**Haskell Cheatsheet.**

* To run, open powershell and type ***ghci***
* To quit, type ***:quit***
* Help, type ***:?***
* Haskell files end .hs
* Load file with **:load** file\_path\name or **:l** file\_path\name
* To load a file with spaces, wrap in double quotes and escape the backslashes,

e.g. :l “C:\\foo bar.hs”

* Reload file with **:reload** or **:r**
* **Fn + home**  moves curser to start of line
* **Fn + end** moves curser to end of line
* **Ctrl + c** stops Haskell hanging

Haskell GHCi options: Turn on with **:set**  followed by option wanted Turn off with **:unset** followed by option

**Options:**

+m Allow multiline commands

+r Revert top-level expressions after each evaluation

+s Print timing/memory stats after each evaluation

+t Print type after evaluation

-<flags>      Most GHC command line flags can also be set here. (eg. -v2, -XFlexibleInstances, etc.)

Get function format:

**:t functionName** e.g. :t map returns format of function, e.g. map : : (a -> b) -> [a] -> [b] note that implix operators such as “:” must be put in brackets to get the type, ie: :t ( : )

Assign to a **Variable:**

let variableName = keyword **let** assigns everything right of the equals to the variable Name

**Dollar** symbol:

Right alignment precedence….ie everything to its right is treated as the parameter to the function on its left. Can also be used as Opening bracket without corresponding closing bracket (end of line is closing bracket) when no more brackets to its right, use: **$** e.g. head $ tail xs == head ( tail xs)

**Dot notation (function composition)**, do first one, then do second: (second**.**first) brackets only necessary when other components are present in function, not required when using point free style.

**Point free style**: This is when input notation is not put in function and Haskell works it out. e.g. functionName = head.tail will work fine without having e.g. xs on each side

**Infix** function: Functions are normally prefixed before the operands, but they can also be infixed and sandwiched in between to improve readability by adding backwards ticks **`** around the function name**`**

**Arithmetic**:

Power symbol ^ e.g. ^2 = power of 2 = squared

Subtract **Subtract** -1 would mean the negative integer rather than

operator, however this works fine sometimes such as 5-1

Remainder **rem x y** Where x is the number to divide by y and

return the remainder, can also use **mod** NOTE – **rem 0 n** returns **zero**

Integral Division **Div x y** x is divided by y and the **decimal** part is

**discarded.**

Pi **pi** value of pi

abs(x) Returns the absolute value of x, e.g. abs(-5) returns 5

Sum elements of a list **sum [**1,2,5,8**]** returns 16

**product** [6,2,1,2] returns the product of all the elements, example returns 24

Prefixed functions can be written:

(+) 3 4 function “+” is applied to arguments 3 and 4

Inequality comparison **/=**

Logical and (wedge) **&&**

Logical or (vee) **||**

Equals comparison **==**

Logical “not” **not**

Check even **even**

Check odd **odd**

if and only if **iff**

**Lists**:

Lists can **only** contain the same type of elements and have order.

Note, lists can contain other lists.

Lists come in two forms:

[ ] an empty list

(x : xs) represents lists with at least one thing in. x = ‘ecks’ = element, xs = ‘exes’ = list : is the ‘cons’ operator. xs is a list which contains elements of type x Can use more than one cons to split up definition and make it more implicit, e.g. (x : y : xs) means ‘x’ stuck on the front of ‘y’ stuck on the front of list ‘xs’

create a list -

‘a’ : [ ] creates a char list

'a' : 'b' : [ ] == [ 'a', 'b' ] also creates a list

[1..20] list of all whole integers between 1 and 20

[3,6..20] define step, note can only define 1 step and not [1,2,4,6,8,16..100]. Be careful with floating point steps as can yield strange results

[20,19..1] define step to get descending list

[50..] Infinite list from 50 and up

Indexing:

[5,6,7,8] **!!** 2 returns 7. Indexing starts at 0, use **!!**

Prepend:

4 : [1,2,3] Prepends 4 to the front of the list, returns [4,1,2,3] Only takes **single elements, not lists**. This is instantaneous in Haskell.

Append:

[1,2,3] ++ [4,5,6] Appends 2nd list to 1st, returns [1,2,3,4,5,6] Must both be **lists.** This can take along time on large lists as Haskell must step through the entire list.

**List comprehension:**

[x | x <- [1..10], condition2, condition3] means “pick x”, | means “such that”, x <- [1..10] is an example meaning “for every element in the list, x is drawn from the list 1 to 10” (binding parts, where the value is bound to x), <- represents membership function ∈. Then the conditions (predicates) are listed separated by commas which mean “AND”. Note you can also have **more than** **one binding part** and draw from multiple lists, these must be specified before the predicates and comma separated. e.g. [x\*y | x <- [1..10], y <- [20..50], condition1, condition2] this will return a list of all possible combinations from both lists which met the conditions.

[partDone2nd | partDone1st] [thisSideEmulatesMap | thisSideEmulatesFilter]

[x | x <- [1..100], x < 10] returns a list of all the natural numbers which are in the list 1 to 100 AND less than 10

[x^2 | x <- [1..20]] returns a list of the square of each number between 1 and 20

Note functions can be applied to list comprehensions:

map (subtract 1) [x | x <- [1..50], even x] returns a list of all the odd numbers between 1 and 50 (without the use of the odd function)

comprehension length function example: length’ xs = sum [1 | \_ <- xs] **\_** means we don’t care what we’ll draw from the list, for each element, 1 will be added to a list and then

all the elements at the end to give the length of the list.

**In-built functions of lists:**

head(x:xs) returns **first** **element** of the list – works on strings also will produce an error if used on an empty list

init (x:xs) returns **list** with **last part removed**, e.g. init [1,2,3,4] returns [1,2,3]

tail(x:xs) returns **list** with **head removed** (xs or ‘exes’), e.g. tail[1,2,3,4] returns [2,3,4] - works on strings also (list of chars)

last (x:xs) returns **last element** of a list , e.g. last [1,2,3,4] returns 4 last [x] returns x

null [ ] returns True if list is empty

take 2 [1,2,3,4] returns the number of elements specified from the front as a list. In example given, would return [1,2]

drop 2 [1,2,3,4] removes number of elements specified from front of list. In example given, would return [3,4]

maximum [1,2,3,4] returns the largest element. Example would return 4

minimum [1,2,3,4] returns the smallest element. Example would return 1

elem 4 [3,4,5,6] returns True if element specified is present in list

take 5 (**cycle** [2,3]) cycles list **infinitely** so be sure to slice it, example returns [2,3,2,3,2]

take 3 (repeat [2,3]) similar to cycle and **infinite**, but with one item, example returns [[2,3],[2,3],[2,3]]

replicate 3 10 similar to repeat, but you specify how many times, example returns [10, 10, 10]

words “some words” Takes a string and returns a list of words, example returns [“some”, “words”]

unwords [“some”, “words”] Takes a list of strings and returns a string, example returns “some words”

zip [1,2,3] [2,3] zips 2 lists together truncating the longer list, returns tuple pairs in a list. Example returns [(1,2),(2,3)]

zipWith (+) [1..5] [5..15] Similar to zip however this takes a function and 2 lists applying the function as it zips the two lists together. Example returns [6, 8, 10, 12, 14]

**fold functions:**

Takes a binary function, an accumulator and a list.

Function takes accumulator on left for left fold and right for right fold.

**foldl (left fold):** Takes the **accumulator** as the **1st input** to the function, and the **head** of the **list** as the **2nd input**. The **result becomes the new accumulator**. This is recursively called until the list has been transited. The final result is returned.

foldl f acc (x:y:ys) = (foldl f (f (f acc x) y) ys)

**foldl1:** Takes the head of the list as the starting accumulator for a left fold.

foldl1 f (x:y:z:zs) = (foldl f (f (f x y) z) zs)

**scanl:** Similar to foldl, however it returns a list where the accumulator is appended to the list each time it is modified by the function. Note that the initial accumulator value will be the first value in the list.

**scanl1:** scanl version of foldl1.

**foldr (right fold):** Similar to foldl, however elements are **partially evaluated starting from the head of the list.** The **last element** of the **list** is taken as the **1st input** to the function and the **accumulator** is taken as the **2nd input**. The **result becomes the new accumulator**. This is recursively called until the list has been transited. Because of partial evaluation, the list **may not** need to be fully transited. The final result is returned.

foldr f acc (x:y:ys) = (f x (f y (foldr f acc ys)))

Right folds can work on **infinite lists**: starting from the left of list (partial application),

foldr (||) \_ xs ………. if True is ever found, computation is stopped and True is returned.

foldr (&&) \_ xs …….. if False is ever found, computation is stopped and False is returned.

**foldr1:** Takes the last element as the starting accumulator for a right fold.

foldr1 f (w:x:y:z) = (f w (f x (foldr f z y)))

**scanr:** Similar to foldr, however it returns a list where the accumulator is prepended to the list each time it is modified by the function. Note that the initial accumulator value will be the last value in the list.

**scanr1:** scanr version of foldr1.

**flip:**

Takes a function and flips the first 2 arguments to return a different function

e.g. flip zip [1,2,3,4,5] “hello” returns: [('h',1),('e',2),('l',3),('l',4),('o',5)]

**filter:**

Filter **“catches”** elements which **evaluate true** for the **given condition**, and returns a **list**.

Although it does not matter which order you do chained filters to obtain the correct answer, you should always do the **EASIER COMPUTATIONAL operation FIRST,** i.e. in the most central brackets

filter :: (a -> Bool) -> [a] -> [a] **type for filter**, red parts are inputs, green part is the output. Outputs are always last. The first input (a -> Bool) is a propositional function. Both the 2nd input and output are lists.

e.g. Filter (>4) [1,2,3,4,5] returns [5]

**map** function:

**Applies** given **function** to **every** **element** in **given** **list**, returns **list**

Can chain maps together, working from **inside bracket out.**

Be CAREFUL of the order in which you do chained map functions.

map :: (a -> b) -> [a] -> [b] type for map, red parts are inputs, green part is the output. Outputs are always last. Takes a function that turns a’s into b’s as an input, it also takes a list of type a. It applies the function given to every element in the list and returns a list of type b.

map (>4) [1,2,3,4,5] returns [False, False, False, False, True] given list could be written [1..5]

map (+1) [1..5] returns a list [2, 3, 4, 5, 6] (maps +1 to range 1..5) Can also be written (+1) `map` [1.. 10]

map toLower “ANDY” returns “andy”

**length** function:

Takes input of a **list** and returns an **integer** representing the number of elements in the list

length :: [a] -> Int type for length

length [ ] returns 0

length [1..5] returns 5

**reverse** function:

Reverses input

reverse [1..5] returns [5,4,3,2,1]

reverse “hello” returns “olleh”

**Section** functions:

Surround **/** with parenthesis and only supply one parameter to either side of the slash to create a function which takes 1 parameter.

divideByTen :: (Floating a) => a -> a divideByTen = (/10)

**takeWhile** function:

Takes a predicate and a list. Starting at the beginning of the list it will return elements while the predicate remains true. E.g. takeWhile (/=' ') "elephants know how to party" returns: “elephants”

Writing **functions:**

Functions can be written in notepad++ and then loaded into powershell using Haskell with command **:load** or **:l** followed by the file path (note a path with spaces should be entered as a string, and each \ should be escaped with another \ character)

Once loaded in, further changes to the file should be saved and then reloaded with command **:reload** or **:r** followed by the file path

Functions can be called from within functions.

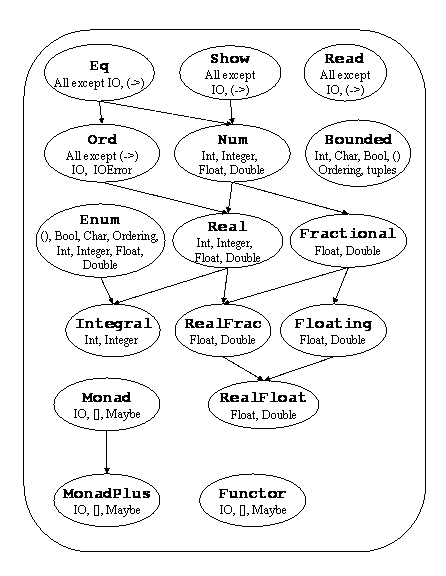
**Type signature** is defined when writing functions as: functionName :: format of inputs and output :: means “has type…” e.g. myFunction :: [a] -> a means it takes a list of type a and returns an element of type a

**Type classes**: Sometimes a type signature is further defined by defining its type class. This is put in front of

the functions inputs followed by a big arrow (the class constraint) functionName :: (Eq a) => a -> [a] -> Bool In this example, ‘a’ has been

restricted to a type class Eq (Equality), and takes inputs of type ‘a’ and a list of type ‘a’ giving an output of type Bool

**Eq**  Equality class, inputs must be a member of a type which supports equality testing This will be used when the function does some sort of equality test such as == or /= **Ord** For types that have an ordering. Used when comparing order such as >, <, >= a <= **Num** Members have the property of being able to act like numbers.



**Comment** with **--** Multi line comment **{-** some comments some more comments **-}**

Reference to inputs on the left and right of the = (such as with n, (x:xs) etc) in a function is NOT required when it is outside the functions defined as Haskell will automatically add to the end. E.g. myFunc :: [Int] -> Int myFunc = head -- Note no reference to inputs, same as: myFunc xs = head xs

**Recursive functions** Functions that can call itself in its definition. When writing recursive functions that take an input such as a list… You should define what happens with **the smallest input** e.g. an empty list With a function that takes numbers as an input… **The smallest input / base case** is **usually 0** Then: You should define how to **break down a large input to a smaller one**

It can be helpful to **write down what you can refer to** (on the right side of the definition based on the left side), especially when defining how to break down a large input

e.g. function to add 1 to numbers in a list and returns a list: addOne :: [Int] -> [Int] addOne [ ] = [ ] **- This is what to return on the smallest input** addOne (x : xs) = x+1 : addOne xs **- Here we can refer to: x i.e. head of list xs i.e. tail of list addOne xs i.e. call self on tail of list**

e.g. function to calculate the factorial (itself multiplied by every number lower than it… …note 0! = 1) fac :: Int -> Int fac 0 = 1 **- Base case** fac n = n\*fac(n-1) **- Here we can refer to: n i.e. the number passed in n – 1 i.e. the next number to pass in fac (n – 1) i.e. the factorial of the next**

**number down**

**Lambda** functions (anonymous functions) Short anonymous functions, normally for sole purpose of passing to a higher-order function. Normally surrounded by parenthesis (or they will extend all the way to the right). Defined with **\** Then write parameters delimited by a **space, NOT comma.** Followed by **-> function body** You CAN pattern match, but you CANT define several patterns for one parameter. Can have **if then else** statements, separated with **spaces**.

**Guards** When writing cases for a function, they are matched from top to bottom, these are called **“Guards”**, and must evaluate to True or False. Guards are indicated with a pipe **|**

The last guard is usually **otherwise** and catches everything by always evaluating to True. If the last guard evaluated False, then evaluation will drop through to the next patten (hence the name). You can use keyword **not** and brackets or dot function (see above) to capture a false evaluation. e.g. – a function named “unusual”, to add 1 to even numbers and double odd numbers

unusual :: Int -> Int unusual x | even x = x + 1 | otherwise = 2 \* x

note pipes must be indented by 2 spaces

keyword **where** can be used after the pipes to define names or **anonymous functions** to avoid having to repeat these throughout the guards or for functions that should **not be accessible outside of the function defined in**, could also use lambda functions. These will **only** be available to that function which they are defined after. These are syntactic constructors. e.g. – two ways to define where:

some function as above (incl. guards) where newRefName = what it does (a, b, c) = (a\_value, b\_value, c\_value)

Where functions can also include guards. Pattern matching can be done with where too, example:

test :: (Eq a) => [a] -> [a]

test xs = removeDups xs

    where removeDups [] = []

          removeDups (x:xs)

            | elem x xs = removeDups xs

            | otherwise = x : removeDups xs

**Let bindings**: Similar to where bindings, but back to front.

**Let** <bindings> **In** <expression>

These are expressions themselves.

**Case expressions:** **case** <expression> **of** pattern -> result pattern -> result pattern -> result

**Maybe** type (Exception handling)

A maybe type is a type like any other, but can be used as the return type of a function. Error cases are returned as **Nothing** and normal cases are wrapped as a **Just (case)** It is usual to call another function for the normal cases which does the work.

e.g. Assume function mySum is a function which sums all the elements in a list. It also returns an error if passed an empty list which would stop the program running. Because of this, the following can be done to protect against errors:

mySumSafe :: [Int] -> Maybe Int mySumSafe [ ] = Nothing <-- Deals with error cases mySumSafe xs = Just (mySum xs) <-- Calls function “mySum” on list xs, which cannot

be an empty list, so no error

Because Maybe is a type of its own, normal recursion doesn’t work properly as you don’t have access to the elements for comparison. Rather than writing an unsafe function and a Safe version which calls the unsafe version as above, **case of** can be used. This gives you access to the elements instead of the Maybe type.

e.g.

mySumSafe2 :: [Int] -> Maybe Int mySumSafe2 [ ] = Nothing <- Base case for Nothing mySumSafe2 [x] = Just x <- Base case for Just mySumSafe2 (x:xs) = **case** (mySumSafe2 xs) **of** <- Recursive call **Just** n -> **Just** (n + x) <- Access to returned n value **Nothing** -> **Nothing** <- If Nothing triggered, this will be returned

**Note** the **arrows** instead of equals for each case This version means the unsafe version **Cannot** be called, as all built into one….**SAFER!**

**Nothing** is the equivalent base case to the empty list [ ] **Just x** is the equivalent constructor case to lists x:xs

Can deal with Maybe types as follows: appendMaybe :: Maybe [a] -> Maybe [a] -> Maybe [a] appendMaybe Nothing Nothing = Nothing appendMaybe Nothing (Just xs) = Just xs appendMaybe (Just xs) Nothing = Just xs appendMaybe (Just xs) (Just ys) = Just (xs ++ ys)

Just make sure the middle context e.g. \_ Nothing = Nothing make sense in context instead of as above (Just xs) Nothing = Just xs

**Tuples**

Defined by parenthesis ( ) **Can** contain **different types** (unlike lists). Tuples of different lengths/sizes are different types. i.e. a pair (1,2) is a different type to a triple (1,2,3). Because of this, and lists only taking the same type, this will cause an error [(1,2),(1,2,3)]

**Tuple functions:** fst (1,2) Function only works on pairs, returns first element in a pair tuple. Example returns 1 snd(1,2) Function only works on pairs, returns second element in a pair tuple. Example returns 2

**Double check below……………..**

let (a,b) = (10,12) in a \* 2 returns 20

let (a:\_) = "xyz" in a retuns ‘x’ :\_ ignores the other values

let abc@(a,b,c) = (10,20,30) in (abc,a,b,c) returns ((10,20,30),10,20,30)