**HASKELL**

> :t head

head :: [a] -> a

> :t fst

fst :: (a,b) -> a

> :t (==)

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

Here *Eq a => a -> a -> Bool* is the type of function. This means that a should be a member of typeclass Eq for it to work in this function. This is called class constraint. All standard haskell types are a member of Eq typeclass.

Any type that can be tested using == or /= should be a part of Eq typeclass.

A function can be used in infix notation by surronding it with `<function-name>`

example, elem can be used like

elem 2 [1,2,3], or

2 `elem` [1,2,3]

Similar to Eq, there is an *Ord* typeclass. All types that are a part of this class can be arranged in an ordered sequence.

Example, :t (>) should be

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

To be a member of Ord typeclass, a type must be a member of Eq typeclass also.

There also exists a *compare* function, whose job is to compare to values that are a member of Ord typeclass. *Compare* returns a value of type *Ordering*. There are 3 possible values for Ordering type; LT, GT and EQ.

example,

> *2 `compare` 3*

LT

*>* :t LT

LT :: Ordering.

*Show*  is another typeclass, which represents types that can be “shown” as strings. The most useful function that delas with Show typeclass values is *show*.

> show 3

“3”

>:t show

show :: Show a => a -> [Char]

Note that [Char] and String are same. There is no difference. Also, types always start with a capital letter.

*read* function is opposite to *show*. It takes in a String and returns a value whose type is of class *Read*. This means it can be parsed from a string into a value of its own type.

> :t read

read :: Read a => String -> a

example

> read “42” + 5

47

> read “20”

\*error\*

This error occurs as haskell doesn't know how to interpret this value. It can be Int, Integer, Float, Double. Unless we provide a context of use, there is no way to know how to parse it. Hence the error.

However, using *explicit type annotation*, we can tell interpreter how to parse the string without context. Example,

> read “20” :: Int

20

Type annotation explicitly states what the type of an expression should be.

Types of *Enum* typeclass can be enumerated, i.e., they can be used in range expressions. They have a sequential ordering from which ranges can be generated.

> succ 41

42

> :t succ

succ :: Enum t => t -> t

Types of *Bounded* typeclass have upper and lower bounds.

> minBound :: Int

-9223372036854775808

> :t minBound

minBound :: Bounded a => a

They are like *polymorphic constants*, as their return value depends on the type, and its constant.

Tuples can also be Bounded if all elements inside them are Bounded.

*Num* is the numeric typeclass.

Whole numbers are also polymorphic constants. They can act as any type that is a member of Num typeclass, eg, Int, Integer, Float etc

> 20 :: Float

20.0

> :t 20

20 :: Num a => a

*Integral* typeclass includes only Int and Integer. *Floating* includes Float and Double.

> :t fromIntegral

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

This is for those general situations when a function returns an Integral type and its result needs to be used with Floating point types for calculation.

RealFloat is another typeclass. It is probably a subclass of Floating. Do not know its exact purpose right now, but maybe it excludes irrational numbers?

*let* bindings are another way to define local variables, other than *where*. They are more local than where, as they do not span across guards.

Syntax of let bindings;

*let <variable definitions> in <expression>*

example

*cylinder r h =*

*let pii = pi*

*sideArea = 2 \* pii \* r \* h*

*topArea = pii \* r^2*

*in sideArea + 2\*topArea*

Note that variables are aligned to left. This is required, similar to where bindings. let binding can be used anywhere an expression can be used.

If we want to bind several variables in one statement in a single line, variable declarations can be separated by semi-colons

> *4 \* (let a=10; b=20 in a+b)*

Even pattern matching is available in let bindings

> *4 \* (let (a,b) = (1,2) in a+b)*