write a derivation sequence to a normal form and obtain the answer for the below term. Note: you **will** have to reduce under lambdas!

```
((λ (z) z)
 (λ (g) ((λ (x) (x x)) (λ (x) x))))
–> (λ (g) ((λ (x) (x x)) (λ (x) x)))
–> (λ (g) ((λ (x) x) (λ (x) x)))
-> (λ (g) (λ (x) x))
```

The solution is **zero**

This also demonstrates the fact that, while β is the primary rule driving computation (function application), determining λ equivalence may require reducing ***under*** *a λ!*

**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) (λ (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 = (λ (f) (λ (x) (f (f … (f x) …)))
```

How can you generate n + 1?

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

"Add another f to the front."

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

“Add another f to the front.”

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

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
```

**Currying**
The λ-calculus supports multi-arg functions easily via currying—every function of `(x0 x1 …)` is written as `(λ (x0) (λ (x1) …))`

***But***, callsites to those functions must be modified as well— `(x0 x1 …)` must become `(…(x0 x1) …)`

**Exercise 5:** Translate the following *Racket* lambda to use the curried style—also translate the callsite of +, assuming it must be curried as well:

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

**Exercise 5:** Translate the following *Racket* lambda to use the curried style—also translate the callsite of +, assuming it must be curried as well:

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

(f x y z)
->
(((f x) y) z)
```

**Question:**

Now how do you do addition…? Observation: need **two** arguments. Use **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))))))
```

Exercise **6**: Write a reduction sequence for the following (after converting 0 and 1 to church numerals)

```
((plus 1) 1)
```

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

```
(n1 f) ;; applies f (to some arg) n1 times
(n0 (n1 f)) ;; "does f n1 times" n0 times in row
```

```
(lambda (n0)
   (lambda (n1)
     (lambda (f) (lambda (x) ((n0 (n1 f)) x)))))
```

```
(lambda (n0)
   (lambda (n1)
     (lambda (f) (lambda (x) ((n0 (n1 f)) x))))
```

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

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