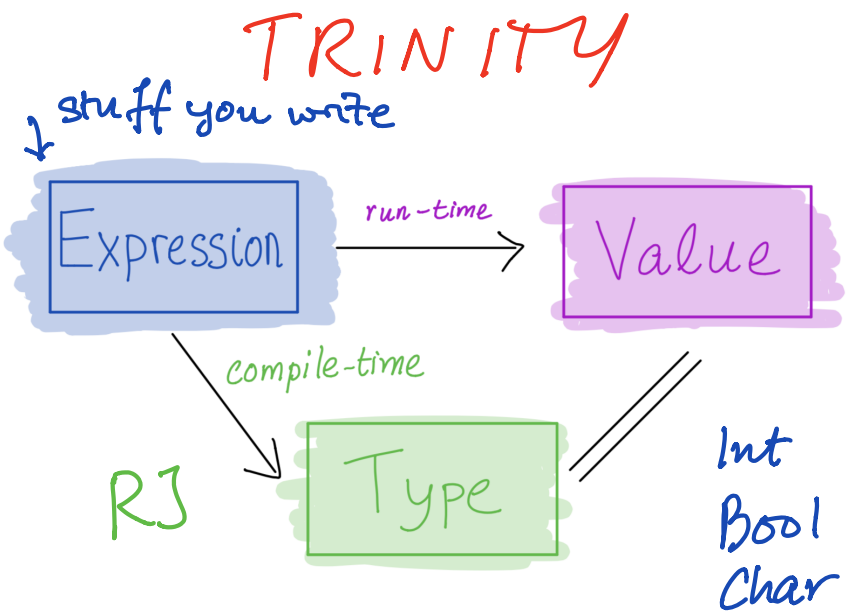
Types



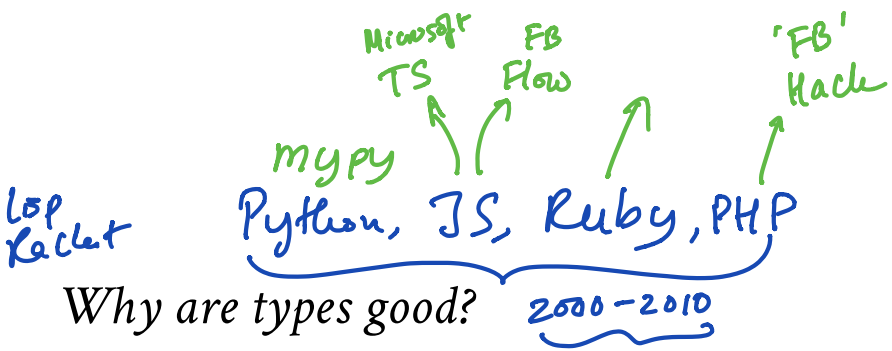
In *Haskell* every expression either

- **ill-typed** and *rejected at compile time* or
- **has a type** and can be *evaluated* to obtain _ **a value** of the same type.

Ill-typed* expressions are rejected statically at *compile-time*, before execution starts

- **like** in Java
- **unlike** λ-calculus or Python ...

```
weirdo = 1 0      -- rejected by GHC
```



Why are types good?

- Helps with program *design*
- Types are *contracts* (ignore ill-typed inputs!)
- Catches errors *early* ✓
- Allows compiler *to generate code*
- Enables compiler *optimizations*

make junk values not representable

Function Types

Functions have **arrow types**:

- `\x -> e` has type `A -> B`
- if `e` has type `B` assuming `x` has type `A`

For example:

```
> :t (\x -> if x then `a` else `b`)  -- ???
```

Always annotate your function bindings

First understand *what the function does*

- Before you think about *how to do it*

```
sum :: Int -> Int
sum 0 = 0
sum n = n + sum (n - 1)
```

*When you have *multiple arguments*

For example

```
add3 :: Int -> (Int -> (Int -> Int))
add3 x y z = x + y + z
```

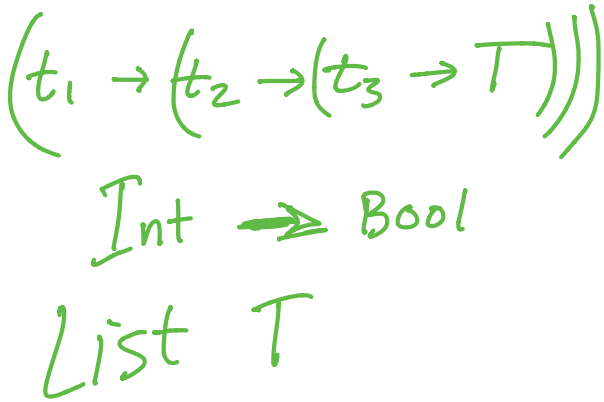
why? because the above is the same as:

```
add3 :: Int -> (Int -> (Int -> Int))
add3 = \x -> (\y -> (\z -> x + y + z))
```

however, as with the lambdas, the `->` **associates to the right** so we will just write:

```
add3 :: Int -> Int -> Int -> Int
add3 x y z = x + y + z
```

(+) = \x y -> 'call x86 inst to add x,y'



Lists

A list is

- either an *empty list*

  ```
  [] -- pronounced "nil"
  ```

- or a *head element* attached to a *tail list*

  ```
  x:xs -- pronounced "x cons xs"
  ```

Examples:

```
[]                -- A list with zero elements

1 : []            -- A list with one element: 1

(:) 1 []          -- As above: for any infix op, `x op y` is same as `(op) x y`

1:(2:(3:(4:[])))  -- A list with four elements: 1, 2, 3, 4

1:2:3:4:[]        -- Same thing (: is right associative)

[1,2,3,4]         -- Same thing (syntactic sugar)
```

Terminology: constructors and values

`[]` and `(:)` are called the *list* **constructors**

We've seen constructors before:

- `True` and `False` are `Bool` constructors
- `0`, `1`, `2` are ... well, you can think of them as `Int` constructors

    - The `Int` constructors don't take any parameters, we just called them *values*

In general, a **value** is a constructor applied to **other values**

- examples above are *list* values

The Type of a List

A list has type `[Thing]` if each of its elements has type `Thing`

Examples:

```
intList :: [Int]
intList = [1,2,3,4]

boolList :: [Bool]
boolList = [True, False, True]

strList :: [String]
strList = ["nom", "nom", "burp"]
```

Lets write some Functions

A Recipe (https://www.htdp.org/)

Step 1: Write some tests

Step 2: Write the type

Step 3: Write the code

Functions on lists: range

1. Tests lo hi

```
-- >>> ???
```

2. Type

```
range :: ???
```

3. Code

```
range = ???
```

Syntactic Sugar for Ranges

There's also syntactic sugar for this!

```
[1..7]    -- [1,2,3,4,5,6,7]
[1,3..7]  -- [1,3,5,7]
```

Functions on lists: length

1. Tests

```
-- >>> ???
```

2. Type

```
len :: ???
```

3. Code

```
len = ???
```

Pattern matching on lists

```
-- | Length of the list
len :: [Int] -> Int
len []     = 0
len (_:xs) = 1 + len xs
```

~~A pattern is either a *variable* (incl. **_**) or a *value*~~

A pattern is

- either a *variable* (incl. _)
- or a *constructor* applied to other *patterns*

Pattern matching attempts to match *values* against *patterns* and, if desired, *bind* variables to successful matches.

Functions on lists: `take`

Let's write a function to `take` first `n` elements of a list `xs`.

1. Tests

```
-- >>> ???
```

2. Type

```
take :: ???
```

3. Code

```
take = ???
```

QUIZ

Which of the following is **not** a pattern?

A. `(1:xs)`

B. `(_:_:_)`

C. `[x]`

D. `[1+2,x,y]`

E. all of the above

Strings are Lists-of-Chars

For example

```
λ> let x = ['h', 'e', 'l', 'l', 'o']
λ> x
"hello"

λ> let y = "hello"

λ> x == y
True

λ> :t x
x :: [Char]

λ> :t y
y :: [Char]
```

shout Shout SHOUT

How can we convert a string to upper-case, e.g.

```
ghci> shout "like this"
"LIKE THIS"

shout :: String -> String
shout s = ???
```

Some useful library functions

```
-- | Length of the list
length :: [t] -> Int

-- | Append two lists
(++) :: [t] -> [t] -> [t]

-- | Are two lists equal?
(==) :: [t] -> [t] -> Bool
```

You can search for library functions on Hoogle (https://www.haskell.org/hoogle/)!

Tuples

```
myPair :: (String, Int)  -- pair of String and Int
myPair = ("apple", 3)
```