Nano: Functions

Let's add

- *lambda abstraction* (aka function definitions)
- *application* (aka function calls)

Fun-Def
\x -> e
formal body

Fun-Call
(e1 e2)
func arg

```
e ::= n                          -- OLD
    | e1 `op` e2
    | x
    | let x = e1 in e2
                                 -- NEW
    | \x -> e                    -- abstraction
    | e1 e2                      -- application
```

Example

```
let incr = \x -> x + 1
in
  incr 10
```

ELet "incr"
(ELam "x"
(EBin (EVar "x") ...
)
(EApp (EVar "incr")
(EInt 10)
)

Representation

```
data Expr
  = ENum Int                -- OLD
  | EBin Binop Expr Expr
  | EVar Id
  | ELet Id Expr Expr
                           -- NEW
  | ???                    -- abstraction \x -> e
  | ???                    -- application (e1 e2)
```

Representation

```
data Expr
  = ENum Int                -- OLD
  | EBin Binop Expr Expr
  | EVar Id
  | ELet Id Expr Expr
                           -- NEW
  | ELam Id Expr           -- abstraction \x -> e
  | EApp Expr Expr         -- application (e1 e2)
```

Example

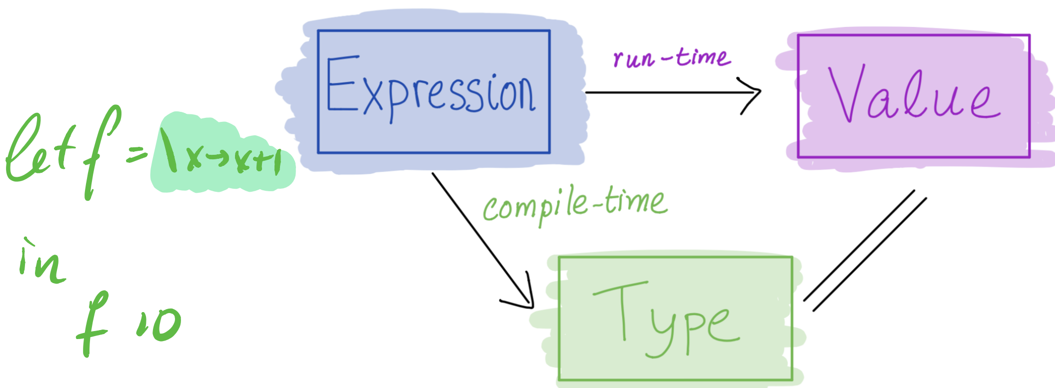
```
let incr = \x -> x + 1
in
   incr 10
```

is represented as

```
ELet "incr" (ELam "x" (EBin Add (EVar "x") (ENum 1)))
  (
    EApp (EVar "incr") (ENum 10)
  )
```

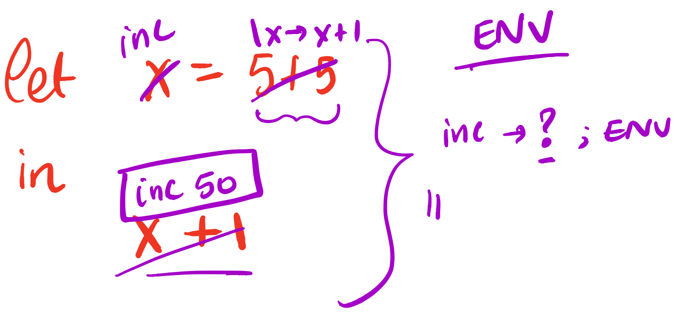
Functions are Values

Recall the trinity



But… what is the *value* of a function?

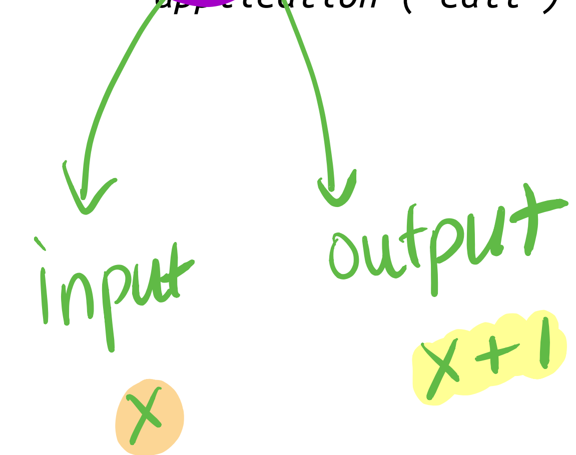
Lets build some intuition with examples.



QUIZ

What does the following expression evaluate to?

```
let incr = \x -> x + 1       -- abstraction ("definition")
in
   incr 10                   -- application ("call")
```



(A) Error/Undefined

(B) `10`

(C) `11`

(D) `0`

(E) `1`

What is the Value of `incr`*?*

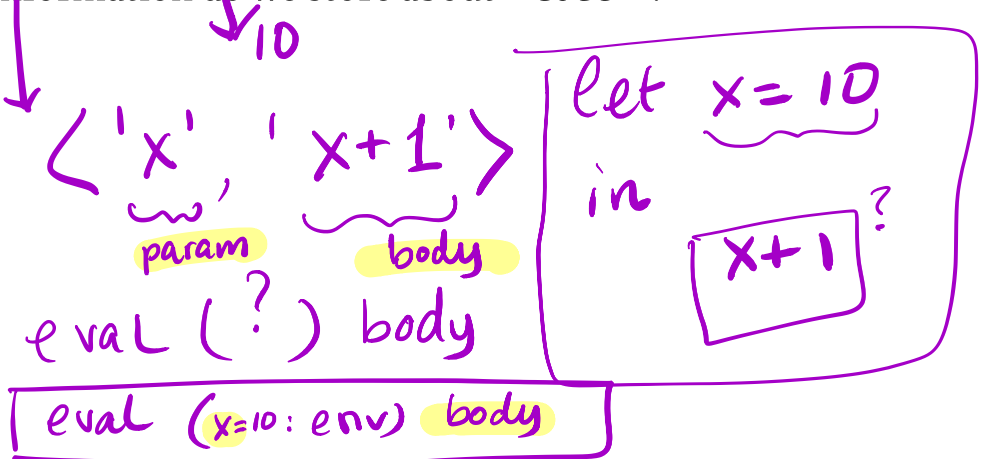
- Is it an `Int` ?
- Is it a `Bool` ?
- Is it a ???

What information do we need to store (in the `Env`) about `incr` ?

A Function's Value is its Code

```
                          -- env
let incr = \x -> x + 1
in                        -- ("incr" := <code>) : env
    incr 10               -- evaluate <code> with parameter := 10
```

What information do we store about `<code>` ?



A Call's Value

How to evaluate the "call" `incr 10` ?

1. Lookup the `<code>` i.e. `<param, body>` for `incr` (stored in the environment),

2. Evaluate `body` with `param` set to `10`!

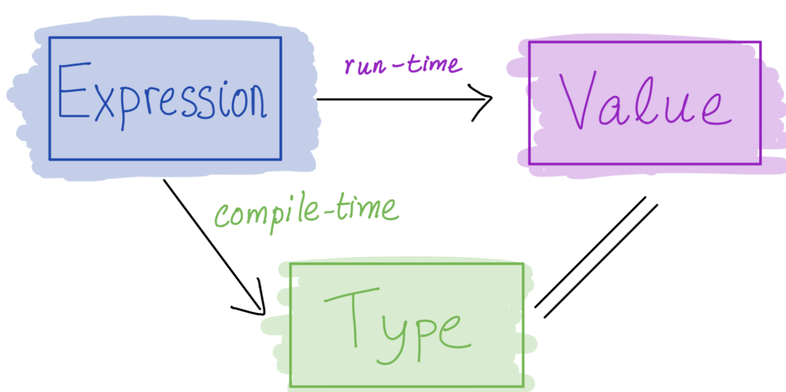
Two kinds of Values

We now have *two* kinds of Values

```
v ::= n                    -- OLD
    | <x, e>               -- <param, body>
```

1. Plain `Int` (as before)
2. A function's "code": a pair of "parameter" and "body-expression"

```haskell
data Value
  = VInt  Int              -- OLD
  | VCode Id Expr          -- <x, e>
```



Evaluating Lambdas and Applications

```
eval :: Env -> Expr -> Value
                                        -- OLD
eval env (ENum n)          = ???
eval env (EVar x)          = ???
eval env (EBin op e1 e2) = ???
eval env (ELet x  e1 e2) = ???
                                        -- NEW
eval env (ELam x e)        = ???
eval env (EApp e1 e2)      = ???
```

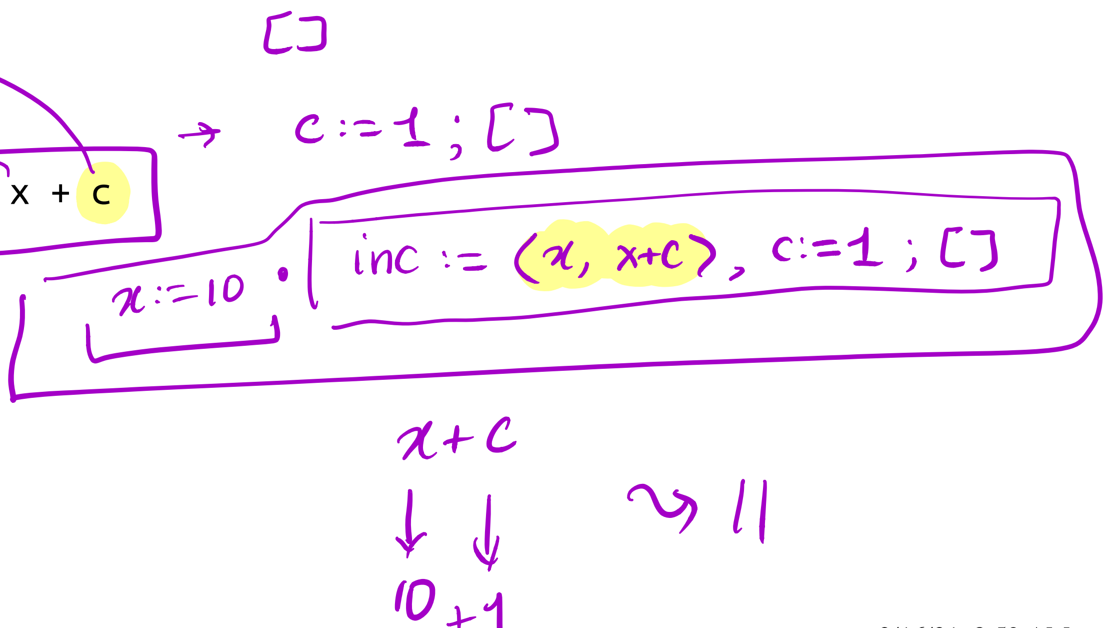
Lets make sure our tests work properly!

```
exLam1 = ELet "incr" (ELam "x" (EBin Add (EVar "x") (ENum 1)))
          (
            EApp (EVar "incr") (ENum 10)
          )


-- >>> eval [] exLam1
-- 11
```

QUIZ

What should the following evaluate to?

```
let c = 1
in
   let inc = \x -> x + c
   in
      inc 10
```



(A) Error/Undefined

(B) `10`

(C) `11`

(D) `0`

(E) `1`

```
exLam2 = ELet "c" (ENum 1)
           (ELet "incr" (ELam "x" (EBin Add (EVar "x") (EVar "
c")))
              (
              EApp (EVar "incr") (ENum 10)
              )
           )

-- >>> eval [] exLam2
-- ???
```

QUIZ

And what should *this* expression evaluate to?

```
let c = 1
in
   let inc = \x -> x + c
   in
      let c = 100
      in
         inc 10
```

(A) Error/Undefined

(B) `110`

(C) `11`

The "Immutability Principle"

A function's behavior should *never change*

- A function must *always* return the same output for a given input

Why?

```
> myFunc 10
0


> myFunc 10
10
```

Oh no! How to find the bug? Is it

- In `myFunc` or
- In a global variable or
- In a library somewhere else or
- …

My worst debugging nightmare

Colbert “Immutability Principle” (https://youtu.be/CWqzLgDc030?t=628)

The Immutability Principle ?

How does our `eval` work?

```
exLam3 = ELet "c" (ENum 1)
           (
           ELet "incr" (ELam "x" (EBin Add (EVar "x") (EVar "c")))
              (
               ELet "c" (ENum 100)
                 (
                 EApp (EVar "incr") (ENum 10)
                 )
              )
           )

-- >>> eval [] exLam3
-- ???
```

Oops?

```
                              -- []
let c = 1
in                            -- ["c" := 1]
   let inc = \x -> x + c
   in                         -- ["inc" := <x, x+c>, c := 1]
      let c = 100
      in                      -- ["c" := 100, "inc" := <x, x+c", "c" :
= 1]   <<< env
        inc 10
```

And so we get

```
eval env (inc 10)

  ==> eval ("x" := 10 : env) (x + c)

  ==> 10 + 100

  ==> 110
```

Ouch.

Enforcing Immutability with Closures

How to enforce immutability principle

- `inc 10` **always** returns `11`?

Key Idea: Closures

At definition: *Freeze* the environment the function's value

At call: Use the *frozen* environment to evaluate the *body*

Ensures that `inc 10` *always* evaluates to the *same* result!

```
                                -- []
let c = 1
in                              -- ["c" := 1]
   let inc = \x -> x + c
   in                           -- ["inc" := <frozenv, x, x+c>, c := 1]
<<< frozenv = ["c" := 1]
      let c = 100
      in                        -- ["c" := 100, "inc" := <frozenv, x, x+
c>, "c" := 1]
        inc 10
```

Now we evaluate

```
eval env (inc 10)

  ==> eval ("x" := 10 : frozenv) (x + c)  where frozenv = ["c" :=
1]

  ==> 10 + 1

  ==> 1
```

tada!

Representing Closures

Lets change the `Value` datatype to also store an `Env`

```haskell
data Value
  = VInt  Int           -- OLD
  | VClos Env Id Expr   -- <frozenv, param, body>
```

Evaluating Function Definitions

How should we fix the definition of `eval` for `ELam` ?

```haskell
eval :: Env -> Expr -> Value

eval env (ELam x e) = ???
```

Hint: What value should we *bind* `incr` to in our example above?

(Recall **At definition** *freeze* the environment the function's value)

Evaluating Function Calls

How should we fix the definition of `eval` for `EApp` ?

```haskell
eval :: Env -> Expr -> Value

eval env (EApp e1 e2) = ???
```

(Recall **At call:** Use the *frozen* environment to evaluate the *body*)

Hint: What value should we *evaluate* `incr 10` to?

1. Evaluate `incr` to get `<frozenv, "x", x + c>`
2. Evaluate `10` to get `10`
3. Evaluate `x + c` in `x:=10 : frozenv`

Let's generalize that recipe!

1. Evaluate `e1` to get `<frozenv, param, body>`
2. Evaluate `e2` to get `v2`
3. Evaluate `body` in `param := v2 : frozenv`

Immutability Achieved

Lets put our code to the test!

```
exLam3 = ELet "c" (ENum 1)
           (
           ELet "incr" (ELam "x" (EBin Add (EVar "x") (EVar "c")))
              (
               ELet "c" (ENum 100)
                 (
                 EApp (EVar "incr") (ENum 10)
                 )
              )
           )

-- >>> eval [] exLam3
-- ???
```

QUIZ

What should the following evaluate to?

```
let add = \x -> (\y -> x + y)
in
  let add10 = add 10
  in
    let add20 = add 20
    in
      (add10 100) + (add20 1000)
```

TODO

Functions Returning Functions Achieved!

```
exLam4 = ...

-- >>> eval [] exLam4
```

TODO

Practice

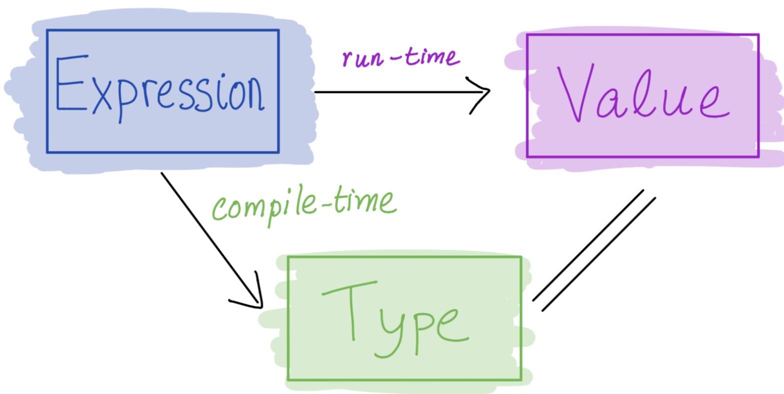
What should the following evaluate to?

```
let add = \x -> (\y -> x + y)
in
  let add10 = add 10
  in
    let doTwice = \f -> (\x -> f (f x))
    in
      doTwice add10 100
```

Functions Accepting Functions Achieved!

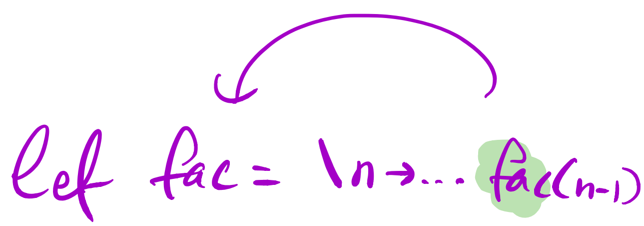
```
exLam5 = ...

-- >>> eval [] exLam4
```

The Nano Language



Features of Nano:

1. Arithmetic expressions *[done]*
2. Variables *[done]*
3. Let-bindings *[done]*
4. Functions *[done]*
5. **Recursion**



... You figure it out **Hw4** ... :-)

(https://ucsd-cse130.github.io/wi21/feed.xml) (https://twitter.com/ranjitjhala)

(https://plus.google.com/u/0/104385825850161331469)
(https://github.com/ranjitjhala)