```
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 λ-calculus

Real languages have lots of features

func

$$\boxed{e := \underset{\substack{\uparrow \\ \text{var}}}{x} \mid \backslash x \to e \mid \underset{\substack{\uparrow \\ \text{calls}}}{(e_1\ e_2)}}$$

- Booleans
- Records (structs, tuples)
- Numbers
- ✓ **Functions** [we got those]
- Recursion cool NOT REQ for assignment!

Lets see how to *encode* all of these features with the λ-calculus.

λ-calculus: Booleans

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

Well, what do we **do** with a Boolean `b` ?

① branching "decisions" ✓

② AND, OR, NOT

ITE

$\backslash x \to (\backslash y \to x)$ "return 1st input"

$\backslash x \to (\backslash y \to y)$ "retur 2nd input"

$(\backslash x_1\ x_2 \to x_1)$

TRUE x y

$\overset{*}{\Rightarrow}$ x

FALSE x y

$\overset{*}{\Rightarrow}$ y

$(\backslash x_1\ x_2 \to x_2)$

Make a *binary choice*

- `if b then e1 else e2`

ITE $\underline{b}$ e_1 e_2

↓

"If-then-else"

ITE TRUE e_1 e_2

$\rightsquigarrow$ e_1

ITE FALSE e_2 e_2

$\rightsquigarrow$ e_2

ITE ✓

Booleans: API

We need to define three functions

```
let TRUE  = ???
let FALSE = ???
let ITE   = \b x y -> ???  -- if b then x else y
```

such that

```
ITE TRUE apple banana =~> apple
ITE FALSE apple banana =~> banana
```

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

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 readability)
```

Example: Branches step-by-step

```
eval ite_true:
  ITE TRUE e1 e2
  =d> (\b x y -> b    x  y) TRUE e1 e2    -- expand def ITE
  =b>   (\x y -> TRUE x  y)      e1 e2    -- beta-step
  =b>     (\y -> TRUE e1 y)         e2    -- beta-step
  =b>            TRUE e1 e2               -- expand def TRUE
  =d>     (\x y -> x) e1 e2               -- beta-step
  =b>       (\y -> e1)   e2               -- beta-step
  =b> e1
```

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) [Note to self: PASTE link in CHAT!]

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

```
let NOT = \b      -> ~~???~~ ITE b FALSE TRUE
let OR  = \b1 b2 -> ???  ITE b1 TRUE b2
let AND = \b1 b2 -> ???  ITE b1 b2 FALSE
```

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 λ-calculus

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

pack / add

get / retrive

(STORE lamp glass)

GET 1 (STORE lamp glass)

=> lamp

GET 2 (STORE lamp glass)

=> glass

$\backslash b \rightarrow \boxed{\begin{matrix} d_1 \\ d_2 \end{matrix}}$

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`)

```
let PAIR = \x y -> (\b -> ITE b x y)
let FST  = \p -> p TRUE   -- call w/ TRUE, get first value
let SND  = \p -> p FALSE  -- call w/ 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
```