**Basic Haskell**

**Basic functions**

&&, ||, == - AND, OR

==, >=, <= - Comparison

succ – returns the successor

min, max – return either min or max

div – divide two integers

infix/prefix syntax.

List – a homogenous data structure.

|  |  |  |  |
| --- | --- | --- | --- |
| List Functions | | | |
| **x : y –** add x to list y’s head. | [] – empty list. | **!! –** item access in a list, zero-based. | ++ - list concatenation |
| **head –** return list’s head. | **tail –** return list without head. | **Init –** return list without tail. | **last –** return list’s tail. |
| **length –** return length of list | **null –** checks if list is empty | **reverse –** returns a reversed list. | **take x L –** take first x elements from list L. |
| **drop x L –** drop x elements from start of list L | **Maximum/minimum –** return max/min element in a list | **elem x L –** check if element x is present in list L. |  |
| **zip –** merges corresponding elements of 2 lists to tuples |  |  |  |

**Ranges** - [1..7], [2,4..10], [10,9..1]

**Infinite lists –** [1..]

**List comprehension –** [x\*2 | x <- [1..10]

**Nested comprehension –** [ [x,y] | x <- [1..3], y <- [10..12]]

**List compare <,>, = -** compares element in appropriate positions.

Head,tail, init and last are partial functions because they don’t return a value for empty lists.

**Tuple compare –** similar to list compare.

**() –** unit tuple.

**:t x –** return the type of x.

**Polymorphic Functions –** functions that have type variables.

|  |  |  |  |
| --- | --- | --- | --- |
| Types | | | |
| **Int –** integers up to 2147483647 | **Integer –** larger integers than Int. | **Float –** real floating point with single precision. | **Double –** real floating point with double precision. |
| **Bool –** true or false. | **Char –** represents a character. | **Type variables –** a variable which can be of any type. |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

[1,2,3] is equivalent to 1 : 2 : 3 :[] which is equivalent to 1 : (2: (3: []))

**Type classes –**  a sort of interface that defines some behavior.

**Pattern matching –** specifying patterns to which some data should conform.

* We should always write a catch-all pattern in functions, in order to avoid unexpected input.
* **as patterns –** performed by putting a name and @ in front of a pattern.

for example all@(x:xs), *all* will get the whole type value.

**Guards –** guards are somewhat similar to switch blocks, they are written with piplelines ‘*|’*.

A last guard can be written as *otherwise,* which is similar to an else statement.

**Common Modules**

**import <module name>** ­ - imports all function that the module exports directly to the global namespace.

**import <module name> (foo, goo) –** imports only foo and goo from the module.

**import <module name> hiding (foo)** – imports everything besides foo.

**Import qualified <module name> -** imports everything but must be referenced with module name.

**Import qualified <module name> as X –** imports everything but can be referenced using X.

**Data.List –** holds functions related to lists.

Some noteworthy functions from Data.List:

* intersperse takes an element and a list and then puts that element in between each pair of elements in the list
* transpose transposes a list of lists. If you look at a list of lists as a 2D matrix, the columns become the rows and vice versa.
* foldl' and foldl1' are stricter versions of their respective lazy incarnations.
* concat flattens a list of lists into just a list of elements.
* any and all take a predicate and then check if any or all the elements in a list satisfy the predicate
* iterate takes a function and a starting value. It applies the function to the starting value, then it applies that function to the result, then it applies the function to that result again, etc. It returns all the results in the form of an infinite list.
* splitAt takes a number and a list. It then splits the list at that many elements, returning the resulting two lists in a tuple.
* takeWhile is a really useful little function. It takes elements from a list while the predicate holds and then when an element is encountered that doesn't satisfy the predicate, it's cut off. It turns out this is very useful.
* dropWhile is similar, only it drops all the elements while the predicate is true. Once predicate equates to **False**, it returns the rest of the list. An extremely useful and lovely function!
* sort simply sorts a list. The type of the elements in the list has to be part of the **Ord** typeclass, because if the elements of a list can't be put in some kind of order, then the list can't be sorted.
* unlines is the inverse function of **lines**. It takes a list of strings and joins them together using a **'\n'**.
* \\ is the list difference function. We also have union and intersect.

**Data.Char –** holds functions related to characters.

Noteworthy function:

* isSpace checks whether a character is a white-space characters. That includes spaces, tab characters, newlines, etc.
* isLower checks whether a character is lower-cased.
* isUpper checks whether a character is upper-cased.
* isAlpha checks whether a character is a letter.
* isAlphaNum checks whether a character is a letter or a number.
* isNumber checks whether a character is numeric.
* toUpper converts a character to upper-case. Spaces, numbers, and the like remain unchanged.
* toLower converts a character to lower-case.

**Data.Map –** holds function related to the data structure Map (key\value pairs).

The most basic convention of a Map is a **list of tuples (with the tuples acting as key\value pairs).**

The keys in the association list must be orderable.

Noteworthy function:

* The fromList function takes an association list (in the form of a list) and returns a map with the same associations. If there are duplicate keys in the original association list, the duplicates are just discarded
* empty represents an empty map. It takes no arguments, it just returns an empty map.
* insert takes a key, a value and a map and returns a new map that's just like the old one, only with the key and value inserted.
* null checks if a map is empty.
* singleton takes a key and a value and creates a map that has exactly one mapping.
* size reports the size of a map.
* map and filter work much like their list equivalents.
* toList is the inverse of **fromList**.

**Data.Map –** holds functions related to sets.

Noteworthy functions:

* The fromList function works much like you would expect. It takes a list and converts it into a set.
* intersection , difference and union works similarly to how they work in set theory.
* The null, size, member, empty, singleton, insert and delete functions all work like you'd expect them to.

**Making our own modules**

At the beginning of a module, we specify the module name. If we have a file called **Geometry.hs**, then we should name our module **Geometry**. Then, we specify the functions that it exports and after that, we can start writing the functions. So we'll start with this.

1. **module** Geometry
2. ( sphereVolume
3. , sphereArea
4. , cubeVolume
5. , cubeArea
6. , cuboidArea
7. , cuboidVolume
8. ) **where**

**Making Our Own Types and Typeclasses**

We can make our own data types using the Data keyword.

**data** Bool = False | True

Bool is the type, true and false are value constructors.

**data** Shape = Circle Float Float Float | Rectangle Float Float Float Float

Shape is the type, Circle and Rectangle are also value constructors.

Value constructors are functions for all purposes and intents.

Example of **record syntax** which is the approach we will use when writing new data types:

1. **data** Person = Person { firstName :: String
2. , lastName :: String
3. , age :: Int
4. , height :: Float
5. , phoneNumber :: String
6. , flavor :: String
7. } **deriving** (Show)

In somewhat similar fashion to how function have type variables, data types also have **type constructors** that can take types as parameters in order to produce new types.

For example,

**data** Maybe a = Nothing | Just a

Or,

1. **data** Car a b c = Car { company :: a
2. , model :: b
3. , year :: c
4. } **deriving** (Show)

Important note about typeclass constraints in data declarations:

* It’s not recommended to add typeclass constraints in data declarations because functions involving those data types have to be constrained anyway, and so it is arbitrary.

**Type synonyms –** provide an ability to give types, other more readable names, for example:

**type** String = [Char]

**Fixity declarations –** defining functions as operators and giving them a fixity – how tightly the operator binds and whether it’ left or right associative.

**Creating typeclasses**

An example for creating a typeclass

1. **class** Eq a **where**
2. (==) :: a -> a -> Bool
3. (/=) :: a -> a -> Bool
4. x == y = not (x /= y)
5. x /= y = not (x == y)

an example of making our own derived instance

1. **instance** Eq TrafficLight **where**
2. Red == Red = True
3. Green == Green = True
4. Yellow == Yellow = True
5. \_ == \_ = False

It is possible to to make a subclass of another type class by using class constraints in instance declarations.

* Data types can also be defined recursively, we saw an example of a binary tree.

**Yes-No Typeclass**

Can be implemented to determine whether one value of a type is considered to hold some concept of trueness.

**The Functor Typeclass**

The functor typeclass represents mapping between categories in the context of category theory. In practice, a functor represents a type that can be mapped over.

* A functor wants a type constructor that takes **one type** and not a concrete type. For example, Maybe is a type, Maybe Int is a concrete type.
* **:k** can be used to retrieve the kind of a type.
* **\*** denotes a concrete type.
* **\* -> \*** , takes a concrete type and returns a concrete type.

**Input and Output**

An I/O action is an action that when performed, will carry out an action that may include a side-effect (contrary to normal functions which don’t have side effects), and will also contain some kind of return value.

The empty tuple is a value of () and also has a type of ()

Using the **do** keyword after main glues together several I/O actions.

The **<-** operator allows binding of results returned from I/O actions.

In a **do** block, that last action can’t be bound to a name, this is because a **do** block will always produce the return value of its last action.

* In an I/O do block, *ifs* have to have a form of **if *condition* then *I/O action* else *I/O action*.**  (remember than every if statement, must include an else statement)
* Unlike imperative languages, in Haskell a return statemen doesn’t end the execution of a block of code, it simply returns some form of I/O action.

putStr – takes a string and returns an I/O action which will print the string to the screen, without jumping into a new line.

putChar takes a character and returns an I/O action that will print it out to the terminal.

print takes a value of any type that's an instance of **Show** (meaning that we know how to represent it as a string), calls **show** with that value to stringify it and then outputs that string to the terminal.

When printing a string we will usually use putrStrLn because it doesn’t keep the quotes around the string like print does.

For almost all other values we will use **print**.

getChar is an I/O action that reads a character from the input.

The when function is found in **Control.Monad** (to get access to it, do **import Control.Monad**). It's interesting because in a *do* block it looks like a control flow statement, but it's actually a normal function. It takes a boolean value and an I/O action if that boolean value is **True**, it returns the same I/O action that we supplied to it. However, if it's **False**, it returns the **return ()**.

sequence takes a list of I/O actions and returns an I/O actions that will perform those actions one after the other.

forever takes an I/O action and returns an I/O action that just repeats the I/O action it got forever. It's located in **Control.Monad**.

**Files and stream**

getContents is an I/O action that reads everything from the standard input until it encounters an end-of-file character.

getContents can be referred to as “Lazy I/O” – meaning it will read from the terminal later on, and not immediately when it was called.

interact. **interact** takes a function of type **String -> String** as a parameter and returns an I/O action that will take some input, run that function on it and then print out the function's result.

openFile receives a FilePath ( which is a string basically) and an IOMode(ReadMode\WriteMode\AppendMode\ReadWriteMode) and returns an IO Handle. We will usually bind our handles so we can do something with them.

hGetContents takes a Handle and returns an IO String. This is identical to getContents except the former reads from the terminal.

hClose, which takes a handle and returns an I/O action that closes the file. You have to close the file yourself after opening it with **openFile**!

withFile function, which has a type signature of **withFile :: FilePath -> IOMode -> (Handle -> IO a) -> IO a**. It takes a path to a file, an **IOMode** and then it takes a function that takes a handle and returns some I/O action. What it returns is an I/O action that will open that file, do something we want with the file and then close it. The result encapsulated in the final I/O action that's returned is the same as the result of the I/O action that the function we give it returns.

Just like we have **hGetContents** that works like **getContents** but for a specific file, there's also hGetLine, hPutStr, hPutStrLn, hGetChar, etc. They work just like their counterparts without the *h*, only they take a handle as a parameter and operate on that specific file instead of operating on standard input or standard output.

readFile takes a path to a file and returns an I/O action that will read that file (lazily, of course) and bind its contents to something as a string. It's usually more handy than doing **openFile** and binding it to a handle and then doing **hGetContents**.

writeFile has a type of **writeFile :: FilePath -> String -> IO ()**. It takes a path to a file and a string to write to that file and returns an I/O action that will do the writing. If such a file already exists, it will be stomped down to zero length before being written on.

appendFile has a type signature that's just like **writeFile**, only **appendFile** doesn't truncate the file to zero length if it already exists but it appends stuff to it.

openTempFile. Its name is pretty self-explanatory. It takes a path to a temporary directory and a template name for a file and opens a temporary file. We used **"."** for the temporary directory, because **.** denotes the current directory on just about any OS.

**Command Line Arguments**

Command line arguments allows giving the program some initial arguments before we run it.

The **System.Environment** module has two cool I/O actions. One is getArgs, which has a type of **getArgs :: IO [String]** and is an I/O action that will get the arguments that the program was run with and have as its contained result a list with the arguments. getProgName has a type of **getProgName :: IO String** and is an I/O action that contains the program name.

**Randomness**

Haskell functions related to randomness are stored in the **System.Random** module.

The RandomGen typeclass is for types that can act as sources of randomness. The Random typeclass is for things that can take on random values. A boolean value can take on a random value.

In order to use the **random** function we have to supply it with a random generator.

The **System.Random** module exports a cool type, namely StdGen that is an instance of the **RandomGen** typeclass. We can either make a **StdGen** manually or we can tell the system to give us one based on a multitude of sort of random stuff.

o manually make a random generator, use the mkStdGen function. It has a type of **mkStdGen :: Int -> StdGen**. It takes an integer and based on that, gives us a random generator.

Example of using random function:

1. ghci> random (mkStdGen 100) :: (Int, StdGen)
2. (-1352021624,651872571 1655838864)

randoms takes a generator and returns an infinite sequence of values based on that generator.

randomR works similarly to random, but also receives a tuple denoting a lower and upper bound for the random values we get.

There's also randomRs, which produces a stream of random values within our defined ranges.

getStdGen returns an I/O action of type IO StdGen, this allows us to ask the system for a random number generator and bind it.

**Packages**

Each Haskell module resides in a package.

The base package is enabled by default, and other packages have to manually enabled, for example **Data.Map** is in the “containers” package.

We can enabled a new package by typing “:set -package <package name>” in the ghci.

**Functors Redux**

As we’ve already seen, functors are things that can be mapped over , lists, tree and such.

Type constructors can be made an instance of functors only if they take a single type parameter, for example **Maybe a**, otherwise we have to partially apply the type constructor until it only takes one type parameter.

I/O actions are also instances of functor, they are defined in the following manner:

1. **instance** Functor IO **where**
2. fmap f action = **do**
3. result <- action
4. return (f result)

Using fmap on functions is essentially performing function composition.