**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)*

Curried functions – Every function in Haskell actually takes only one parameter. When we create functions that take more than 1 parameter, we are creating intermediate functions. Example-

> max :: Ord a => a -> a -> a

This is same as max :: Int a => a -> (a -> a)

This means that mult is a function that takes an Int as parameter, and returns another function that takes an int as param and returns an integer.

Therefore,

> :t (max 10)

(max 10) :: (Num a, Ord a) => a -> a

Infix functions can also be curried (applied partially) by using sections. This is just a fancy name for converting an infix function to prefix notation by using parenthesis

addTen :: (Num a) => a -> a

addTen = (+) 10, or

addTen = (+10)

Note that this does not apply to ‘-’ function. If it was so, (-4) would mean a function that takes a parameter and subtracts 4 from it, instead of -4, simply.

To actually get a function that subtracts -4 from parameter, use (subtract 4).

This works for both the parameters in infix operations.

(3+) is also a function, but it adds parameter to 3, instead of the other way round. To make it more clear, take this example

> (++ “That’s what YOU think!”)

This function appenmds this annoying appendage to every string passed to it. Instead, if we used

> (“Basically, ” ++)

This function will prefix any parameter with the given phrase. Lesson – Order matters here.

A function that flips the parameters -

*flip' :: (a → b → c) → (b → a → c)*

*flip' f = g*

*where g x y = f y x*

Here, *(g x y)* returns *(f y x)*. *(g x)* returns a function that takes one param and swaps it with the in built param x before returning result. g is a function that takes 2 parameters and returns *f y x*.

Since → is right- associative, we can remove the parenthesis and write it like

*flip' :: (a → b → c) → b → a → c*

This means flip' is a function that can be called with 3 params ( first param is a function) and that would return a value such that f is applied with its parameters reversed

*flip' f x y = f y x*

This works, since, if we call flip' with just one parameter (function f), it returns a partial function, which is exactly what we need flip' to do.

**or** function in haskell takes a list of boolean values as input and returns a Bool as output.

MODULES

Haskell standard library is split into *modules*. A module can be imported in current script by

*import <module-name>*

All the functions used so far were a part of *Prelude* module. They are imported by default when ghci is fired up.

When in ghci, modules can be imported by

*> :m + <module-name>*

This imports all functions in the module in a global namespace.

*nub* function in Data.List module removes duplicates in a list. It only keeps the first element and removes the rest.

To import all the functions in a module, except a few, use

> import Data.List hiding (nub)

If you don’t want to pollute the global namespace by imports, use *qualified* imports as follows

> import qualified Data.List

This means that each function in Data.List has to be prefixed by *Data.List.*

However, there’s a better way to do this, if writing Data.List everytime seems too verbose

> import qualified Data.List as M

Now functions like nub can be called as

> M.nub [1,2,2,23]

Similar to *or*, *and* also takes a predicate and a list and checks if all of the resulting Booleans are true.

> and (==4) [4,8 `div` 2]

True

Other such functions are

> any (==4) [1,2,3,4,5]

True

> all (>4) [6,7,8]

True