What if we pass $\Omega$ as an argument to another function?

```haskell
let OMEGA = (\x -> x x) (\x -> x x) (\x -> (\y -> y)) OMEGA
```

Does this reduce to a normal form? Try it at home!

---

**Programming in $\lambda$-calculus**

*Real languages have lots of features*

- Booleans
- Records (structs, tuples)
- Numbers
- **Functions** [we got those]
- Recursion

---

API

- Operations
Lets see how to *encode* all of these features with the $\lambda$-calculus.

**$\lambda$-calculus: Booleans**

How can we encode Boolean values (TRUE and FALSE) as functions?

Well, what do we do with a Boolean $b$?

$$\begin{align*}
\text{TRUE} \text{ OR } \text{FALSE} \quad \text{AND} : (\text{BOOL}, \text{BOOL}) \rightarrow \text{BOOL} \\
\text{NOT} : \text{BOOL} \rightarrow \text{BOOL} \\
\text{IF } \text{COND THEN } \_ \_ \text{ ELSE } \_ 
\end{align*}$$
Make a binary choice

- if \( b \) then \( e_1 \) else \( e_2 \)

\[ b \ ? \ e_1 : e_2 \]

**Booleans: API**

We need to define three functions

\[
\begin{align*}
\text{let } TRUE &= ??? \\
\text{let } FALSE &= ??? \\
\text{let } ITE &= \lambda b \times y \rightarrow ??? \quad -- \text{if } b \text{ then } x \text{ else } y \\
\end{align*}
\]

such that

\[
\begin{align*}
ITE \ TRUE \ apple \ banana &= \rightarrow \ apple \\
ITE \ FALSE \ apple \ banana &= \rightarrow \ banana
\end{align*}
\]

(Here, let NAME = e means NAME is an abbreviation for e)

\[
\begin{align*}
\text{let } TRUE &= ? \\
\text{let } FALSE &= ?
\end{align*}
\]

\[
\begin{align*}
TRUE \ apple \ ban &= \Rightarrow \ apple \\
FALSE \ apple \ ban &= \Rightarrow \ ban
\end{align*}
\]
Booleans: Implementation

let TRUE = \x \y -> \x        -- Returns its first argument
let FALSE = \x \y -> \y        -- Returns its second argument
let ITE = \b \x \y -> \b \x \y  -- Applies condition to branches
                           -- (redundant, but improves read ability)

Example: Branches step-by-step
Example: Branches step-by-step

Now you try it!

Can you fill in the blanks to make it happen? (http://goto.ucsd.edu:8095/index.html#?demo=ite.lc)

eval ite_false:

ITE FALSE e1 e2

-- fill the steps in!

=b> e2
**EXERCISE: Boolean Operators**

ELSA: https://goto.ucsd.edu/elsa/index.html Click here to try this exercise (https://goto.ucsd.edu/elsa/index.html#?demo=permalink%2F1585435168_24442.lc)

Now that we have ITE it’s easy to define other Boolean operators:

```plaintext
let NOT = \b -> ???
let OR = \b1 b2 -> ???
let AND = \b1 b2 -> ???
```

When you are done, you should get the following behavior:
eval ex_not_t:
  NOT TRUE  =>  FALSE

eval ex_not_f:
  NOT FALSE =>  TRUE

eval ex_or_ff:
  OR FALSE FALSE  =>  FALSE

eval ex_or_ft:
  OR FALSE TRUE  =>  TRUE

eval ex_or_ft:
  OR TRUE FALSE  =>  TRUE

eval ex_or_tt:
  OR TRUE TRUE  =>  TRUE

eval ex_and_ff:
  AND FALSE FALSE  =>  FALSE

eval ex_and_ft:
  AND FALSE TRUE  =>  FALSE

eval ex_and_ft:
  AND TRUE FALSE  =>  FALSE

eval ex_and_tt:
  AND TRUE TRUE  =>  TRUE
Programming in $\lambda$-calculus

- **Booleans** [done]
- Records (structs, tuples)
- Numbers
- Functions [we got those]
- Recursion

Pairs

\[ \text{get first elem} \]
\[ \text{get second elem} \]

\[ \text{getFst}(\text{mkPair } \text{elem1 } \text{elem2}) \]
\[ = \text{elem1} \]

\[ \text{getSnd}(\text{mkPair } \text{elem1 } \text{elem2}) \]
\[ = \text{elem2} \]
λ-calculus: Records

Let’s start with records with two fields (aka pairs)

What do we do with a pair?

1. Pack two items into a pair, then
2. Get first item, or
3. Get second item.

Pairs: API

We need to define three functions
let PAIR = \x y -> ??? -- Make a pair with elements x and y

-- { fst : x, snd : y }

let FST = \p -> ??? -- Return first element
-- p.fst

let SND = \p -> ??? -- Return second element
-- p.snd

such that

eval ex_fst:
  FST (PAIR apple banana) =*\> apple

eval ex_snd:
  SND (PAIR apple banana) =*\> banana

Pairs: Implementation

A pair of x and y is just something that lets you pick between x and y!
(i.e. a function that takes a boolean and returns either \( x \) or \( y \))

```haskell
let PAIR = \( x \ y \rightarrow (\b \rightarrow \text{ITE } b \ x \ y) )
let FST = \( p \rightarrow p \ TRUE \)  -- call w/ \text{TRUE}, get first value
let SND = \( p \rightarrow p \ FALSE \)  -- call w/ \text{FALSE}, get second value
```

**EXERCISE: Triples**

How can we implement a record that contains three values?

ELSA: https://goto.ucsd.edu/elsa/index.html

Click here to try this exercise (https://goto.ucsd.edu/elsa/index.html#?demo=permalink%2F1585434814__24436.lc)
let TRIPLE = \x y z -> ???
let FST3 = \t -> ???
let SND3 = \t -> ???
let THD3 = \t -> ???

eval ex1:  
  FST3 (TRIPLE apple banana orange)  
  =>> apple  

eval ex2:  
  SND3 (TRIPLE apple banana orange)  
  =>> banana  

eval ex3:  
  THD3 (TRIPLE apple banana orange)  
  =>> orange

Programming in λ-calculus
λ-calculus: Numbers

Let’s start with natural numbers (0, 1, 2, ...)

What do we do with natural numbers?

- Count: 0, inc
- Arithmetic: dec, +, -, *
- Comparisons: ==, <=, etc
Natural Numbers: API

We need to define:

- A family of **numerals**: ZERO, ONE, TWO, THREE, ...
- Arithmetic functions: INC, DEC, ADD, SUB, MULT
- Comparisons: IS_ZERO, EQ, LEQ

Such that they respect all regular laws of arithmetic, e.g.

IS_ZERO ZERO =~> TRUE
IS_ZERO (INC ZERO) =~> FALSE
INC ONE =~> TWO
...

https://ucsd-cse230.github.io/fa20/lectures/01-lambda.html
Natural Numbers: Implementation

Church numerals: a number $N$ is encoded as a combinator that calls a function on an argument $N$ times

```plaintext
let ONE = /f x -> f x
let TWO = /f x -> f (f x)
let THREE = /f x -> f (f (f x))
let FOUR = /f x -> f (f (f (f x)))
let FIVE = /f x -> f (f (f (f (f x))))
let SIX = /f x -> f (f (f (f (f (f x)))))
...
let N = /f x -> f ... (f (f x))
```

$N$-times
**QUIZ: Church Numerals**

Which of these is a valid encoding of ZERO?

- A: `let ZERO = \f x -> x`
- B: `let ZERO = \f x -> f`
- C: `let ZERO = \f x -> f x`
- D: `let ZERO = \x -> x`
- E: None of the above

Does this function look familiar?

\[ \text{let } N = \lambda f \ x \rightarrow f \ldots (f (f (f x))) \]

\(N\)-times

\[ \lambda\text{-calculus: Increment} \]