

CSE 116: Fall 2019

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

2

Your favorite language

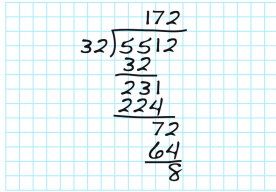
- Probably has lots of features:
 - Assignment ($x = x + 1$)
 - Booleans, integers, characters, strings,...
 - Conditionals
 - Loops, return, break, continue
 - Functions
 - Recursion
 - References / pointers
 - Objects and classes
 - Inheritance
 - ... and more

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

3

What is computable?

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

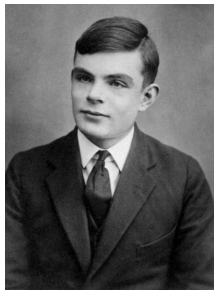


One that can be computed by a human with pen and paper, following an algorithm

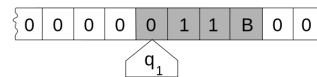
4

What is computable?

- 1936: Formalization



Alan Turing: Turing machines



5

What is computable?

- 1936: Formalization



Alonzo Church: lambda calculus

```
e ::= x
    | \x -> e
    | e1 e2
```

6

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

7

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

8

The Lambda Calculus

- Features
 - Functions
 - (that's it)

9

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

10

Describing a Programming Language

- Syntax

- What do programs *look like*?

- Semantics

- What do programs *mean*?
- Operational semantics:
 - How do programs *execute step-by-step*?

11

Syntax: What programs look like

```
e ::= x
    | \x -> e
    | e1 e2
```

- Programs are **expressions** e (also called λ -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*

12

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)
```

13

QUIZ: Lambda syntax

Which of the following terms are syntactically incorrect? *

- ☐ A. $\lambda x. \lambda x. x \rightarrow y$
- ☐ B. $\lambda x. x \rightarrow x x$
- ☐ C. $\lambda x. x \rightarrow x (y x)$
- ☐ A and C
- ☐ All of the above



<http://tiny.cc/cse116-lambda-ind>

14

QUIZ: Lambda syntax

Which of the following terms are syntactically incorrect? *

- ☐ A. $\lambda x. \lambda x. x \rightarrow y$
- ☐ B. $\lambda x. x \rightarrow x x$
- ☐ C. $\lambda x. x \rightarrow x (y x)$
- ☐ A and C
- ☐ All of the above



<http://tiny.cc/cse116-lambda-grp>

15

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

16

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

17

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)
```

18

Syntactic Sugar

- Convenient notation used as a shorthand for valid syntax

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

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

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

19

Semantics: What programs mean

- How do I “run” or “execute” a λ -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

20

Rewrite rules of lambda calculus

- α -step (aka renaming formals)
- β -step (aka function call)

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

21

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 λx)

22

Semantics: Scope of a Variable

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

```
 $\lambda x \rightarrow x$   
 $\lambda x \rightarrow (\lambda y \rightarrow x)$ 
```

- 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!  
 $\lambda y \rightarrow x\ y$     -- no  $\lambda x$  binder  
 $(\lambda x \rightarrow \lambda y \rightarrow y)\ x$  --  $x$  is outside the scope  
                  -- of the  $\lambda x$  binder;  
                  -- intuition: it's not "the same"  $x$ 
```

23

QUIZ: Variable scope

In the expression $(\lambda x \rightarrow x)\ x$, is x bound or free? *

- ☐ A. bound
- ☐ B. free
- ☐ C. first occurrence is bound, second is free
- ☐ D. first occurrence is bound, second and third are free
- ☐ E. first two occurrences are bound, third is free



<http://tiny.cc/cse116-scope-ind>

24

QUIZ: Variable scope

In the expression $(\lambda x \rightarrow x) x$, is x bound or free? *

- ☐ A. bound
- ☐ B. free
- ☐ C. first occurrence is bound, second is free
- ☐ D. first occurrence is bound, second and third are free
- ☐ E. first two occurrences are bound, third is free



<http://tiny.cc/cse116-scope-grp>

25

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:

$FV(x) = ???$
 $FV(\lambda x \rightarrow e) = ???$
 $FV(e1\ e2) = ???$

26

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:

$FV(x) = \{x\}$
 $FV(\lambda x \rightarrow e) = FV(e) \setminus \{x\}$
 $FV(e1\ e2) = FV(e1) \cup FV(e2)$

27

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?

28

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: $\lambda x \rightarrow x$

29

Rewrite rules of lambda calculus

1. α -step (aka renaming formals)
2. β -step (aka function call)

30

Semantics: β -Reduction

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

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 $(\lambda x \rightarrow e1) e2$ β -steps to $e1[x := e2]$

31

Examples

$(\lambda x \rightarrow x) \text{ apple}$
 $\Rightarrow \text{apple}$

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

$(\lambda f \rightarrow f (\lambda x \rightarrow x)) (\text{give apple})$
 $\Rightarrow ???$

32

Examples

$(\lambda x \rightarrow x) \text{ apple}$
 $\Rightarrow \text{apple}$

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

$(\lambda f \rightarrow f (\lambda x \rightarrow x)) (\text{give apple})$
 $\Rightarrow \text{give apple } (\lambda x \rightarrow x)$

33

QUIZ: β -Reduction 1

$(\lambda x \rightarrow (\lambda y \rightarrow y)) \text{ apple} = b \rightarrow ???$ *

- ☐ A. apple
- ☐ B. $\lambda y \rightarrow \text{apple}$
- ☐ C. $\lambda x \rightarrow \text{apple}$
- ☐ D. $\lambda y \rightarrow y$
- ☐ E. $\lambda x \rightarrow y$



<http://tiny.cc/cse116-beta1-ind>

34

QUIZ: β -Reduction 1

$(\lambda x \rightarrow (\lambda y \rightarrow y)) \text{ apple} = b \rightarrow ???$ *

- ☐ A. apple
- ☐ B. $\lambda y \rightarrow \text{apple}$
- ☐ C. $\lambda x \rightarrow \text{apple}$
- ☐ D. $\lambda y \rightarrow y$
- ☐ E. $\lambda x \rightarrow y$



<http://tiny.cc/cse116-beta1-grp>

35

QUIZ: β -Reduction 2

$(\lambda x \rightarrow x (\lambda x \rightarrow x)) \text{ apple} = b \rightarrow ???$ *

- ☐ A. $\text{apple } (\lambda x \rightarrow x)$
- ☐ B. $\text{apple } (\lambda \text{apple} \rightarrow \text{apple})$
- ☐ C. $\text{apple } (\lambda x \rightarrow \text{apple})$
- ☐ D. apple
- ☐ E. $\lambda x \rightarrow x$



<http://tiny.cc/cse116-beta2-ind>

36

QUIZ: β -Reduction 2

$(\lambda x \rightarrow x (\lambda x \rightarrow x)) \text{ apple} = b \rightarrow ???$ *

- ☐ A. $\text{apple } (\lambda x \rightarrow x)$
- ☐ B. $\text{apple } (\lambda \text{apple} \rightarrow \text{apple})$
- ☐ C. $\text{apple } (\lambda x \rightarrow \text{apple})$
- ☐ D. apple
- ☐ E. $\lambda x \rightarrow x$



<http://tiny.cc/cse116-beta2-grp>

37

A Tricky One

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

Is this right?

Problem: the free y in the argument has been *captured* by λy !

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

38

Capture-Avoiding Substitution

- We have to fix our definition of β -reduction:

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

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

39

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???
```

40

Rewrite rules of lambda calculus

1. α -step (aka renaming formals)
2. β -step (aka function call)

41

Semantics: α -Reduction

```
\x -> e  =>  \y -> e[x := y]
where not (y in FV(e))
```

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

42

Semantics: α -Reduction

```
\x -> e    =a>   \y -> e[x := y]  
  where not (y in FV(e))
```

- Example:

```
\x -> x    =a>   \y -> y    =a>   \z -> z
```

- All these expressions are α -equivalent

43

Example

What's wrong with these?

-- (A)

```
\f -> f x    =a>   \x -> x x
```

-- (B)

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

-- (C)

```
\x -> \y -> x y    =a>   \apple -> \orange -> apple orange
```

44

The Tricky One

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

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

45

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!

46

Normal Forms

A **redex** is a λ -term of the form

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

A λ -term is in **normal form** if it contains no redexes.

47

QUIZ: Normal form

Which of the following terms are not in normal form ? *

- ☐ A. x
- ☐ B. x y
- ☐ C. $(\lambda x. x) y$
- ☐ D. $x (\lambda y. y)$
- ☐ E. C and D



<http://tiny.cc/cse116-norm-ind>

48

QUIZ: Normal form

Which of the following terms are not in normal form ? *

- ☐ A. x
- ☐ B. xy
- ☐ C. $(\lambda x \rightarrow x) y$
- ☐ D. $x (\lambda y \rightarrow y)$
- ☐ E. C and D



<http://tiny.cc/cse116-norm-grp>

49

Semantics: Evaluation

- A λ -term e evaluates to e' if

1. There is a sequence of stops

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

where each \Rightarrow is either $=a>$ or $=b>$ and $N \geq 0$

2. e' is in *normal form*

50

Example of evaluation

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

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

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

51

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)
=?> ???
```

52

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
```

53

Elsa shortcuts

- Named λ -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
```

54

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
 - $e1 \Rightarrow e2$: $e1$ evaluates to $e2$
- *What is the difference?*

55

Non-Terminating Evaluation

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

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

56

Non-Terminating Evaluation

- What if we pass Ω 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!

57

Programming in λ -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 λ -calculus.

58

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`

59

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`)

60

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)
```

61

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
```

62

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
```

63

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
```

64

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 -> ???
```

65

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
```

66

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?*

67

Programming in λ -calculus

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

68

λ -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.

69

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
```

70

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
```

71

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 -> ???
```

72

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)
```

73

Programming in λ -calculus

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

74

λ -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

75

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
...
```

76

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

77

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



<http://tiny.cc/cse116-church-ind>

78

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



<http://tiny.cc/cse116-church-grp>

79

λ -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
```

80

QUIZ: ADD

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)



<http://tiny.cc/cse116-add-ind>

81

QUIZ: ADD

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)



<http://tiny.cc/cse116-add-grp>

82

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
```

83

QUIZ: MULT

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



<http://tiny.cc/cse116-mult-ind>

84

QUIZ: MULT

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



<http://tiny.cc/cse116-mult-grp>

85

λ -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
```

86

Programming in λ -calculus

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

87

λ -calculus: Recursion

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

$\lambda n \rightarrow \dots \quad \text{-- } 1 + 2 + \dots + n$

88

QUIZ: SUM

Is this a correct implementation of SUM? *

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



- ☐ A. Yes
- ☐ B. No

<http://tiny.cc/cse116-sum-ind>

89

QUIZ: SUM

Is this a correct implementation of SUM? *

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



- ☐ A. Yes
- ☐ B. No

<http://tiny.cc/cse116-sum-grp>

90

λ-calculus: Recursion

- 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?*

91

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
```

92

λ-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)))
```

93

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)
```

94

λ-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
```

95

λ-calculus: Fixpoint Combinator

- So how do we define **FIX**?
- Remember Ω ? It *replicates itself*!

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

- We need something similar but more involved.

96

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

97

Programming in λ -calculus

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

98

Next time: Intro to Haskell



99