Quiz

a ✓ b ✗ c ✗

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

e_1 e_2

$e_1[x := e_2]$

A. `(apple (\x -> x))`

B. `(apple (\apple -> apple))`

C. `(apple (\x -> apple))`

D. `apple`

E. `(\x -> x)`

(\x -> (x (\x -> x)))

def (binder)

```
func (x) {
  return x+1
}
```

use (occurrence)

$(1+2)+3$
$= 3+3$
$= 6$

$(\backslash x \to \boxed{e_1})\ e_2$

EXERCISE

What is a λ-term `fill_this_in` such that

```
fill_this_in apple
=b> banana
```

(??? apple)
=b> banana

ELSA: https://goto.ucsd.edu/elsa/index.html

Click here to try this exercise (https://goto.ucsd.edu/elsa/index.html#?demo=permalink%2F1585434473_24432.lc)

(\x-> ?) apple =b> banana x

? [x := apple] ≡ banana

banana

A Tricky One

$ make

"downloading GHC"

(1) Make sure your $PATH is set (cs131w)

(2) rm -rf ~/.stack

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

=b> \y -> y
```

Is this right?

SYNTAX

e ::= x, y, z

(\x -> e) formal body

(e1 e2) "arg" "function"

$\boxed{(\lambda x \to e_1) \quad e_2}$ REDEX

formal body arg

((\x -> (z x)) apple)

=b> (apple apple)

$e_1 [x := e_2]$

Something is Fishy

λy → ...

formal body arg

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

body

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

Is this right?

Problem: The *free* `y` in the argument has been **captured** by `\y` in *body*!

Solution: Ensure that *formals* in the body are **different from** *free-variables* of argument!

Capture-Avoiding Substitution

We have to fix our definition of β-reduction:

```
(\x -> e1) e2   =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

Formally:

```
x[x := e]            = e

y[x := e]            = y                 -- as x /= y

(e1 e2)[x := e]      = (e1[x := e]) (e2[x := e])

(\x -> e1)[x := e]   = (\x -> e1)        -- Q: Why leave `e1` unch
anged?

(\y -> e1)[x := e]
  | not (y in FV(e)) = \y -> e1[x := e]
```

Oops, but what to do if `y` is in the *free-variables* of `e`?

- i.e. if `\y -> ...` may *capture* those free variables?

Rewrite Rules of Lambda Calculus

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

```
function (x) {
  return x+1
}
```

```
function(y){
  return y+1
}
```

```
function (z){
  return z+1;
}
```

Semantics: α*-Renaming*

$\lambda x \to y \quad \neq_{\alpha} \quad \lambda y \to y$

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

λx→x =a> λy→y =a> λz→z

- We rename a formal parameter `x` to `y`
- By replace all occurrences of `x` in the body with `y`
- We say that `\x -> e` α-steps to `\y -> e[x := y]`

Example:

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

All these expressions are **α-equivalent**

What's wrong with these?

```
-- (A)
(\f -> (f x))     =a>    (\x -> (x x))

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

outside scope of binder

Tricky Example Revisited

```
((\x -> (\y -> x)) y)
                                     -- rename 'y' to 'z' to avoid capt
ure
=a> ((\x -> (\z -> x)) y)
                                     -- now do b-step without capture!
=b> (\z -> y)
```

To avoid getting confused,

- you can **always rename** formals,
- so different **variables** have different **names**!

Normal Forms

Recall **redex** is a λ-term of the form

```
((\x -> e1) e2)
```

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

$(\lambda x \to y)$

QUIZ

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

ie contain further b-steps?
b-redexes?

A. `x` no-redex

B. `(x y)` no-redex

✓ **C.** `((\x -> x) y)` yes-redex $\xrightarrow{b}$ y ✓

$$((\lambda x \to e_1)\ e_2)$$

(left func) (right arg)

D. `(x (\y -> y))` no-redex

E. C and D

Semantics: Evaluation

(2+2) - 3

=

4 - 3

=

1 (normal)

A λ-term `e` **evaluates to** `e'` if

1. There is a sequence of steps

```
e =?> e_1 =?> ... =?> e_N =?> e'
```

where each `=?>` is either `=a>` or `=b>` and `N >= 0`

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

Examples of Evaluation

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

  =b> apple
```

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

  =?> ???
```

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

Elsa shortcuts

Named λ-terms:

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

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

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

Evaluation:

- `e1 =*> e2` : `e1` reduces to `e2` in 0 or more steps
  - where each step is `=a>` , `=b>` , or `=d>`
- `e1 =~> e2` : `e1` evaluates to `e2` and `e2` is **in normal form**

$(\lambda x \to x\ x)\ (\lambda y \to y\ y)$

$=b> (\lambda y \to y\ y)\ (\lambda y \to y\ y)$

EXERCISE

Fill in the definitions of `FIRST` , `SECOND` and `THIRD` such that you get the following behavior in `elsa`

```
let FIRST  = fill_this_in
let SECOND = fill_this_in
let THIRD  = fill_this_in

eval ex1 :
  FIRST apple banana orange
  =*> apple

eval ex2 :
  SECOND apple banana orange
  =*> banana

eval ex3 :
  THIRD apple banana orange
  =*> orange
```

(\x -> x)

(((\x -> x) apple) ban) orang)

(\x1 -> (\x2 -> (\x3 -> x1)))

apple

ban

orange

ELSA: https://goto.ucsd.edu/elsa/index.html

Click here to try this exercise (https://goto.ucsd.edu/elsa/index.html#?demo=permalink%2F1585434130_24421.lc)

Non-Terminating Evaluation

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

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

Some programs loop back to themselves … *never* reduce to a normal form!

This combinator is called Ω

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!

Programming in λ-calculus

Real languages have lots of features

- Booleans → T, F, if-then-else, &&, ||
- Records (structs, tuples) → { fst : _ , snd: _ } { x:_ , y:_ , z:_ }
- Numbers → 2 + 5   lists, trees
- **Functions** [we got those]
- Recursion

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

Syntactic Sugar

instead of	**we write**
`\x -> (\y -> (\z -> e))`	`\x -> \y -> \z -> e`
`\x -> \y -> \z -> e`	`\x y z -> e`
`(((e1 e2) e3) e4)`	`e1 e2 e3 e4`

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

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

bool/cond ?

TRUE $\swarrow$ $\searrow$ FALSE

res_true res_false

λ-calculus: Booleans

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

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

decisions / condition / choice

Make a *binary choice*

- `if b then e1 else e2`

$b \; ? \; e_1 : e_2$

Booleans: API

We need to define three functions

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

(b x 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 readabili
ty)
```

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)

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

```
let NOT = \b     -> ???
let OR  = \b1 b2 -> ???
let AND = \b1 b2 -> ???
```

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
```

AND TT FF ~> FF
AND FF FF ~> FF
AND FF TT ~> FF
AND TT TT ~> TT

Programming in λ-calculus

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

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
```

Programming in λ-calculus

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

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 -> ???)
```

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
```

EXERCISE

Fill in the implementation of `ADD` so that you get the following behavior

Click here to try this exercise (https://goto.ucsd.edu/elsa/index.html#?demo=permalink%2F1585436042_24449.lc)

```
let ZERO = \f x -> x
let ONE  = \f x -> f x
let TWO  = \f x -> f (f x)
let INC  = \n f x -> f (n f x)

let ADD  = fill_this_in

eval add_zero_zero:
  ADD ZERO ZERO =~> ZERO

eval add_zero_one:
  ADD ZERO ONE =~> ONE

eval add_zero_two:
  ADD ZERO TWO =~> TWO

eval add_one_zero:
  ADD ONE ZERO =~> ONE

eval add_one_zero:
  ADD ONE ONE =~> TWO

eval add_two_zero:
  ADD TWO ZERO =~> TWO
```

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)`

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]
- **Lists**
- **Functions** [we got those]
- Recursion

λ-calculus: Lists

Lets define an API to build lists in the λ-calculus.

An Empty List

```
NIL
```

Constructing a list

A list with 4 elements

```
CONS apple (CONS banana (CONS cantaloupe (CONS dragon NIL)))
```

intuitively `CONS h t` creates a *new* list with

- *head* `h`
- *tail* `t`

Destructing a list

- `HEAD l` returns the *first* element of the list
- `TAIL l` returns the *rest* of the list

```
HEAD (CONS apple (CONS banana (CONS cantaloupe (CONS dragon NI
L))))
=~> apple

TAIL (CONS apple (CONS banana (CONS cantaloupe (CONS dragon NI
L))))
=~> CONS banana (CONS cantaloupe (CONS dragon NIL)))
```

λ-calculus: Lists

```
let NIL  = ???
let CONS = ???
let HEAD = ???
let TAIL = ???

eval exHd:
  HEAD (CONS apple (CONS banana (CONS cantaloupe (CONS dragon NI
L))))
  =~> apple

eval exTl
  TAIL (CONS apple (CONS banana (CONS cantaloupe (CONS dragon NI
L))))
  =~> CONS banana (CONS cantaloupe (CONS dragon NIL)))
```

EXERCISE: Nth

Write an implementation of `GetNth` such that

- `GetNth n l` returns the `n`-th element of the list `l`

Assume that `l` has n or more elements

```
let GetNth = ???

eval nth1 :
  GetNth ZERO (CONS apple (CONS banana (CONS cantaloupe NIL)))
  =~> apple

eval nth1 :
  GetNth ONE (CONS apple (CONS banana (CONS cantaloupe NIL)))
  =~> banana

eval nth2 :
  GetNth TWO (CONS apple (CONS banana (CONS cantaloupe NIL)))
  =~> cantaloupe
```

Click here to try this in elsa (https://goto.ucsd.edu/elsa/index.html#?demo=permalink%2F1586466816_52273.lc)

λ-calculus: Recursion

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

```
let SUM = \n -> ...  -- 0 + 1 + 2 + ... + n
```

such that we get the following behavior

```
eval exSum0: SUM ZERO  =~> ZERO
eval exSum1: SUM ONE   =~> ONE
eval exSum2: SUM TWO   =~> THREE
eval exSum3: SUM THREE =~> SIX
```

Can we write sum **using Church Numerals**?

Click here to try this in Elsa (https://goto.ucsd.edu/elsa/index.html#?demo=permalink%2F1586465192_52175.lc)

QUIZ

You *can* write `SUM` using numerals but its *tedious*.

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 yet defined!
```

Recursion:

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

Looks like we can't do recursion!

- Requires being able to refer to functions *by name*,
- But λ-calculus functions are *anonymous*.

Right?

λ-calculus: Recursion

Think again!

Recursion:

Instead of

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

Lets try

- Inside *this* function I want to call *some* function `rec` on `DEC n`
- And BTW, I want `rec` 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 some magic to `STEP`, so `rec` is itself

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

That is, obtain a term `MAGIC` such that

```
MAGIC =*> STEP MAGIC
```

λ-calculus: Fixpoint Combinator

Wanted: a λ-term `FIX` such that

- `FIX STEP` calls `STEP` with `FIX STEP` 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   =*>   FIX STEP  =*>   STEP (FIX STEP)   =*>   STEP SUM
```

and so now we compute:

```
eval sum_two:
  SUM TWO
  =*> STEP SUM TWO
  =*> ITE (ISZ TWO) ZERO (ADD TWO (SUM (DEC TWO)))
  =*> ADD TWO (SUM (DEC TWO))
  =*> ADD TWO (SUM ONE)
  =*> ADD TWO (STEP SUM ONE)
  =*> ADD TWO (ITE (ISZ ONE) ZERO (ADD ONE (SUM (DEC ONE))))
  =*> ADD TWO (ADD ONE (SUM (DEC ONE)))
  =*> ADD TWO (ADD ONE (SUM ZERO))
  =*> ADD TWO (ADD ONE (ITE (ISZ ZERO) ZERO (ADD ZERO (SUM DEC ZER
O)))
  =*> ADD TWO (ADD ONE (ZERO))
  =*> THREE
```

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))
```

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

That's all folks, Haskell Curry was very clever.

Next week: We'll look at the language named after him (`Haskell`)

(https://ucsd-cse130.github.io/wi21/feed.xml) (https://twitter.com/ranjitjhala)
(https://plus.google.com/u/0/104385825850161331469)
(https://github.com/ranjitjhala)