Representing Values

- LambdaCalculus is powerful model:

- λ-Calculus does not come with numerals or arithmetic operations
  but they can be defined...

Q: What does it mean to be "0", "1" or ...

A: Their interaction
wrt *successor*, *addition*, *multiplication*
should work:
$0+3=3 \quad s(0)=1 \quad 0\times 3=0 \quad 1\times 15=15 \quad \ldots$

Representation

- Represent

0	by	$\mathtt{(\lambda f \mid (\lambda x \mid x))}$,
1	by	$\mathtt{(\lambda f \mid (\lambda x \mid (f\ x)))}$,
2	by	$\mathtt{(\lambda f \mid (\lambda x \mid (f\ (f\ x))))}$,
...		
k	by	$\mathtt{(\lambda f \mid (\lambda x \mid \underbrace{(f\ \ldots (f}_{k}\ x)\ldots)))}$
...		

- Successor: $\mathtt{(\lambda x\ s\ z \mid s(x\ s\ z))}$

  Addition: $\mathtt{(\lambda w\ z\ yx \mid wy\ (z\ y\ x))}$

  Multiplication: $\mathtt{(\lambda w\ z\ y \mid w\ (zy))}$

- Don't forget parenthesis order

Does it work?

- What is successor(2)?

$$
\begin{aligned}
&\mathtt{(\lambda \underline{x}\ s\ z \mid s(x\ s\ z))\ \overline{(\lambda f\ x \mid f(f\ x))}}\\
\mapsto\ &\mathtt{(\lambda s\ z \mid s(\ \underline{(\lambda f\ x \mid f(f\ x))}\ s\ z)\ )}\\
\mapsto\ &\mathtt{(\lambda s\ z \mid s(\ (\lambda x \mid s(s\ x))\ z)\ )}\\
\mapsto\ &\mathtt{(\lambda s\ z \mid \underbrace{s(\ s(s}_{3}\ z)))} \quad = 3
\end{aligned}
$$

- Confirm $\mathtt{succ} = \mathtt{add\ 1}$

$$
\begin{aligned}
&\mathtt{add\ 1} \equiv \mathtt{(\lambda \underline{w}zyx \mid wy\ (z\ y\ x))\ \overline{(\lambda f\ x \mid (f\ x))}}\\
\mapsto\ &\mathtt{(\lambda zyx \mid (\lambda \underline{f}\ \overline{x} \mid (f\ x))\ \underline{y}\ \overline{(z\ y\ x)})}\\
\mapsto\ &\mathtt{(\lambda zyx \mid y\ (z\ y\ x))}\\
\mapsto_\alpha\ &\mathtt{(\lambda xsz \mid s\ (x\ s\ z))} \quad \equiv \quad \mathtt{Succ}
\end{aligned}
$$

What is 0×1?

$\mathtt{(Mult\ 0\ 1)}$ should return $0 \equiv \mathtt{(\lambda f\ x \mid x)}$

$$
\begin{aligned}
&\mathtt{Mult\ 0\ 1} \equiv\\
&\mathtt{(\lambda \underline{w}\ \overline{z}\ y \mid w\ (z\ y))\ \underline{(\lambda f\ x \mid x)}\ \overline{(\lambda f\ x \mid f\ x)}}\\
\mapsto\ &\mathtt{(\lambda y \mid [(\lambda f\ x \mid f\ x)\ /\ z]\ [(\lambda f\ x \mid x)\ /\ w]\ (w\ (z\ y)\ ))}\\
\mapsto\ &\mathtt{(\lambda y \mid ((\lambda f\ x \mid x)\ (\ (\lambda \underline{f}\ x \mid f\ x)\ \overline{y})\ )\ )}\\
\mapsto\ &\mathtt{(\lambda y \mid ((\lambda \underline{f}\ x \mid x)\ \overline{(\lambda x \mid y\ x)})\ )\ )}\\
\mapsto\ &\mathtt{(\lambda y \mid (\lambda x \mid x)\ )}\\
\mapsto^\alpha\ &\mathtt{(\lambda f\ x \mid x)}
\end{aligned}
$$

...doesn't matter whether parameter is $\mathtt{y}$ or $\mathtt{f}$

What is 2×3?

$\mathtt{mult\ 2\ 3} = (\lambda\, \mathtt{wzy} \mid \mathtt{w\ (zy)})\ \mathtt{2\ 3}$

$\mapsto (\lambda\, \mathtt{y} \mid \mathtt{2\ (3\ y)})$

$\quad \equiv (\lambda\, \mathtt{y} \mid (\lambda\, \underline{\mathtt{f}}\ \mathtt{x} \mid \mathtt{f\ (f\ x)})\ \overline{\mathtt{(3\ y)}}\ )$

$\mapsto (\lambda\, \mathtt{y} \mid (\lambda\, \mathtt{x} \mid \mathtt{(3\ y)\ ((3\ y)\ x)})\ )$

$\mapsto (\lambda\, \mathtt{y\ x} \mid ((\lambda\, \underline{\mathtt{f}}\ \mathtt{u} \mid \mathtt{f(f(f\ u))})\ \overline{\mathtt{y}})\ \mathtt{((3\ y)\ x)}\ )$

$\mapsto (\lambda\, \mathtt{y\ x} \mid (\lambda\, \underline{\mathtt{u}} \mid \mathtt{y(y(y\ u))})\ \overline{\mathtt{((3\ y)\ x)}})$

$\mapsto (\lambda\, \mathtt{y\ x} \mid \mathtt{y(y(y\ ((3\ y)\ x)\ ))}\ )$

$\mapsto (\lambda\, \mathtt{y\ x} \mid \mathtt{y(y(y\ ((}\ (\lambda\, \underline{\mathtt{f}}\ \mathtt{u} \mid \mathtt{f(f(f\ u))})\ \overline{\mathtt{y}})\ \mathtt{x)\ )))}$

$\mapsto (\lambda\, \mathtt{y\ x} \mid \mathtt{y(y(y\ (}\ (\lambda\, \underline{\mathtt{u}} \mid \mathtt{y(y(y\ u))})\ \overline{\mathtt{x}})\ \mathtt{)))}$

$\mapsto (\ \lambda\, \mathtt{y\ x} \mid \underbrace{\mathtt{y(y(y(\ y(y(y}}_{6}\ \mathtt{x))))))}\ )$

$\equiv 6$

Boolean Expressions

- Define boolean values (T, F)

  $\mathtt{T} = (\lambda\, \mathtt{xy} \mid \mathtt{x})$

  $\mathtt{F} = (\lambda\, \mathtt{xy} \mid \mathtt{y})$

  So T "extract" 1st arg, and ignores 2nd
  F "extract" 2st arg, and ignores 1st

  + ... boolean connectives:
  not, and, or, if

- $\mathtt{Not} = (\lambda\, \mathtt{x} \mid \mathtt{x\ F\ T})$

Eg: $\mathtt{Not\ T} \equiv (\lambda\, \underline{\mathtt{x}} \mid \mathtt{x\ F\ T})\ \overline{(\lambda\, \mathtt{xy} \mid \mathtt{x})}$

$\mapsto (\lambda\, \underline{\mathtt{x}}\ \overline{\mathtt{y}} \mid \mathtt{x})\ \underline{\mathtt{F}}\ \overline{\mathtt{T}}$

$\mapsto \mathtt{F}$

$\mathtt{Not\ F} \equiv (\lambda\, \underline{\mathtt{x}} \mid \mathtt{x\ F\ T})\ \overline{(\lambda\, \mathtt{xy} \mid \mathtt{y})}$

$\mapsto (\lambda\, \underline{\mathtt{x}}\ \overline{\mathtt{y}} \mid \mathtt{y})\ \underline{\mathtt{F}}\ \overline{\mathtt{T}}$

$\mapsto \mathtt{T}$

Other Connectives

- $\mathtt{And} = (\lambda\, \mathtt{x\ y} \mid \mathtt{x\ y\ F})$

  So if x is F, will return "2nd" arg, F
  Otherwise, if x is T, will return "1st" arg, y

Eg: $\mathtt{And\ T\ F} \equiv (\lambda\, \underline{\mathtt{x}}\ \overline{\mathtt{y}} \mid \mathtt{x\ y\ F})\ \underline{(\lambda\, \mathtt{x\ y} \mid \mathtt{x})}\ \overline{\mathtt{F}}$

$\mapsto (\lambda\, \underline{\mathtt{x}}\ \overline{\mathtt{y}} \mid \mathtt{x})\ \underline{\mathtt{F}}\ \overline{\mathtt{F}}$

$\mapsto \mathtt{F}$

$\mathtt{And\ F\ T} \equiv (\lambda\, \underline{\mathtt{x}}\ \overline{\mathtt{y}} \mid \mathtt{x\ y\ F})\ \underline{(\lambda\, \mathtt{xy} \mid \mathtt{y})}\ \overline{\mathtt{T}}$

$\mapsto (\lambda\, \underline{\mathtt{x}}\ \overline{\mathtt{y}} \mid \mathtt{y})\ \underline{\mathtt{T}}\ \overline{\mathtt{F}}$

$\mapsto \mathtt{F}$

- $\mathtt{Or} = (\lambda\, \mathtt{x\ y} \mid \mathtt{x\ T\ y})$

  So if x is F, will return "2nd" arg, y
  Otherwise, if x is T, will return "1st" arg, T

Eg: $\mathtt{Or\ T\ F} \equiv (\lambda\, \underline{\mathtt{x}}\ \overline{\mathtt{y}} \mid \mathtt{x\ T\ y})\ \underline{(\lambda\, \mathtt{xy} \mid \mathtt{x})}\ \overline{\mathtt{F}}$

$\mapsto (\lambda\, \underline{\mathtt{x}}\ \overline{\mathtt{y}} \mid \mathtt{x})\ \underline{\mathtt{T}}\ \overline{\mathtt{F}}$

$\mapsto \mathtt{T}$

$\mathtt{Or\ F\ F} \equiv (\lambda\, \underline{\mathtt{x}}\ \overline{\mathtt{y}} \mid \mathtt{x\ T\ y})\ \underline{(\lambda\, \mathtt{xy} \mid \mathtt{y})}\ \overline{\mathtt{F}}$

$\mapsto (\lambda\, \underline{\mathtt{x}}\ \overline{\mathtt{y}} \mid \mathtt{y})\ \underline{\mathtt{T}}\ \overline{\mathtt{F}}$

$\mapsto \mathtt{F}$

Conditional

- $\mathtt{if\ P\ then\ M\ else\ N} \equiv (\lambda\, \mathtt{uvw} \mid \mathtt{uvw})\ \mathtt{PMN}$

Eg: $\mathtt{if\ T\ then\ M\ else\ N} = (\lambda\, \mathtt{uvw} \mid \mathtt{uvw})\ \mathtt{TMN}$

$\mapsto (\lambda\, \mathtt{xy} \mid \mathtt{x})\ \mathtt{M\ N} \quad \mapsto \quad \mathtt{M}$

$\mathtt{if\ F\ then\ M\ else\ N} = (\lambda\, \mathtt{uvw} \mid \mathtt{uvw})\ \mathtt{FMN}$

$\mapsto (\lambda\, \mathtt{xy} \mid \mathtt{y})\ \mathtt{M\ N} \quad \mapsto \quad \mathtt{N}$

Recall: T pulls out first arg,
F pulls out 2nd arg...

Lists

- Define Lists by...

  `[M, N] = (λz | z M N)`

- Form/Decompose list (`cons`, `car`, `cdr`) by:

  `cons = (λm n z | z m n)` "cons"

  `first = (λx | x T)` "car"

  `rest = (λx | x F)` "cdr"

Eg: $\texttt{cons M N} = (\lambda \underline{\texttt{m}}\ \overline{\texttt{n}}\ \texttt{z | z m n})\ \underline{\texttt{M}}\ \overline{\texttt{N}}$
$\mapsto (\lambda \texttt{z | z M N}) \equiv \texttt{[M, N]}$

$\texttt{first [M, N]} = (\lambda \underline{\texttt{x}}\ \texttt{| x T})\ \overline{(\lambda \texttt{z | z M N})}$
$\mapsto (\lambda \texttt{z | z M N})\ \texttt{T} \mapsto \texttt{T M N} \mapsto \texttt{M}$

$\texttt{rest [M, N]} = (\lambda \underline{\texttt{x}}\ \texttt{| x F})\ \overline{(\lambda \texttt{z | z M N})}$
$\mapsto (\lambda \texttt{z | z M N})\ \texttt{F} \mapsto \texttt{F M N} \mapsto \texttt{N}$

Alternative Def'n of Numbers

- Define numerals as recursive lists:

  $\texttt{0} \equiv (\lambda \texttt{x | x})$

  $\texttt{n + 1} \equiv \texttt{[F, n]}$ for all $n \geq 0$

Eg: `0 = (λx | x)`

`1 = [F, 0] = (λz | z F 0)`
`(λz | z (λfx | x) (λx | x) )`

`2 = [F, 1] = [F, [F, 0]]`

`3 = [F, 2] = [F, [F, [F, 0]]]`

...

$\texttt{k} = \underbrace{\texttt{[F, ..., [F,}}_{k}\ \texttt{0] ...]}$

- Define arithmetic operations

  `succ = (λx | [F, x])`

  `pred = rest = (λx | x F)`

  `zeroP = first = (λx | x T)`

Eg: `zeroP 0 = first 0 = (λx | x T) (λy | y)`
$\mapsto (\lambda \texttt{y | y})\ \texttt{T} \mapsto \texttt{T}$

`zeroP 1 = first [F, 0] = F`

Definition of Plus?

- To define "plus"...
  intuitively should be:

```
plus =
(λx y | if (zeroP x)
        then y
        else (plus (pred x) (succ y)) )
```

but... violates "referential transparency"
Use of "plus" on right-hand side
⇒ a never ending process.

⇒ need a technique to solve recursion

Recursive Definitions

Challenge: Need to remove self-reference from recursively defined λ-exp, while still allowing lambda expression to be computed recursively.

```
plus = (λx y | if (zeroP x)
     then y
     else plus (pred x) (succ y))
plus 1 2  ≡
(λx y | if (zeroP x) then y else (plus (pred x) (succ y)))
 1 2  ↦
if (zeroP 1) then 2 else (plus (pred 1) (succ 2))  ↦
(plus (pred 1) (succ 2))  ↦
plus 0 3 ≡
(λx y | if (zeroP x) then y else (plus (pred x) (succ y)))
 0 3  ↦
if (zeroP 0) then 3 else (plus (pred 0) (succ 3))
 ↦ 3
```

How?

Evaluate Recursively

- ∃ lambda expr's that evaluate recursively without self-references:

```
Eg: (λx | x x) (λx | x x)
    ↦ (λx | x x) (λx | x x)
```

```
Let: A = (λx | R(x x)) (λx | R(x x))
     ↦ R( (λx | R(x x)) (λx | R(x x)) )
     ↦ R(R( (λx | R(x x)) (λx | R(x x)) ))
     ↦ ...
     R(R(...R( (λx | R(x x)) (λx | R(x x)) )...))
     = R(R(...R(A)...))
```

- Easy to generate recursive calls
  but need to generate recursive calls
  based on *value of argument*,
  and **stop when the "base" case occurs**

Fixed-point Combinator

- Define "fixed-point combinator"

```
Y = (λy | (λx | y(x x)) (λx | y(x x)))

Y R  ↦ (λy | (λx | y(x x)) (λx | y(x x)))R
↦ (λx | R(x x)) (λx | R(x x))
↦ R ((λx | R(x x)) (λx | R(x x))) = R(Y R)
↦ ... ↦
R(R(R(... R(Y R) ...)))
```

- If R is function of multiple arguments then:

```
((Y R) A B ...)   ↦ (R(Y R) A B ...)
```

⇒ first argument to R is (Y R),
  others are A, B, ...

- If R is constructed properly, then
  will recur until certain values of A, B, ...
  end recursion.

Recursive Def'n of Plus

```
Rplus ≡
(λf x y | if (zeroP x) then y else (f (pred x) (succ y)))

((Y Rplus) 1 2)  ↦ (Rplus (Y Rplus) 1 2)  ↦

(λf x y | if (zeroP x) then y else (f (pred x) (succ y)))
  (Y Rplus) 1 2  ↦

if (zeroP 1) then 2 else ((Y Rplus) (pred 1) (succ 2))

 ↦  ((Y Rplus) (pred 1) (succ 2))  ↦

((Y Rplus) 0 3)  ↦ (Rplus (Y Rplus) 0 3)  ↦

(λf x y | if (zeroP x) then y else (f (pred x) (succ y)))
 (YRplus) 0 3  ↦

if (zeroP 0) then 3 else ((Y Rplus) (pred 0) (succ 3)
(YRplus))

 ↦  3
```


Summation

```
Sum(n) ≡ if (zeroP n) then 0 else (n + sum(n-1))

Rsum ≡
(λf n | if (zeroP n) then 0 else (+ n (f (pred x))))
((Y Rsum) 2)  ↦ (Rsum (Y Rsum) 2)  ↦

if (zeroP 2) then 0 else (+ 2 ((Y RSum) (pred 2)))
↦   (+ 2 ((Y RSum) 1))
```

Now focus on ...

```
((Y RSum) 1)  ↦ (Rsum (Y Rsum) 1)  ↦
if (zeroP 1) then 0 else (+ 1 ((Y RSum) (pred 1)))

 ↦   (+ 1 ((Y RSum) 0))
```

Finally, focus on ...

```
((Y RSum) 0)  ↦ (Rsum (Y Rsum) 0)  ↦
if (zeroP 0) then 0 else (+ 0 ((Y RSum) (pred 0)))  ↦

0
```

Putting the pieces together. . .

```
((Y Rsum) 2)  ↦ (+ 2 ((Y Rsum) 1))  ↦
(+ 2 (+ 1 ((Y Rsum) 0)))  ↦ (+ 2 (+ 1 0))
↦ (+ 2 1)  ↦ 3
```