Programming in $\lambda$-calculus

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

\[
\begin{array}{c|c}
v_1 & v_2 \\
\end{array}
\]

```
"pack $v_1, v_2" = \lambda \text{choice} \to \text{if choice then } v_1 \text{ else } v_2$
```

```
\text{fst } \text{box} &= \lambda \text{choice} \to \text{if } \text{choice} \text{ then } \text{TRUE} \text{ else } \text{box TRUE} \\
\text{snd } \text{box} &= \lambda \text{box} \to \text{box FALSE}
```

```
F x = e \\
F = \lambda x \to e
```

\textit{\textbf{\textlambda}-calculus: Records}

Let’s start with records with two fields (aka \textit{pairs})

What do we \textit{do} with a pair?

1. Pack \textit{two} items into a pair, then
2. Get \textit{first} item, or
3. Get \textit{second} item.
**Pairs : API**

We need to define three functions

```haskell
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 $y$)

```plaintext
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?

\[
\begin{align*}
\text{let } & \text{TRIPLE }= \lambda x\ y\ z \to ??? \\\ (1) \\
\text{let } & \text{FST3 }= \lambda t \to ??? \\
\text{let } & \text{SND3 }= \lambda t \to ??? \\
\text{let } & \text{TRD3 }= \lambda t \to ??? \\
\end{align*}
\]
Programming in $\lambda$-calculus

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

\[
\begin{align*}
5 & \equiv \text{3}^{\text{3}} \quad \lambda x \to \text{f(f(f(x)))} \\
\text{"0"} & \equiv \lambda x \to x \\
\text{"5"} & \equiv \lambda x \to \text{f(f(f(f(f(x)))))} \\
\text{"n"} & \equiv \lambda x \to \text{f} \ldots \text{f}(x)
\end{align*}
\]

$\lambda$-calculus: Numbers

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

What do we do with natural numbers?
- Count: \( \emptyset \), \text{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.
\begin{align*}
\text{IS}_\text{ZERO} \ \text{ZERO} & = \text{TRUE} \\
\text{IS}_\text{ZERO} \ (\text{INC} \ \text{ZERO}) & = \text{FALSE} \\
\text{INC} \ \text{ONE} & = \text{TWO} \\
\end{align*}

\textbf{Natural Numbers: Implementation}

\textit{Church numerals:} a \textit{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: \texttt{let ZERO = \textbackslash f x -> x}
• B: \texttt{let ZERO = \textbackslash f x -> f}
• C: \texttt{let ZERO = \textbackslash f x -> f x}
• D: \texttt{let ZERO = \textbackslash 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 -> ???)

\((\n f x)\)

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

\(f oo \ a a\)

\(\Rightarrow g oo (g oo (g oo (a a))))\)

\(n \ g oo \ a a\)

\(\Rightarrow g oo (g oo \ldots (g oo a a a)\))

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

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

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

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

\[ MUL \ m \ n \]

\[ = \ m + m + \ldots + (m + 0) \]

\[ = \ m + (m + (m + 0)) \]

\[ \underbrace{\ldots}_{h} \]

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

-- Call `f` on `x` exactly `n + m` times

let ADD = \n m -> n INC m
Example:

```plaintext
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

\textit{λ-calculus: Multiplication}

-- Call \texttt{f} on \texttt{x} exactly \texttt{n * m} times

let MULT = \n m -> n (ADD m) ZERO
Example:

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

*Programming in \(\lambda\)-calculus*

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

<λ-calculus: Recursion>

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

\[ \text{sum} = n \rightarrow 1 + 2 + \ldots + n \]
**QUIZ**

Is this a correct implementation of $\text{SUM}$?

```plaintext
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 $\lambda$-calculus: replace each name with its definition

```latex
\text{n} \rightarrow \text{ITE} (\text{ISZ} \ n)
    \quad \text{ZERO}
    \quad (\text{ADD} \ n (\text{SUM} (\text{DEC} \ n))) \quad \text{-- But SUM is not a thing!}
```

Recursion:

- Inside this function I want to call the same function on $\text{DEC} \ n$
Looks like we can’t do recursion, because it requires being able to refer to functions \textit{by name}, but in \(\lambda\)-calculus functions are \textit{anonymous}.

Right?

\textbf{\(\lambda\)-calculus: Recursion}

Think again!
• 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)))
```
Wirte: a combinator FIX such that FIX STEP calls STEP with itself as the first argument:

\[
\text{FIX} \ \text{STEP} \\
\rightarrow \ \text{STEP} (\text{FIX} \ \text{STEP})
\]

(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:

\[
\text{let SUM} = \text{FIX} \ \text{STEP}
\]
Then by property of FIX we have:

\[
\text{SUM} \implies \text{STEP SUM} \quad - \quad (1)
\]

eval sum_one:

\[
\begin{align*}
\text{SUM ONE} \\
\implies \text{STEP SUM ONE} & \quad - \quad (1) \\
\implies (\text{rec } n \rightarrow \text{ITE (ISZ n) ZERO (ADD n (rec (DEC n))))) \quad \text{SUM ONE} \\
\implies (\text{n } \rightarrow \text{ITE (ISZ n) ZERO (ADD n (SUM (DEC n))))) \quad \text{ONE} & \quad - \quad ^^^ \text{ the magic happened!} \\
\implies \text{ITE (ISZ ONE) ZERO (ADD ONE (SUM (DEC ONE)))} \\
\implies \text{ADD ONE (SUM ZERO)} & \quad - \quad \text{def of ISZ, ITE, DEC, ...} \\
\implies \text{ADD ONE (STEP SUM ZERO)} & \quad - \quad (1) \\
\implies (\text{rec } n \rightarrow \text{ITE (ISZ n) ZERO (ADD n (rec (DEC n))))) \quad \text{SUM ZERO} \\
\implies \text{ADD ONE (\text{n } \rightarrow \text{ITE (ISZ n) ZERO (ADD n (SUM (DEC n))))) \quad ZERO} \\
\implies \text{ADD ONE (ITE (ISZ ZERO) ZERO (ADD ZERO (SUM (DEC ZERO))))} \\
\implies \text{ADD ONE ZERO} \\
\implies \text{ONE}
\end{align*}
\]

How should we define FIX???
The **Y combinator**

Remember Ω?

\[
(\lambda x \to x \ x) \ (\lambda x \to x \ x)
\]
\[
=b> \ (\lambda x \to x \ x) \ (\lambda x \to x \ x)
\]

This is *self-replicating 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))
```
How does it work?

eval fix_step:

\[
\begin{align*}
\text{fix step} & : \mathbb{N} \to \mathbb{N} \\
\text{fix step} & = d > (x \to \text{step} (x x)) (x \to \text{step} (x x)) \\
& = b > (x \to \text{step} (x x)) (x \to \text{step} (x x)) \\
& = b > \text{step} (x \to \text{step} (x x)) (x \to \text{step} (x x)) \\
& \quad \quad \quad \text{this is FIX STEP} \\
\end{align*}
\]

That’s all folks!
