**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 deals 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

> intersperse 1 [2,3,4]

[2,1,3,1,4]

Adds the first parameter between every 2 consecutive elements of supplied list.

> intercalate “ “ [“Hey”, “There”]

“Hey There”

Inserts first parameter (a list) between elements of a nested list, and then flattens the resulting list.

> concat [“foo”, “bar”, “baz”]

“foobarbaz”

Flattens a list of lists into a list.

> :t concatMap :: (a -> [a]) -> [a] -> [a]

It first applies a function of type (a -> [a]) to a list. This creates a nested list, and then flattens the resulting nested list using concat.

In essence, this is same as *concat.(map f)*

> splitAt 3 “Kaustubh”

(“Kau”, “stubh”)

It can take any numeric value as parameter.

> iterate (\*2) 1

[1,2,4,8,16…]

This produces an infinite list. Takes a function and a starting value. Function is applied successively to result infinitely. Should be used in conjugation with take or takeWhile type functions.

Similar to *takeWhile*, we have *dropWhile*, which drops the elements of the list until the predicate is true.

*span* is like *takeWhile*, but it returns a pair of list. The first part is what would be returned if takeWhile was called instead, and second part is is the remainder of list.

*break* is like opposite to *span.* It takes the list elements until the predicate is true and then breaks at that point

> break (==4) [1,2,3,4,5,6,7]

([1,2,3],[4,5,6,7])

*group* takes a list and groups adjacent elemenets into sublists if they are equal.

> group [1,2,2,3,3,3,4,5,5]

[[1],[2,2],[3,3,3],[4],[5,5]]

*inits* returns a sequence of all possible prefixes of a list

> inits “abcd”

[“”, “a”, “ab”, “abc”, “abcd”]

*tails* is similar. It returns the sequence of all possible suffixes, beginning with the longest.

> tails “abcd”

[“abcd”, “bcd”, “cd”, “d”, “”]

*isInfixOf* function is the Haskell equivalent of substring search.

> “cat” isInfixOf “concatenate”

True

To search for a pattern at beginning or end, use *isPrefixOf* and *isSuffixOf* respectively.

*Maybe* is an interesting type. It can have 2 values, *Just <something>,* or *Nothing*

It is typicaly used as a return value for functions that may or maynot return something. For example, let’s say we are trying to find the first element in a list that satisfies a particular predicate. It is not manadatory that we’ll find such an element. In that case, returning Nothing makes sense. If we want to return something, return it as a Just value. Note that Just can only encapsulate a single value

> find (>3) [1,2,3,4,5]

Just 4

> find (>3) [1,2,3]

Nothing

Another good example is *elemIndex*.

> ‘i’ `elemindex` “kaustubh”

Nothing

> ‘a’ `elemindex` “kaustubh”

Just 1

Now, there are many ways to extract the value from Maybe type. Go lookup *fromJust*, *fromMaybe, maybeToList* etc.

*elemIndices* – returns a list of indices that satisfy the predicate

*findIndex* – finds first index at which predicate holds. Wrapped in Maybe.

*findIndices* – list of such indices.

*zip –* zips 2 lists together to create a list of tuple. Bigger list is cut to size.

*zipWith* – zips 2 lists together using a binary function. Result is a list.

Since we may require to zip more than 2 lists together, zip3, zip4… zip7 variants exist. Similarly, zipWith3 etc also exist.

*lines* – used to break a String into list of strings, delimited by a \n character

> lines “This\sis\artificial”

[“This”, “is”, “artificial”]

*unlines* – takes a list of strings and converts them into a single string, with individual componenets separated by \n delimiter.

*words –* break strings using whitespace as delimiter.

*unwords* – joins list of words to a string, delimited by a single whitespace

> delete ‘h’ “hi there!”

“i there!”

Deletes first occurrence of 1st params from supplied list

\\ is the list difference function.

> [1,2,2,3,4] \\ [2,4]

[1,2,3]

Removes first occurrence of each character in second list from first list.

*union* – adds all elements of 2nd list that aren’t already in 1st list.

*intersect* - returns only the elements that can be found in both the lists.

*insert* - inserts the element in list before the first element which is larger.

Many functions like *length, drop, take, splitAt* take or return **Int***,* which can be restrictive, if we want a general Num type value. Data.List module has generic equivalents which can produce/take Num type values, such as *genericLength, genericDrop* etc

*nubBy* is similar to *nub*, but takes an additional predicate to check for equality, instead of using (==) by default. This can be used where we want to supply custom equality test.

Similar functions exist for other such functions, like *deleteBy, unionBy, intersectBy* and *groupBy.* All such predicates are binary functions.

These 'By' type functions use an equality test, where definition of equality is changed. Let's say we want to group elements by their sign (positive or negative).

The predicate can be written like -

*(\x y -> ((x > 0)&&(y>0)) || ((x <= 0)&&(y<=0)))*

A better way to write it could be

*(\x y -> (x > 0) == (y > 0))*

Turns out this is something we want to do fairly often when working with 'By' type of functions. There exists an abstraction in the form of *on* function, defined in *Data.Function,* which is defined as follows

*on :: (a -> a -> b) -> (c -> a) -> c -> c -> b*

*on f g x y= f (g x) (g y)*

Note that in previous example, f can be substituted for (==) and g for (>0). The predicate can then be written as

*(==) `on` (>0)*

Think of it as – f is applied on g, where g is a unary function, and f is a binary function.

*sortBy* is similar. It can be used to sort based on a user defined function, rather than built in *compare*. Let's say we have a list of lists and we want to arrange it such that sublists are in ascending order of length. *sortBy* expects the passed in function to be of type

*a -> a -> Ordering*

The function to be passed can be written as

*(\x y -> (length x) `compare` (length y))*

Using 'on' function, it can be written more elegantly as

*(compare `on` length)*

It also reads quite nicely :)

Data.Char module defines an enumeration type called *GeneralCategory*. It presents a few categories (31 to be precise) to which a character can belong. To check what category a character belongs to, there's a function called *generalCategory.*

> generalCategory '.'

OtherPunctuation

> map generalCategory “a >#$”

[LowercaseLetter,Space,MathSymbol,OtherPunctuation,CurrencySymbol]

*GeneralCategory* type is a part of Eq typeclass. Hence, characters can be compared on GeneralCategory also.

> generalCategory ' ' == Space

True

*intToDigit –* Int to Char, valid values are 0 through 15

*digitToInt –* Char to Int. Valid values are '0' through 'F' or 'f'

*ord* – Ascii value of character

*chr* – Character represented by Ascii value

Data.Map exports functions that conflict with those of Prelude and Data.List, so always do a qualified import.

*fromList* function takes an association list (list of Key-value pairs) and returns a Map

If there are duplicates in the association list, they are discarded. Just one is kept, and there is no control over which one.

To use with map, keys have to be part of Ord typeclass.

> M.insert 10 100 M.empty

fromList [(10,100)]

M.empty creates an empty map.

Note what ghci prints. Since maps can't be displayed on console, ghci is displaying the equivalent *fromList* function that would generate the same map.

*M.null* *–* Checks if the map is empty. Note tha M.empty creates an empty map

*M.size –* Returns the number of elements in the map.

*M.singleton* creates a new map with a single element. We can add elements to it later

> M.singleton 2 3

fromlist [(2,3)]

*M.member* is a predicate that checks a key for list membership.

*M.map* and *M.filter* work much the same as list equivalents

*M.toList* creates a list from map.

*M.keys, M.elems –* gets the keys and values from map

*M.fromListWith* – Just like *fromList* function, except that it doesn't discard the duplicate keys. It takes a function that decides what to do with them

As we import more and more modules in ghci, the prompt gets quite long. It can be set to anything –

> :set prompt “ghci> “

To reset it back, use

> :set prompt “%s> “

> :info (+)

infixl 6 +

This tells us that (+) operator has a precendence of 6, is left associative, and infix operator

> putStrLn “Hello Haskell”

Hello Haskell

If you want to print without newline, use *putStr*.

>