TOIPES & TOIPEKLASHES

haskell has type inference. it can infer type of an expression on its own.

```
λ :t 'a'
'a' :: Char
λ :t True
True :: Bool
λ :t "HELLO!"
"HELLO!" :: [Char]
λ :t (True, 'a')
(True, 'a') :: (Bool, Char)
λ :t 4==5
4==5 :: Bool
```

first character in capital case

has type of

when writing functions, wee can choose to give them an explicit type declaration. this is considered good practice except when writing very short functions.

```
removeNonUppercase :: [char] -> [char]
removeNonUppercase st = [c | c <- st, c `elem` ['A'..'Z']]

removeNonUppercase :: String -> String ✓

addThree :: Int -> Int -> Int -> Int
addThree x y z = x + y + z

:t addThree ✓
```

Int : int32 / int64 (int in C)

Integer : BigIntege (not bounded)

```
factorial :: Integer -> Integer
factorial n = product [1..n]
```

```
λ factorial 50
```

hopefully {

30414093201713378043612608166064768844377641568960512000000000000

}

Float single precision floating point

Double double precision floating point

```
circumference :: Float -> Float
circumference r = 2 * pi * r
```

```
λ circumference 4.0
25.132742
```

```
circumference' :: Double -> Double
circumference r = 2 * pi * r
25.132741228718345
```

Bool = True | False

Char : 'x'

() ← empty tuple

polymorphic function

```
λ :t head
```

type variable

```
head :: [a] -> a
```

λ :t fst

fst :: (a, b) -> a

λ :t (==)

(==) :: (Eq a) => a -> a -> Bool

all except IO, functions

class constraint

==, +, *, -, / = functions

only special characters => infix by default

to examine type

pass it to another function

Call it as a prefix function

} surround with parentheses.

λ :t elem

elem :: (Eq a) => a -> [a] -> Bool

(Eq) types that support equality testing.

functions its members implement - ==, /=

λ 5 == 5

True

λ 5 /= 5

False

λ 'a' == 'a'

True

λ "Ho Ho" == "Ho Ho"

True

λ 3.432 == 3.432

True

Ord types that have ordering. (except fns.)

λ :t (>)

(>) :: (Ord a) => a -> a -> Bool

compare (Ord a) => a -> a -> Ordering

Ordering = GT | LT | EQ

λ "Abrakadabra" < "Zebra"

True

λ "Abrakadabra" `compare` "Zebra"

LT

λ 5 >= 2

True

λ 5 `compare` 3

GT

Show can be presented as string

(Show) → String

λ show 3

"3"

λ show 5.334

"5.334"

λ show True

"True"

Read parses string to type.

(read) → String → Type (Read)

```
λ read "True" || False
True
λ read "8.2" + 3.8
12.0
λ read "5" - 2
3
λ read "[1,2,3,4]" ++ [3]
[1, 2, 3, 4, 3]
```

```
λ read "4"   X        requires type signature
```

```
λ :t read
read :: (Read a) => String -> a
                    └ needs explicit type annotation
```

```
λ read "5" :: Int
5
λ read "5" :: Float
5.0
λ (read "5" :: Float) * 4
20.0
λ read "[1,2,3,4]" :: [Int]
[1, 2, 3, 4]
λ read "(3, 'a')" :: (Int, Char)
(3, 'a')
```

Enum members are sequentially ordered types

they can be enumerated → can be used in list ranges

have defined successors & predecessors
(succ) (pred)

(), Bool, Char, Ordering, Int, Integer, Float, Double

```
λ ['a'..'e']
"abcde"
λ [LT .. GT]
[LT, EQ, GT]
λ [3..5]
[3,4,5]
λ succ 'B'
'C'
```

Bounded members have an upper and lower bound

```
λ minBound :: Int
-2147483648
λ maxBound :: Char
'\1114111'
λ maxBound :: Bool
True
λ minBound :: Bool
False
```

False ← polymorphic constant

```
λ :t maxBound
maxBound :: (Bounded a) => a
```

λ maxBound :: (Bool, Int, Char)
(True, 2147483647, '\1114111')

Num numeric type class (needs Show, Eq)

λ :t 20 ——— polymorphic constant
20 :: (Num t) => t

λ 20 :: Int
20
λ 20 :: Integer
20
λ 20 :: Float
20.0
λ 20 :: Double
20.0

λ :t (*)
(*) :: (Num a) => a -> a -> a

(5::Int) * (6 :: Integer) ✗
(5 * (6 :: Integer) ✓

Integral only integral (whole) numbers.
(Int, Integer)

Floating only floating point numbers
(Float, Double)

fromIntegral :: (Num b, Integral a) => a -> b

length :: [a] -> Int

length [1, 2, 3, 4] + 3.2 ✗

fromIntegral (length [1, 2, 3, 4]) + 3.2 ✓