Functional Programming Lecture II.

```
x :: Integer
x = x + 1
    (x+1) + 1
    ((x+1) + 1) + 1
    ⊥   -- bottom.

infinity :: Integer
infinity = infinity + 1

two :: Int → Int        ← signature.
two n = 2
```

(domain: Int; codomain / range: Int)

```
  two (infinity)   :: Int
= two (infinity + 1)
= two (infinity + ... )
```

} applicative / eager

```
  two (infinity)
= 2
```

} normal / lazy

```
double :: Int -> Int
double x = x + x
```

eager	lazy
double (3+7)	double (3+7)
= double 10	= (3+7) + (3+7)
= 10 + 10	= 10 + (3+7)
= 20	= 10 + 10
	= 20

```
let y = (3+7) in double y.
```

```
plus :: Int -> Int -> Int
plus x y = x + y
```

```
add :: (Int, Int) -> Int
add (x, y) = x + y
```

```
Int -> Int -> Int
=
Int -> (Int -> Int)
```

```
plus7 :: Int -> Int
plus7 = plus 7
```

↑ partially evaluated

Recursion.

```
fac :: Int → Int
fac 0 = 1
fac n = fac (n-1) * n
```

Fibonacci numbers.

	1	1	2	3	5	8	...
fib	0	1	2	3	4	5	

```
fib :: Int → Int
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
  | n > 0  =
  | otherwise = error "fib exploded!"
min :: Int → Int → Int
min x y = if x < y then x else y
```

* pattern guards.

```
min :: Int → Int → Int
min x y
  | x < y     = x
  | otherwise = y
```

otherwise = x ≥ y

otherwise = True

```
error :: String → a
```

polymorphic type.

Lists

[1, 1, 2, 3, 5, 8, ...]

[] empty list

(x : xs)

x: element, xs: list of elements.

1 : 1 : 2 : ...

= ['h', 'e', 'l', 'l', 'o']

'h' : 'e' : 'l' : 'l' : 'o' : []

[] :: [a]

x : xs

(:) :: ~~Char -> [Char] -> [Char]~~ (too specific!)

:: a -> [a] -> [a]

head :: [a] -> a

head [5, 3, 7] = 5

head [] = error "empty list"

head (x:xs) = x

```
tail :: [a] -> [a]
tail [] = error "empty list"
tail (x:xs) = xs
```

```
head (tail [5, 4, 3])
```

```
nats :: [Int]
nats = [1..]
```

```
take :: Int -> [a] -> [a]
take n [] = []
take n (x:xs)
  | n == 0 = []
  | otherwise = x : take (n-1) xs
```

pattern matching.

```
take 0 xs = []
take n (x:xs) = x : tak(n-1) xs.
```

```
filter :: (a -> Bool) -> [a] -> [a]
```