Environments

Past three weeks

How to *use* essential language constructs?

- Data Types
- Recursion
- Higher-Order Functions

Next two weeks

How to *implement* language constructs?

- Local variables and scope
- Environments and Closures
- *(skip)* Type Inference

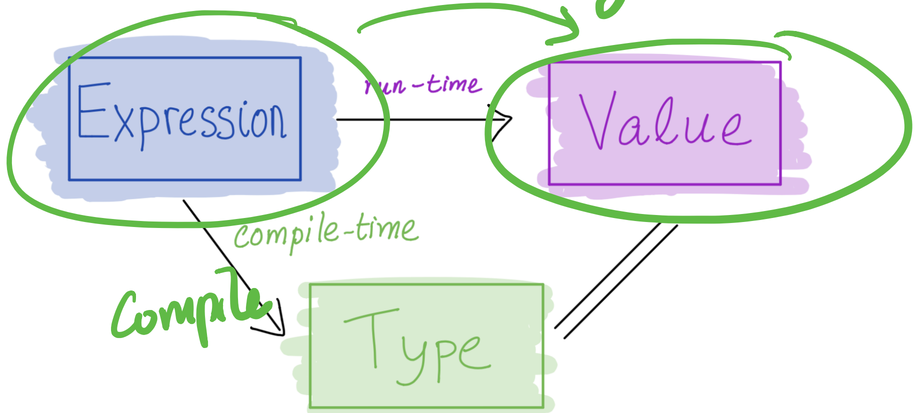
Interpreter

How do we *represent* and *evaluate* a program?

Roadmap: The Nano Language

Features of Nano:

1. **Arithmetic**
2. Variables
3. Let-bindings
4. Functions
5. Recursion



1. Nano: Arithmetic

A "grammar" of arithmetic expressions:

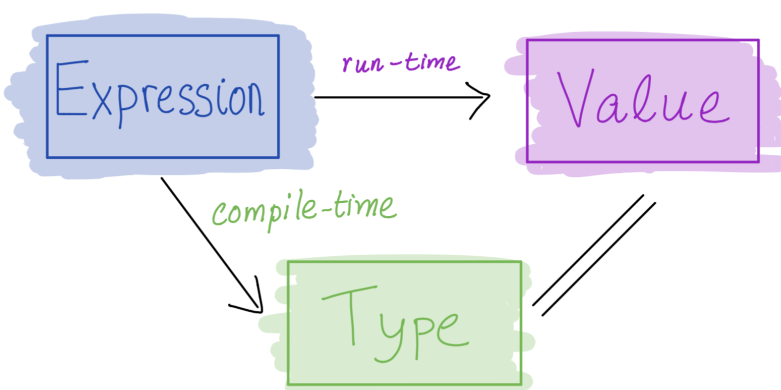
```
e ::= n
    | e1 + e2
    | e1 - e2
    | e1 * e2
```

(A) have heard of

(B) huh?

Expressions		Values
`4`	`==>`	4
`4 + 12`	`==>`	16
`(4+12) - 5`	`==>`	11

Representing Arithmetic Expressions and Values



Lets *represent* arithmetic expressions as type

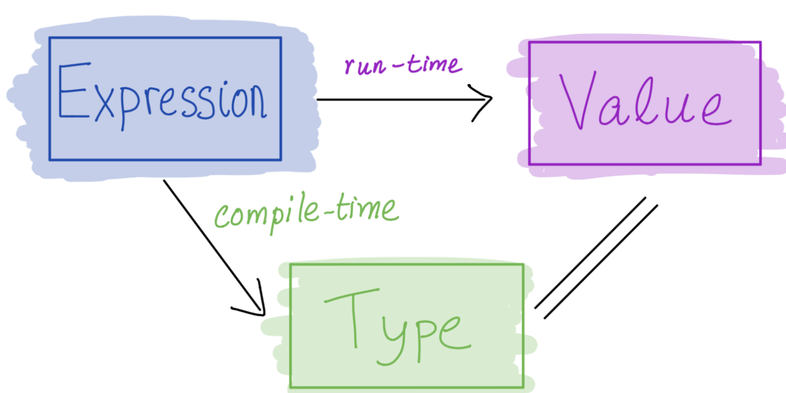
```haskell
data Expr
  = ENum Int            -- ^ n
  | EAdd Expr Expr      -- ^ e1 + e2
  | ESub Expr Expr      -- ^ e1 - e2
  | EMul Expr Expr      -- ^ e1 * e2
```

Lets *represent* arithmetic values as a type

```haskell
type Value = Int
```

data Value = VInt Int

Evaluating Arithmetic Expressions



We can now write a Haskell function to *evaluate* an expression:

```haskell
eval :: Expr -> Value
eval (ENum n)     = n
eval (EAdd e1 e2) = eval e1 + eval e2
eval (ESub e1 e2) = eval e1 - eval e2
eval (EMul e1 e2) = eval e1 * eval e2
```

Alternative representation

Lets pull the *operators* into a separate type

```
data Binop = Add                        -- ^ `+`
           | Sub                        -- ^ `-`
           | Mul                        -- ^ `*`

data Expr  = ENum Int                    -- ^ n
           | EBin Binop Expr Expr        -- ^ e1 `op` e2
```

QUIZ

Evaluator for alternative representation

```
eval :: Expr -> Value
eval (ENum n)        = n
eval (EBin op e1 e2) = evalOp op (eval e1) (eval e2)
```

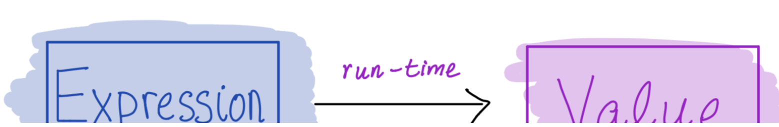
What is a suitable type for `evalOp`?

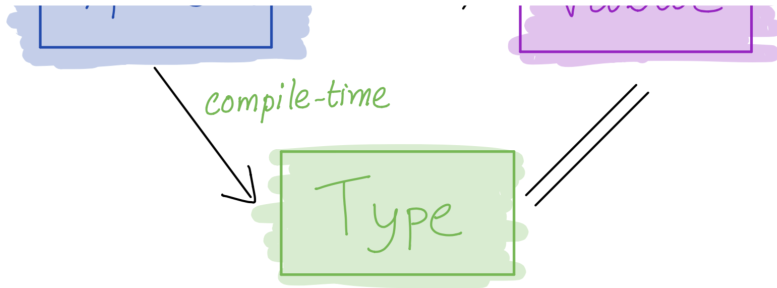
```
{- 1 -} evalOp :: BinOp -> Value
{- 2 -} evalOp :: BinOp -> Value -> Value -> Value
{- 3 -} evalOp :: BinOp -> Expr  -> Expr -> Value
{- 4 -} evalOp :: BinOp -> Expr -> Expr -> Expr
{- 5 -} evalOp :: BinOp -> Expr -> Value
```

The Nano Language

Features of Nano:

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





2. Nano: Variables

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

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

Lets extend our datatype

```haskell
type Id = String

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

QUIZ

What should the following expression evaluate to?

```
x + 1
```

EAdd (EVar "x") (EInt 1)

(A) `0`

(B) `1`

(C) Error

Haskell "eval"

Nano

Environment

An expression is evaluated in an **environment**

- A **phone book** which maps *variables* to *values*

```
[ "x" := 0, "y" := 12, ...]
```

A type for *environments*

```
type Env = [(Id, Value)]
```

Evaluation in an Environment

We write

```
(eval env expr) ==> value
```

to mean

When `expr` is **evaluated in environment** `env` the result is `value`

That is, when we have variables, we modify our `eval` uator to take an input environment `env` in which `expr` must be evaluated.

```
eval :: Env -> Expr -> Value
eval env expr = ... value-of-expr-in-env...
```

First, lets update the evaluator for the arithmetic cases `ENum` and `EBin`

```
eval :: Env -> Expr -> Value
eval env (ENum n)        = ???
eval env (EBin op e1 e2) = ???
```

QUIZ

What is a suitable `?value` such that

```
eval [ "x" := 0, "y" := 12, ...] (x + 1)  ==>  ?value
```

(A) `0`

(B) `1`

(C) Error

QUIZ

What is a suitable `env` such that

```
eval env (x + 1)    ==>    10
```

(A) `[]`

(B) `[x := 0, y := 9]`

(C) `[x := 9, y := 0]`

(D) `[x := 9, y := 10, z := 666]`

(E) `[y := 10, z := 666, x := 9]`

Evaluating Variables

Using the above intuition, lets update our evaluator to handle variables i.e. the `EVar` case:

```
eval env (EVar x)        = ???
```

Lets confirm that our `eval` is ok!

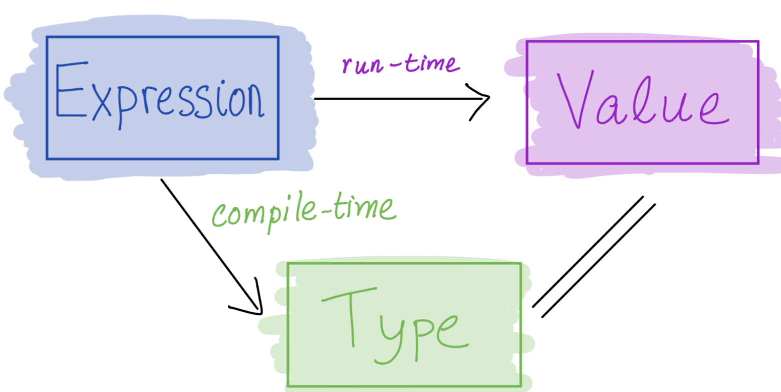
```
envA = []
envB = ["x" := 0 , "y" := 9]
envC = ["x" := 9 , "y" := 0]
envD = ["x" := 9 , "y" := 10 , "z" := 666]
envE = ["y" := 10, "z" := 666, "x" := 9  ]

-- >>> eval envA (EBin Add (EVar "x") (ENum 1))
-- >>> eval envB (EBin Add (EVar "x") (ENum 1))
-- >>> eval envC (EBin Add (EVar "x") (ENum 1))
-- >>> eval envD (EBin Add (EVar "x") (ENum 1))
-- >>> eval envE (EBin Add (EVar "x") (ENum 1))
```

The Nano Language

Features of Nano:

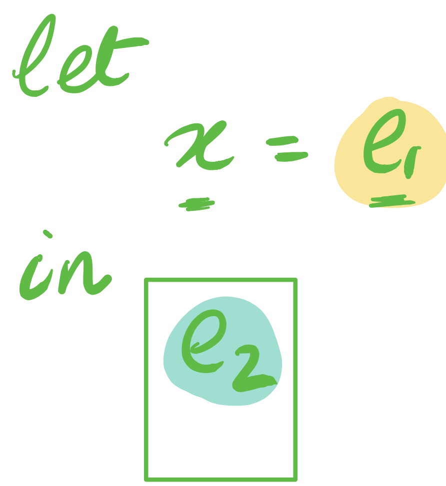
1. Arithmetic expressions *[done]*
2. Variables *[done]*
3. **Let-bindings**
4. Functions
5. Recursion



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 = e_1

y = e_2

in

e_3

(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
```

(A) Error

(B) `0`

(C) `1`

(D) `100`