Haskell Quick Reference

GHCi

Some useful commands for the GHCi interpreter:

- **`:quit`** – exit
- **`:load`** *`file.hs`* – load contents of *`file.hs`*
- **`:reload`** – reload the last file (refresh after editing)
- **`:cd`** *`dir`* – switch to *`dir`* (if didn't launch GHCi in appropriate directory)
- **`:type`** *`expr`* – print the inferred type of *`expr`*

Primitive Types

- **`Bool`**: Boolean, values are **`True`**, **`False`**
- **`Int`**: 32-bit integer
- **`Integer`**: arbitrary-precision integer
- **`Char`**: character
- **`Float`**,**`Double`**: floating-point types
- **`List a`**: (aka **`[a]`**)
  - Contains arbitrary number of elements, all of the same type
  - **`[]`**: empty list
  - **`x:xs`**: cons cell
  - **`[x,y,z]`**: explicit list
  - **`[a,b] ++ [c,d]`**: concatenation
- **`(X,Y,Z)`**: *Tuple* – fixed number of elements, different types
  - **`(x,y,z)`**: an element
  - **`()`**: empty tuple; "void" type
- **`X -> Y -> Z`**: *Function* – a function from X and Y to Z
  - First-class in Haskell; can be stored in other data structures, passed to functions, etc.
  - Syntactic sugar for creating lambda expressions with binary operators:
    - **`(+)`** means **`(\a b -> a + b)`**
    - **`(x-)`** means **`(\a -> x - a)`**
    - **`(>x)`** means **`(\a -> a > x)`**
    - **`(-2)`** is -2, use **`(subtract 2)`** for the function

List Processing

- **`length lst`** – length of list
  `length [2,3] == 2`
- **`lst !! n`** – indexing operator (get n'th element; 0-indexed)
  `[2,3,5,7] !! 1 == 3`
- **`concat lstOfLst`** – flatten list of lists
  `concat [[3,4],[2],[9,6,8]] == [3,4,2,9,6,8]`

- **`head lst`** – first element of list
  `head [1,2,3] == 1`
- **`tail lst`** – all but first element of list
  `tail [1,2,3] == [2,3]`
- **`last lst`** – final element of list
  `last [1,2,3] == 3`
- **`init lst`** – all but final element of list
  `last [1,2,3] == [1,2]`
- **`replicate n x`** – list of n copies of x
  `replicate 3 'x' == ['x','x','x']`
- **`repeat x`** – infinite list of x's
  `repeat 42 == [42,42..]`
- **`take n lst`** – first n elements of list
  `take 2 [1,2,3,4] == [1,2]`
- **`drop n lst`** – all but first n elements of list
  `drop 2 [1,2,3,4] == [3,4]`
- **`splitAt n lst`** – split list into two lists at position n
  `splitAt 2 [1,2,3,4] == ([1,2],[3,4])`
- **`reverse lst`** – reverse list
  `reverse [1,2,3,4] == [4,3,2,1]`
- **`zip xs ys`** – "zip" two lists into list of pairs
  `zip [1,2,3] [1,4,9] == [(1,1),(2,4),(3,9)]`
- **`unzip lst`** – reverse zip
  `unzip [(1,1),(2,4),(3,9)] == ([1,2,3],[1,4,9])`
- **`and lst`** – lazy "and" of all elements in [Bool]
  `and [True,False,False] == False`
- **`or lst`** – lazy "or" of all elements in [Bool]
  `or [True,False,False] == True`
- **`sum lst`** – sum of numeric list
  `sum [9,16,25] == 50`
- **`product lst`** – product of numeric list
  `product [9,16,25] == 3600`
- **`minimum lst`** – minimum element of numeric list
  `minimum [9,16,25] == 9`
- **`maximum lst`** – maximum element of numeric list
  `maximum [9,16,25] == 25`

Higher-order Functions

- **`map f list`** – applies f to each element of list
  `map (^2) [3,0,-2,7] == [9,0,4,49]`
- **`filter p lst`** – keeps only elements of list matching predicate
  `filter (>7) [5,9,7,8] == [9,8]`

- **`all p lst`** – is predicate true for all elements of list?
  `all (>7) [5,9,7,8] == False`
- **`any p lst`** – is predicate true for any element of list?
  `any (>7) [5,9.7,8] == True`
- **`foldl f init lst`** – combines elements of list left-associatively using binary function f; init is used as starting value
  `foldl (-) 5 [3,8,9] == (((5-3)-8)-9) == -15`
  - Also **`foldl1`** that uses first element of list as init
- **`foldr f init lst`** – right-associative fold variant
  `foldr (-) 5 [3,8,9] == (3-(8-(9-5))) == -1`
  - Also **`foldr1`** that uses last element of list as init
- **`iterate f x`** – infinite list `[x,f(x),f(f(x)),…]`
  `iterate (*2) 1 == [1,2,4,8,16..]`
- **`takeWhile p lst`** – sequence at start of list for which p is true
  `takeWhile (<10) [1,1,2,3,5,8,13,21] == [1,1,2,3,5,8]`
- **`dropWhile p lst`** – all but `takeWhile`
  `dropWhile (<10) [1,1,2,3,5,8,13,21] == [13,21]`
- **`span p lst`** – tuple of `takeWhile`, `dropWhile` (breaks on p false)
  `span (<10) [1,1,2,3,5,8,13,21] == ([1,1,2,3,5,8],[13,21])`
- **`break p lst`** – `span (not p) lst` (breaks on p true)
  `break (\x -> (x `mod` 3) == 0) [1,1,2,3,5,8] == ([1,1,2,3],[5,8])`
- **`zipWith f xs ys`** – zip by binary function:
  `zipWith (-) [4,7,6,67] [2,6,2,45] == [2,1,4,22]`
- **`groupBy f xs`** – split list into groups of adjacent elements by predicate [*`Data.List`*]
  `groupBy (==) [1,1,2,3,2,2] == [[1,1],[2],[3],[2,2]]`
- **`sortBy cmp xs`** – sort list according to comparator [*`Data.List`*]
  `sortBy (compare `on` length) [[1],[],[2,2]] == [[],[1],[2,2]]`
  - **`compare :: Ord a => a -> a -> Ord`** – compares two elements, produces one of `LT`, `EQ`, `GT` (for less-than, equal-to, or greater-than, respectively)
    `compare 2 3 == LT`, `compare 3 3 == EQ`, `compare 4 3 == GT`
- **`on f g`** – equivalent to `(\x y -> f (g x) (g y))` [*`Data.Function`*]
  `groupBy ((==) `on` (`mod`2)) [0,2,1,3] == [[0,2],[1,3]]`
- **`flip f`** – take f's arguments in reverse order
  `(flip (++)) [2,4] [5,6] == [5,6,2,4]`
- **`(.)`** – function composition
  `(f . g) x == f (g x)`
- **`($)`** – function evaluation (low syntactic precedence)
  `f $ g $ h $ x == f (g (h x))`

Text-processing and I/O

- **`lines str`** – split input by lines
  `lines "foo\nbar\nbaz" == ["foo", "bar", "baz"]`

- **`unlines strs`** – transform list of strings into newline-delimited strings
  `unlines ["foo", "bar", "baz"] == "foo\nbar\nbaz"`
- **`words str`** – split input by whitespace
  `words "foo bar baz" == ["foo", "bar", "baz"]`
- **`unwords strs`** – joins input list with spaces
  `unwords ["foo", "bar", "baz"] == "foo bar baz"`
- **`show x`** – formatted string of type implementing `Show` class
  `show 47 == "47"`
- **`readMaybe str`** – failable parse of type implementing `Read` class [*`Text.Read`*]
  `readMaybe "47" :: Maybe Int == Just 47`
  `readMaybe "hello" :: Maybe Int == Nothing`
- **`read str`** – non-failable parse of type implementing Read class
  `read "47" :: Int == 47`
- **`getLine`** – reads a line from standard input (in IO context)
  `getLine -- returns input line in IO String`
- **`print x`** – sends any showable value to standard output, followed by a newline (in IO context)
  `print 42 -- prints "42"`
- **`readFile path`** – reads whole file from *path* (in IO context)
  `readFile "input.txt" -- returns contents of input.txt`
- **`writeFile path str`** – stores whole string in *path* (in IO context)
  `writeFile "output.txt" "hello, world!\n" -- stores hello, world!`