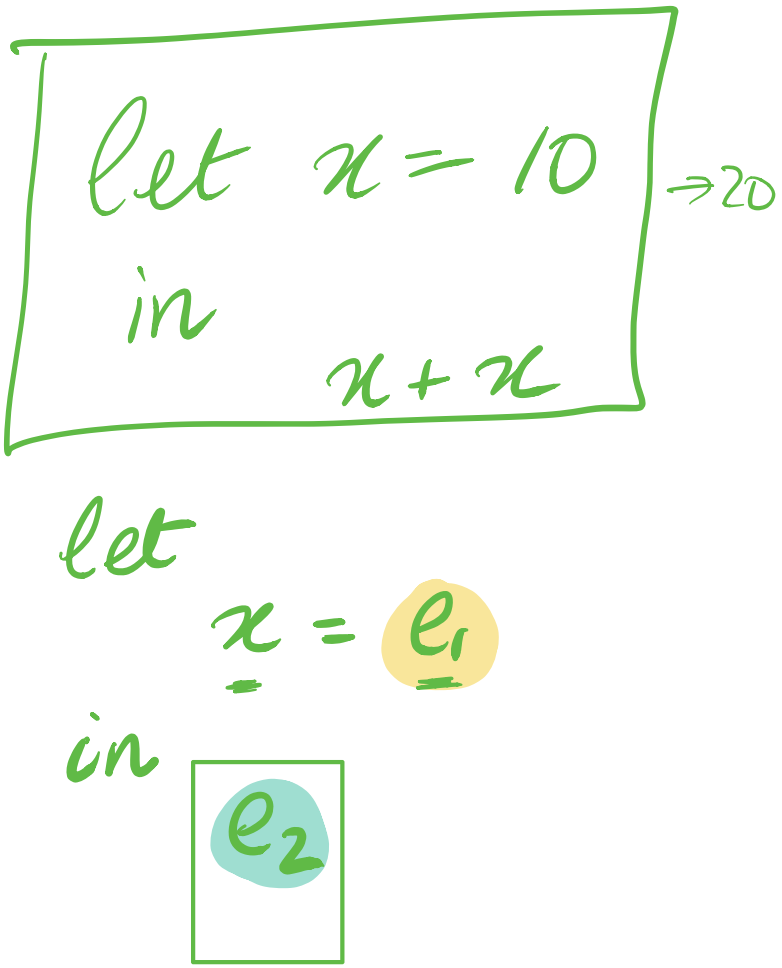
2. Nano: Variables

Let's add variables and **`let`** bindings!

```
e ::= n                          -- OLD
    | e1 + e2
    | e1 - e2
    | e1 * e2
    | x

                                 -- NEW
    | let x = e1 in e2
```



Lets extend our datatype

```
type Id = String

data Expr
  = ENum Int                     -- OLD
  | EBin Binop Expr Expr
  | EVar Id

                                 -- NEW
  | ELet Id Expr Expr
```

How should we extend `eval` ?

QUIZ

What *should* the following expression evaluate to?

```
let x = 0
in
  x + 1
```

ELet "x" (EInt 0)
(EAdd (EVar "x") (EInt 1))

(A) Error VUndef

(B) 1 VInt 1

(C) 0 VInt 0

QUIZ

What *should* the following expression evaluate to?

```
let x = 0
in
  let y = 100
  in
    x + y
```

let x = e1
y = e2
in e3

let x = 0
y = x+100
in

(A) Error

(B) `0`

(C) `1`

(D) `100`

(E) `101`

QUIZ

What *should* the following expression evaluate to?

```
let x = 0
in
  let x = 100
  in
    x + 1
```

x should get most recent def

(A) Error

(B) `0`

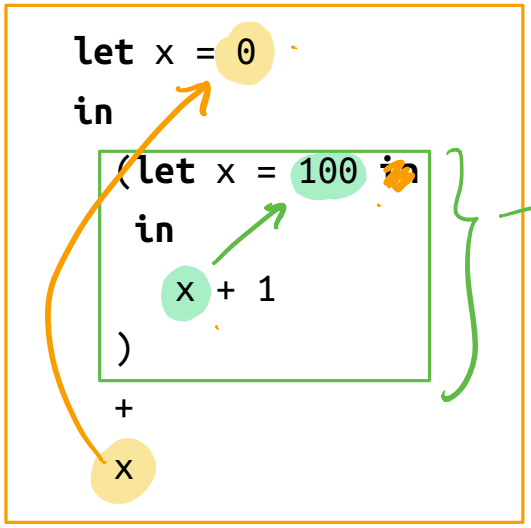
(C) `1`

(D) `100`

(E) `101`

QUIZ

What *should* the following expression evaluate to?



let x = 0 in
(let y = 100 in
y + 1)
+ x

(A) Error

(B) `1`

(C) `101` ✓

(D) `102`

(E) `2`

Principle: Static/Lexical Scoping

Every variable *use* gets its value from a unique *definition*:

- "Nearest" **`let`** -binder in program *text*

"Static" means you can tell *without running the program*

Great for readability and debugging

1. Define *local* variables
2. Be sure *where* each variable got its value

Don't have to scratch head to figure where a variable got "assigned"

How to **implement** static scoping?

QUIZ

Lets re-evaluate the quizzes!

```
let x = 0
in
  x + 1
```

```
-- env

-- ??? what env to use for `x + 1`?
```

(A) env

(B) [ ]

✗ (C) [("x" := 0)]

✓ (D) ("x" := 0) : env

✗ (E) env ++ ["x" := 0]

QUIZ

```
let x = 0
in
  let y = 100
  in
    x + y
```

```
<-- env

-- (x := 0) : env

<-- ??? what env to use for `x + y` ?
```

✗ (A) ("x" := 0) : env

✗ (B) ("y" := 100) : env

(C) `("y" := 100) : ("x" := 0) : env`

(D) `("x" := 0) : ("y" := 100) : env`

(E) `[("y" := 100), ("x" := 0)]` oops forgot env

QUIZ

Lets re-evaluate the quizzes!

```
                -- env
let x = 0
in                  -- ("x" := 0) : env
  let x = 100
  in                  -- ??? what env to use for `x + 1`?
    x + 1
```

(A) `("x" := 0) : env`

(B) `("x" := 100) : env`

(C) `("x" := 100) : ("x" := 0) : env`

(D) `("x" := 0) : ("x" := 100) : env`

(E) `[("x" := 100)]`

Extending Environments

Lets fill in `eval` for the **`let`** `x = e1` **`in`** `e2` case!

```
eval env (ELet x e1 e2) = ???
```

1. **Evaluate** `e1` in `env` to get a value `v1`
2. **Extend** environment with value for `x` i.e. to `(x := v1) : env`
3. **Evaluate** `e2` using *extended* environment.

Lets make sure our tests pass!

Run-time Errors

Haskell function to *evaluate* an expression:

```
eval :: Env -> Expr -> Value
eval env (Num n)        = n
eval env (Var x)        = lookup x env      -- (A)
eval env (Bin op e1 e2) = evalOp op v1 v2   -- (B)
  where
    v1                  = eval env  e1      -- (C)
    v2                  = eval env  e2      -- (C)
eval env (Let x e1 e2)  = eval env1 e2
  where
    v1                  = eval env e1
    env1                = (x, v1) : env     -- (D)
```

QUIZ

Will `eval env expr` always return a `value` ? Or, can it *crash*?

(A) operation at `A` may fail **(B)** operation at `B` may fail **(C)** operation at `C` may fail **(D)** operation at `D` may fail **(E)** nah, its all good..., always returns a `Value`

Free vs bound variables

Undefined Variables

How do we make sure `lookup` doesn't cause a run-time error?

Bound Variables

Consider an expression `let x = e1 in e2`

- An occurrence of `x` is **bound** in `e2`
- i.e. when occurrence of form `let x = ... in ... x ...`
- i.e. when `x` occurs "under" a `let` binding for `x`.

Free Variables

An occurrence of `x` is **free** in `e` if it is **not bound** in `e`

Closed Expressions

An expression `e` is **closed** in environment `env`:

- If all **free** variables of `e` are defined in `env`

Successful Evaluation

`lookup` will never fail

- If `eval env e` is only called on `e` that is closed in `env`

QUIZ

Which variables occur free in the expression?

```
let y = (let x = 2
         in x      ) + x
in
  let x = 3
  in
   x + y
```

FREE

better eval in env containing 'x'

(A) None

(B) x

(C) y

(D) x and y

Exercise

Consider the function

```
evaluate :: Expr -> Value
evaluate e
  | isOk e    = eval emptyEnv e
  | otherwise = error "Sorry! bad expression, it will crash `eval`!"
  where
    emptyEnv  = []                -- has NO bindings
```

guarantee no VUndef

it may return VUndef

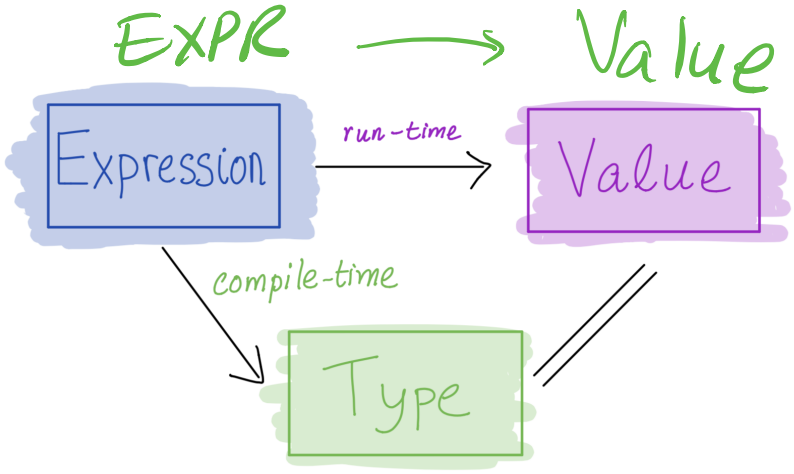
What should `isOk` check for? (Try to implement it for `nano` ...)

The Nano Language

Features of Nano:

(\x -> e)

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

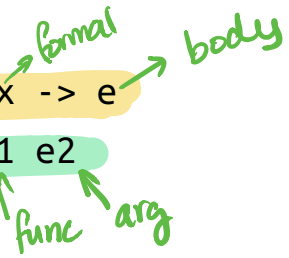


Nano: Functions

Let's add

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

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



Example

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

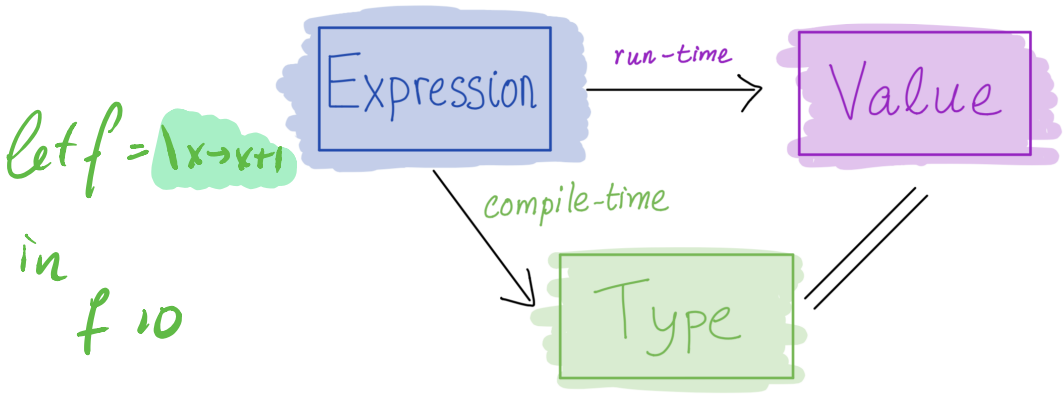
```
let incr = \x -> x + 1
in
   incr 10
```

is represented as

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

Functions are Values

Recall the trinity



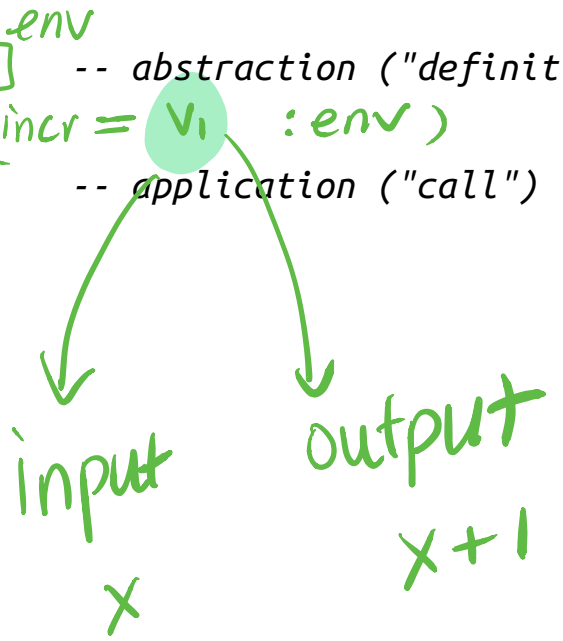
But... what is the *value* of a function?

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
```

param + body

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),