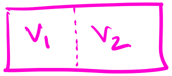
Programming in λ-calculus

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



"pack v_1 v_2" = λchoice → ITE choice v_1 v_2

fst box = \choice → choice TRUE

= box TRUE

snd = \box → box False

F x = e

F = \x → e

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

```
FST (PAIR apple banana) =~> apple
SND (PAIR apple banana) =~> banana
```

Pairs: Implementation

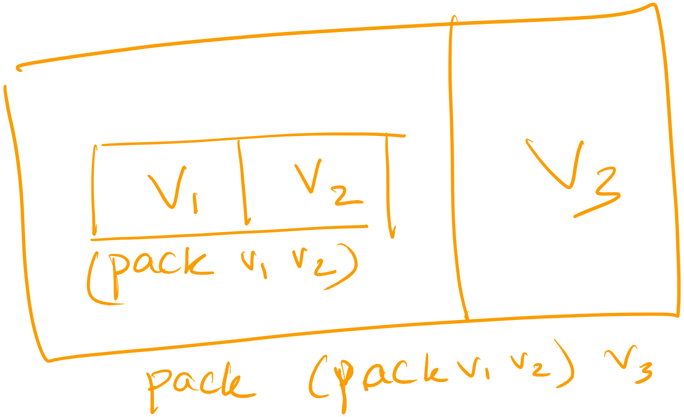
A pair of `x` and `y` is just something that lets you pick between `x` and `y`! (I.e. a function that takes a boolean and returns either `x` or `y`)

```
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?

```
let TRIPLE = \x y z -> ???(
let FST3   = \t -> ???
let SND3   = \t -> ???
let TRD3   = \t -> ???
```



Programming in λ-calculus

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

5

"3" `\f x -> f(f(f x))`

"0" `\f x -> x`

"5" `\f x -> f(f(f(f(f x))))`

"n" $\backslash f\ x \rightarrow \underbrace{f \dots (f}_{n}\ x)$

operators

inc, dec, add, sub, mul, sqrt

compare

eq, less

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`

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

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

Natural Numbers: Implementation

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

```
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)))))
...
```

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?

λ-calculus: Increment

```
-- Call `f` on `x` one more time than `n` does
let INC   = \n -> (\f x -> ???)
```

f(n f x)

THREE = \f x -> f (f (f x)))

goo

aa

FIVE goo aa

goo (goo

=> goo (goo (goo (aa)))

n goo aa

$$=> goo(\underbrace{goo \ldots (goo\ aaa)}_{n}))$$

Example:

```
eval inc_zero :
  INC ZERO
  =d> (\n f x -> f (n f x)) ZERO
  =b> \f x -> f (ZERO f x)
  =*> \f x -> f x
  =d> ONE
```

QUIZ

How shall we implement `ADD` ?

A. **`let`** `ADD = \n m -> n INC m`

num

f

n f x

n Inc m

$x \; \underbrace{(Inc \ldots Inc (Inc (Inc\ m)))}_{n}$

B. `let ADD = \n m ->` $\overset{n+1}{(\texttt{INC n})}$ `m`

3(4)

C. `let ADD = \n m -> n m INC`

D. `let ADD = \n m -> n (m INC)`

E. `let ADD = \n m -> n (INC m)`

$$\text{MUL } m \ n$$

$$= m + m + \ldots + (m + 0)$$

$$\underbrace{m \qquad m + (m + (m + 0))}_{n}$$

λ-calculus: Addition

```
-- Call `f` on `x` exactly `n + m` times
let ADD = \n m -> n INC m
```

Example:

```
eval add_one_zero :
  ADD ONE ZERO
  =~> ONE
```

QUIZ

How shall we implement `MULT`?

A. `let MULT = \n m -> n ADD m`

B. `let MULT = \n m -> n (ADD m) ZERO`

C. `let MULT = \n m -> m (ADD n) ZERO`

D. `let MULT = \n m -> n (ADD m ZERO)`

E. `let MULT = \n m -> (n ADD m) ZERO`

λ-calculus: Multiplication

```
--  Call `f` on `x` exactly `n * m` times
let MULT = \n m -> n (ADD m) ZERO
```

Example:

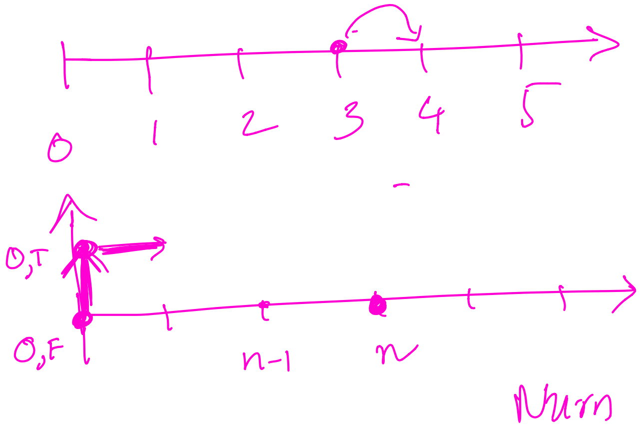
```
eval two_times_three :
  MULT TWO ONE
  =~> TWO
```

Programming in λ-calculus

- **Booleans** [done] ✓

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

- ISZERO
- DEC
- SKIP1



n f x

λ-calculus: Recursion

I want to write a function that sums up natural numbers up to `n`:

```
\n -> ...              -- 1 + 2 + ... + n
```

sum = \n → 1+2+3+...+N

QUIZ

Is this a correct implementation of `SUM`?

```
let SUM = \n -> ITE (ISZ n)
            ZERO
            (ADD n (SUM (DEC n)))
```

A. Yes

B. No

No!

- Named terms in Elsa are just syntactic sugar
- To translate an Elsa term to λ-calculus: replace each name with its definition

```
\n -> ITE (ISZ n)
        ZERO
        (ADD n (SUM (DEC n))) -- But SUM is not a thing!
```

Recursion:

- Inside this function I want to call *the same function* on `DEC n`

Looks like we can't do recursion, because it requires being able to refer to functions *by name*, but in λ-calculus functions are *anonymous*.

Right?

λ-calculus: Recursion

Think again!

Recursion:

- ~~Inside this function I want to call *the same function* on `DEC n`~~
- Inside this function I want to call *a function* on `DEC n`
- *And BTW*, I want it to be the same function

Step 1: Pass in the function to call “recursively”

```
let STEP =
  \rec -> \n -> ITE (ISZ n)
                  ZERO
                  (ADD n (rec (DEC n))) -- Call some rec
```

Step 2: Do something clever to `STEP`, so that the function passed as `rec` itself becomes

```
\n -> ITE (ISZ n) ZERO (ADD n (rec (DEC n)))
```

λ-calculus: Fixpoint Combinator

Wanted: a combinator `FIX` such that `FIX STEP` calls `STEP` with itself as the first argument:

```
FIX STEP
=*> STEP (FIX STEP)
```

(In math: a *fixpoint* of a function $f(x)$ is a point x, such that $f(x) = x$)

Once we have it, we can define:

```
let SUM = FIX STEP
```

Then by property of `FIX` we have:

```
SUM =*> STEP SUM -- (1)
```

```
eval sum_one:
  SUM ONE
  =*> STEP SUM ONE                    -- (1)
  =d> (\rec n -> ITE (ISZ n) ZERO (ADD n (rec (DEC n)))) SUM ONE
  =b> (\n -> ITE (ISZ n) ZERO (ADD n (SUM (DEC n)))) ONE
                                      -- ^^^ the magic happened!
  =b> ITE (ISZ ONE) ZERO (ADD ONE (SUM (DEC ONE)))
  =*> ADD ONE (SUM ZERO)              -- def of ISZ, ITE, DEC, ...
  =*> ADD ONE (STEP SUM ZERO)         -- (1)
  =d> ADD ONE
        ((\rec n -> ITE (ISZ n) ZERO (ADD n (rec (DEC n)))) SUM ZERO)
  =b> ADD ONE ((\n -> ITE (ISZ n) ZERO (ADD n (SUM (DEC n)))) ZERO)
  =b> ADD ONE (ITE (ISZ ZERO) ZERO (ADD ZERO (SUM (DEC ZERO))))
  =b> ADD ONE ZERO
  =~> ONE
```

How should we define `FIX` ???

The Y combinator

Remember Ω?

```
(\x -> x x) (\x -> x x)
=b> (\x -> x x) (\x -> x x)
```

This is *self-replcating code*! We need something like this but a bit more involved…

The Y combinator discovered by Haskell Curry:

```
let FIX   = \stp -> (\x -> stp (x x)) (\x -> stp (x x))
```

$$\underbrace{m + (m + (\ldots + (m}_{n} + 0)))$$

How does it work?

$$m + m + m + m$$

```
eval fix_step:
  FIX STEP
  =d> (\stp -> (\x -> stp (x x)) (\x -> stp (x x))) STEP
  =b> (\x -> STEP (x x)) (\x -> STEP (x x))
  =b> STEP ((\x -> STEP (x x)) (\x -> STEP (x x)))
  --       ^^^^^^^^^^^ this is FIX STEP ^^^^^^^^^^^^
```

That's all folks!

(https://ucsd-cse130.github.io/sp19/feed.xml) (https://twitter.com/ranjitjhala)
(https://plus.google.com/u/0/104385825850161331469) (https://github.com/ranjitjhala)