

CSE 114A: Fall 2021

# Introduction to Functional Programming

*Lambda Calculus*

Owen Arden

UC Santa Cruz

*Based on course materials developed by Ranjit Jhala*

# Your favorite language

---

- Probably has lots of features:
  - Assignment ( $x = x + 1$ )
  - Booleans, integers, characters, strings,...
  - Conditionals
  - Loops, return, break, continue
  - Functions
  - Recursion
  - References / pointers
  - Objects and classes
  - Inheritance
  - ... and more

# Your favorite language

---

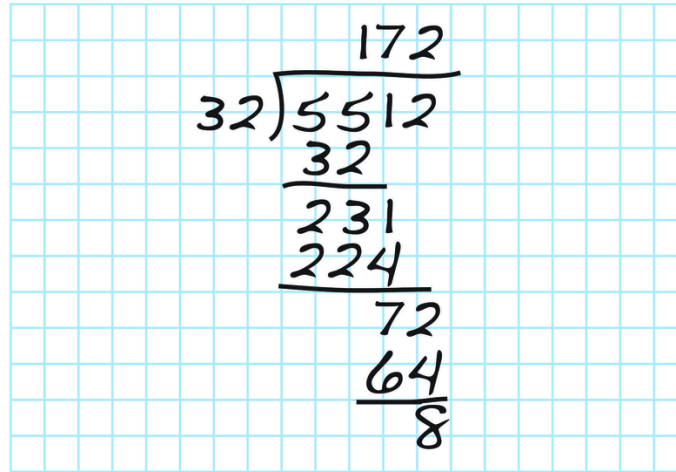
- Probably has lots of features:
  - Assignment ( $x = x + 1$ )
  - Booleans, integers, characters, strings,...
  - Conditionals
  - Loops, return, break, continue
  - References / pointers
  - Objects and classes
  - Inheritance
  - ... and more

Which ones can we do without?  
What is the smallest universal language?

# What is computable?

---

- Prior to 1930s
  - Informal notion of an effectively calculable function:



A handwritten long division problem is shown on a light blue grid background. The problem is  $32 \overline{) 5512}$ . The quotient is 172, and the remainder is 8. The steps are as follows: 32 goes into 55 one time (1), leaving a remainder of 23. Bring down the next digit, 1, to get 231. 32 goes into 231 seven times (7), leaving a remainder of 7. Bring down the next digit, 2, to get 72. 32 goes into 72 two times (2), leaving a remainder of 8.

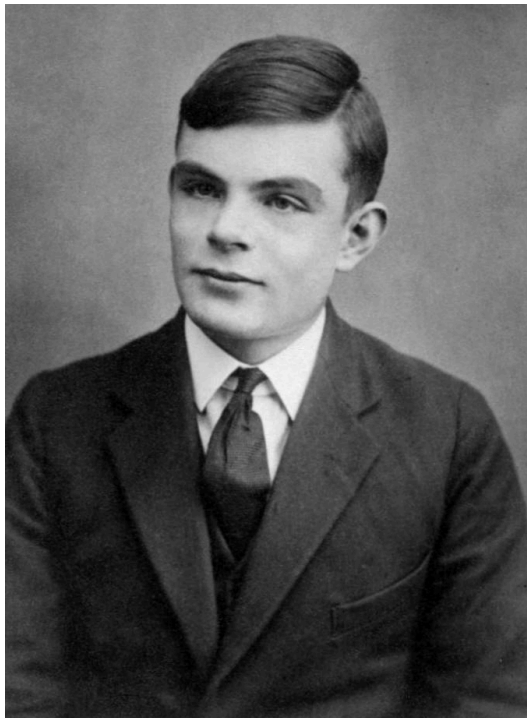
$$\begin{array}{r} 172 \\ 32 \overline{) 5512} \\ \underline{32} \phantom{00} \\ 231 \phantom{0} \\ \underline{224} \phantom{0} \\ 72 \\ \underline{64} \\ 8 \end{array}$$

One that can be computed by a human with pen and paper, following an algorithm

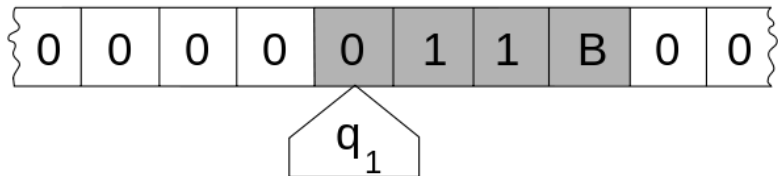
# What is computable?

---

- 1936: Formalization



Alan Turing: Turing machines



# What is computable?

---

- 1936: Formalization



Alonzo Church: lambda calculus

$$\begin{array}{l} e ::= x \\ \quad | \lambda x . e \\ \quad | e_1 e_2 \end{array}$$

# The Next 700 Languages

---

- Big impact on language design!



*Whatever the next 700 languages turn out to be, they will surely be variants of lambda calculus.*

Peter Landin, 1966

# Your favorite language

---

- Probably has lots of features:
  - Assignment ( $x = x + 1$ )
  - Booleans, integers, characters, strings,...
  - Conditionals
  - Loops, return, break, continue
  - Functions
  - Recursion
  - References / pointers
  - Objects and classes
  - Inheritance
  - ... and more



# The Lambda Calculus

---

- Features
  - Functions
  - (that's it)

# The Lambda Calculus

---

- Seriously...
  - ~~Assignment ( $x = x + 1$ )~~
  - ~~Booleans, integers, characters, strings,...~~
  - ~~Conditionals~~
  - ~~Loops, return, break, continue~~
  - **Functions**
  - ~~Recursion~~
  - ~~References / pointers~~
  - ~~Objects and classes~~
  - ~~Inheritance~~
  - ~~... and more~~

The only thing you can do is:  
**Define** a function  
**Call** a function

# Describing a Programming Language

---

- **Syntax**
  - What do programs *look like*?
- **Semantics**
  - What do programs *mean*?
  - **Operational semantics:**
    - How do programs *execute step-by-step*?

# Syntax: What programs look like

---

$$\begin{array}{l} e ::= x \\ \quad | \lambda x \rightarrow e \\ \quad | e1 \ e2 \end{array}$$

- Programs are ***expressions***  $e$  (also called  $\lambda$ -terms)
- **Variable:**  $x, y, z$
- **Abstraction** (aka nameless function definition):
  - $\lambda x \rightarrow e$  “*for any  $x$ , compute  $e$* ”
  - $x$  is the *formal parameter*,  $e$  is the *body*
- **Application** (aka function call):
  - $e1 \ e2$  “*apply  $e1$  to  $e2$* ”
  - $e1$  is the *function*,  $e2$  is the *argument*

# Examples

---

-- The identity function ("for any x compute x")

```
\x -> x
```

-- A function that returns the identity function

```
\x -> (\y -> y)
```

-- A function that applies its argument to

-- the identity function

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

# Examples

---

-- The identity function ("for any x compute x")

```
\x -> x
```

-- A function that returns the identity function

```
\x -> (\y -> y)
```

-- A function that applies its argument to

-- the identity function

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

- How do I define a function with two arguments?
  - e.g. a function that takes x and y and returns y

# Examples

---

-- *A function that returns the identity function*

```
\x -> (\y -> y)
```

OR: a function that takes two arguments  
and returns the second one!

- How do I define a function with two arguments?
  - e.g. a function that takes x and y and returns y

# Examples

---

- How do I apply a function to two arguments?
  - e.g. apply `\x -> (\y -> y)` to apple and banana?

*-- first apply to apple, then apply the result to banana*

```
(((\x -> (\y -> y)) apple) banana)
```



# Syntactic Sugar

---

- Convenient notation used as a shorthand for valid syntax

instead of	we write
<code>\x -&gt; (\y -&gt; (\z -&gt; e))</code>	<code>\x -&gt; \y -&gt; \z -&gt; e</code>
<code>\x -&gt; \y -&gt; \z -&gt; e</code>	<code>\x y z -&gt; e</code>
<code>((e1 e2) e3) e4</code>	<code>e1 e2 e3 e4</code>

`\x y -> y`      *-- A function that that takes two arguments  
-- and returns the second one...*

`(\x y -> y) apple banana` *-- ... applied to two arguments*

# Semantics: What programs mean

---

- How do I “run” or “execute” a  $\lambda$ -term?
- Think of middle-school algebra:

*-- Simplify expression:*

$(x + 2) * (3 * x - 1)$

=

???

- **Execute** = rewrite step-by-step following simple rules until no more rules apply

# Rewrite rules of lambda calculus

---

1.  $\alpha$ -step (aka renaming formals)
2.  $\beta$ -step (aka function call)

But first we have to talk about **scope**

# Semantics: Scope of a Variable

---

- The part of a program where a **variable is visible**
- In the expression  $\lambda x \rightarrow e$ 
  - $x$  is the newly introduced variable
  - $e$  is the **scope** of  $x$
  - any **occurrence** of  $x$  in  $\lambda x \rightarrow e$  is **bound** (by the **binder**  $\lambda x$ )

# Semantics: Scope of a Variable

---

- For example,  $x$  is **bound** in:

```
\x -> x
\x -> (\y -> x)
```

- An occurrence of  $x$  in  $e$  is **free** if it's *not bound* by an enclosing abstraction
- For example,  $x$  is **free** in:

```
x y           -- no binders at all!
\y -> x y      -- no \x binder
(\x -> \y -> y) x -- x is outside the scope
                  -- of the \x binder;
                  -- intuition: it's not "the same" x
```

# Free Variables

---

- An variable  $x$  is **free** in  $e$  if there exists a free occurrence of  $x$  in  $e$
- We can formally define the set of all free variables in a term like so:

$FV(x) = ???$

$FV(\lambda x \rightarrow e) = ???$

$FV(e1 \ e2) = ???$

# Free Variables

---

- An variable  $x$  is **free** in  $e$  if there exists a free occurrence of  $x$  in  $e$
- We can formally define the set of all free variables in a term like so:

$$FV(x) = \{x\}$$

$$FV(\lambda x \rightarrow e) = FV(e) \setminus \{x\}$$

$$FV(e1 \ e2) = FV(e1) \cup FV(e2)$$

# Closed Expressions

---

- If  $e$  has no free variables it is said to be closed
- Closed expressions are also called **combinators**
  - Q: What is the *shortest* closed expression?



# Closed Expressions

---

- If  $e$  has no free variables it is said to be closed
- Closed expressions are also called **combinators**
  - **Q:** What is the *shortest* closed expression?
  - **A:**  $\lambda x. x$

# Rewrite rules of lambda calculus

---

1.  $\alpha$ -step (aka renaming formals)
2.  $\beta$ -step (aka function call)

# Semantics: $\beta$ -Reduction

---

$$(\lambda x \rightarrow e1) e2 \rightarrow_b e1[x := e2]$$

where  $e1[x := e2]$  means “ $e1$  with all free occurrences of  $x$  replaced with  $e2$ ”

- Computation by *search-and-replace*:
  - If you see an *abstraction* applied to an argument, take the *body* of the abstraction and replace all free occurrences of the *formal* by that argument
  - We say that  $(\lambda x \rightarrow e1) e2$   $\beta$ -steps to  $e1[x := e2]$

# Examples

---

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

Is this right? Ask [Elsa!](#)

```
(\f -> f (\x -> x)) (give apple)  
=b> ???
```

# Examples

---

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

Is this right? Ask [Elsa!](#)

```
(\f -> f (\x -> x)) (give apple)  
=b> give apple (\x -> x)
```

# A Tricky One

---

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

Is this right?

**Problem:** the free  $y$  in the argument has been *captured* by  $\backslash y$ !

**Solution:** make sure that all *free variables* of the argument are different from the *binders* in the body.

# Capture-Avoiding Substitution

---

- We have to fix our definition of  $\beta$ -reduction:

$$(\lambda x \rightarrow e1) e2 \rightarrow_b e1[x := e2]$$

where  $e1[x := e2]$  means ~~“ $e1$  with all free occurrences of  $x$  replaced with  $e2$ ”~~

- $e1$  with all *free* occurrences of  $x$  replaced with  $e2$ , **as long as** no free variables of  $e2$  get captured
- undefined otherwise

# Capture-Avoiding Substitution

---

Formally:

```
x[x := e]           = e
y[x := e]           = y      -- assuming x /= y
(e1 e2)[x := e]     = (e1[x := e]) (e2[x := e])
(\x -> e1)[x := e]   = \x -> e1 -- why just `e1`?

(\y -> e1)[x := e]
  | not (y in FV(e)) = \y -> e1[x := e]
  | otherwise       = undefined -- but what then???
```



# Rewrite rules of lambda calculus

---

1.  $\alpha$ -step (aka renaming formals)
2.  $\beta$ -step (aka function call)

# Semantics: $\alpha$ -Reduction

---

$\lambda x \rightarrow e \quad =_a \quad \lambda y \rightarrow e[x := y]$   
where not  $(y \text{ in } FV(e))$

- We can rename a formal parameter and replace all its occurrences in the body
- We say that  $(\lambda x \rightarrow e)$  *a-steps* to  $(\lambda y \rightarrow e[x := y])$

# Semantics: $\alpha$ -Reduction

---

$\lambda x \rightarrow e \quad =_{\alpha} \quad \lambda y \rightarrow e[x := y]$   
where not  $(y \text{ in } FV(e))$

- Example:

$\lambda x \rightarrow x \quad =_{\alpha} \quad \lambda y \rightarrow y \quad =_{\alpha} \quad \lambda z \rightarrow z$

- All these expressions are  $\alpha$ -equivalent

# Example

---

What's wrong with these?

-- (A)

$\backslash f \rightarrow f\ x \quad =a> \quad \backslash x \rightarrow x\ x$

-- (B)

$(\backslash x \rightarrow \backslash y \rightarrow y)\ y \quad =a> \quad (\backslash x \rightarrow \backslash z \rightarrow z)\ z$

-- (C)

$\backslash x \rightarrow \backslash y \rightarrow x\ y \quad =a> \quad \backslash \text{apple} \rightarrow \backslash \text{orange} \rightarrow \text{apple}\ \text{orange}$

# The Tricky One

---

```
(\x -> (\y -> x)) y  
=a> ???
```

To avoid getting confused, you can always rename formals, so that different variables have different names!

# The Tricky One

---

```
(\x -> (\y -> x)) y  
=a> (\x -> (\z -> x)) y  
=b> \z -> y
```

To avoid getting confused, you can always rename formals, so that different variables have different names!

# Normal Forms

---

A **redex** is a  $\lambda$ -term of the form

$$(\lambda x \rightarrow e1) e2$$

A  $\lambda$ -term is in **normal form** if it contains no redexes.

# Semantics: Evaluation

---

- A  $\lambda$ -term  $e$  evaluates to  $e'$  if

1. There is a sequence of steps

$$e \Rightarrow e_1 \Rightarrow \dots \Rightarrow e_N \Rightarrow e'$$

where each  $\Rightarrow$  is either  $\Rightarrow_a$  or  $\Rightarrow_b$  and  $N \geq 0$

2.  $e'$  is in *normal form*



# Example of evaluation

---

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

```
(\f -> f (\x -> x)) (\x -> x)  
=?> ???
```

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

# Example of evaluation

---

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

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

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

# Example of evaluation

---

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

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

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

# Elsa shortcuts

---

- Named  $\lambda$ -terms

```
let ID = \x -> x  -- abbreviation for \x -> x
```

- To substitute a name with its definition, use a `=d>` step:

```
ID apple
=d> (\x -> x) apple  -- expand definition
=b> apple           -- beta-reduce
```

# Elsa shortcuts

---

- Evaluation
  - $e1 \Rightarrow^* e2$ :  $e1$  reduces to  $e2$  in 0 or more steps
    - where each step is  $\Rightarrow a$ ,  $\Rightarrow b$ , or  $\Rightarrow d$
  - $e1 \Rightarrow^\sim e2$ :  $e1$  evaluates to  $e2$
- *What is the difference?*

# Non-Terminating Evaluation

---

$(\lambda x \rightarrow x x) (\lambda x \rightarrow x x)$   
 $=b> (\lambda x \rightarrow x x) (\lambda x \rightarrow x x)$

- Oh no... we can write programs that loop back to themselves
- And never reduce to normal form!
- This combinator is called  $\Omega$

# Non-Terminating Evaluation

---

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

```
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
- Let's see how to encode all of these features with the  $\lambda$ -calculus.



# λ-calculus: Booleans

---

- How can we encode Boolean values (TRUE and FALSE) as functions?
- Well, what do we **do** with a Boolean **b**?

- We make a *binary choice*

**if** b **then** e1 **else** e2

# 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 first argument
let FALSE = \x y -> y      -- Returns second argument
let ITE    = \b x y -> b x y -- Applies cond. 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
```

# Example: Branches step-by-step

---

```
eval ite_false:
```

```
  ITE FALSE e1 e2
```

```
=d> (\b x y -> b      x  y) FALSE e1 e2  -- expand def ITE
```

```
=b>   (\x y -> FALSE x  y)      e1 e2  -- beta-step
```

```
=b>     (\y -> FALSE e1 y)      e2  -- beta-step
```

```
=b>           FALSE e1 e2          -- expand def TRUE
```

```
=d>       (\x y -> y) e1 e2          -- beta-step
```

```
=b>         (\y -> y)      e2        -- beta-step
```

```
=b> e2
```

# Boolean operators

---

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

```
let NOT = \b      -> ???
```

```
let AND = \b1 b2 -> ???
```

```
let OR  = \b1 b2 -> ???
```

# Boolean operators

---

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

```
let NOT = \b      -> ITE b FALSE TRUE
```

```
let AND = \b1 b2 -> ITE b1 b2 FALSE
```

```
let OR  = \b1 b2 -> ITE b1 TRUE b2
```



# Boolean operators

---

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

```
let NOT = \b      -> b FALSE TRUE
```

```
let AND = \b1 b2 -> b1 b2 FALSE
```

```
let OR  = \b1 b2 -> b1 TRUE b2
```

- (since ITE is redundant)
- *Which definition to do you prefer and why?*

# Programming in $\lambda$ -calculus

---

- Real languages have lots of features
  - **Booleans** [done]
  - Records (structs, tuples)
  - Numbers
  - **Functions** [we got those]
  - Recursion

# $\lambda$ -calculus: Records

---

- Let's start with records with two fields (aka pairs)?
- Well, 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 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 1st value
let SND  = \p -> p FALSE   -- call w/ FALSE, get 2nd 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 -> ???
```

# Exercise: Triples?

---

- How can we implement a record that contains **three** values?

```
let TRIPLE = \x y z -> PAIR x (PAIR y z)
```

```
let FST3   = \t -> FST t
```

```
let SND3   = \t -> FST (SND t)
```

```
let TRD3   = \t -> SND (SND t)
```

# Programming in $\lambda$ -calculus

---

- Real languages have lots of features
  - **Booleans** [done]
  - **Records (structs, tuples)** [done]
  - Numbers
  - **Functions** [we got those]
  - Recursion



# $\lambda$ -calculus: Numbers

---

- Let's start with **natural numbers** (0, 1, 2, ...)
- What do we do with natural numbers?
  1. **Count**: 0, inc
  2. **Arithmetic**: dec, +, -, \*
  3. **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
...
```

# Pairs: 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)))))
...

```

# $\lambda$ -calculus: Increment

---

-- Call `f` on `x` one more time than `n` does

```
let INC    = \n -> (\f x -> ???)
```

- 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
```

# $\lambda$ -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

# $\lambda$ -calculus: Multiplication

---

-- Call `f` on `x` exactly `n \* m` times

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

- Example

```
eval two_times_one :
```

```
  MULT TWO ONE
```

```
=~> TWO
```

# Programming in $\lambda$ -calculus

---

- Real languages have lots of features
  - **Booleans** [done]
  - **Records (structs, tuples)** [done]
  - **Numbers** [done]
  - **Functions** [we got those]
  - **Recursion**

# $\lambda$ -calculus: Recursion

---

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

$\backslash n \rightarrow \dots$                        $-- 1 + 2 + \dots + n$



# $\lambda$ -calculus: Recursion

---

- No! Named terms in Elsa are just syntactic sugar
- To translate an Elsa term to  $\lambda$ -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  $\lambda$ -calculus functions are *anonymous*.
- *Right?*

# $\lambda$ -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
```

# $\lambda$ -calculus: Recursion

---

- 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)
```

# $\lambda$ -calculus: Fixpoint Combinator

---

```
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
```

# $\lambda$ -calculus: Fixpoint Combinator

---

- So how do we define **FIX**?
- Remember  $\Omega$ ? It *replicates itself*!

$(\lambda x. x x) (\lambda x. x x)$   
 $=_{\beta} (\lambda x. x x) (\lambda x. x x)$

- We need something similar but more involved.

# λ-calculus: Fixpoint Combinator

- 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:
```

```
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 ^^^^^^^^^^^^^
```

# Programming in $\lambda$ -calculus

---

- Real languages have lots of features
  - **Booleans** [done]
  - **Records (structs, tuples)** [done]
  - **Numbers** [done]
  - **Functions** [we got those]
  - **Recursion** [done]



# Next time: Intro to Haskell

---

