

```

1  -- Haskell is a functional programming language
2  -- Everything is immutable so once a value is set it is set forever
3  -- Functions can be passed as a parameter to other functions
4  -- Recursion is used often
5  -- Haskell has no for, while, or technically variables, but it does have
6  -- constants
7  -- Haskell is lazy in that it doesn't execute more than is needed and instead
8  -- just checks for errors
9
10 -- Best Free Haskell Book
11 -- http://learnyouahaskell.com/chapters
12
13 -- Type ghci to open it up in your terminal
14 -- Load script with :l haskelltut
15 -- :quit exits the GHCi
16
17 -- Import a module
18 import Data.List
19 import System.IO
20
21 {-
22 Beginning of multiline comment
23 -}
24
25 -- ----- DATA TYPES -----
26 -- Haskell uses type inference meaning it decides on the data type based on the -- value stored
27 -- Haskell is statically typed and can't switch type after compiling
28 -- Values can't be changed (Immutable)
29 -- You can use :t in the terminal to get the data type (:t value)
30
31 -- Int : Whole number -2^63 - 2^63
32 -- :: Int defines that maxInt is an Int
33 maxInt = maxBound :: Int
34 minInt = minBound :: Int
35
36 -- Integer : Unbounded whole number
37
38 -- Float : Single precision floating point number
39 -- Double : Double precision floating point number (11 pts precision)
40 bigFloat = 3.9999999999 + 0.00000000005
41
42 -- Bool : True or False
43 -- Char : Single unicode character denoted with single quotes
44 -- Tuple : Can store a list made up of many data types
45
46 -- You declare the permanent value of a variable like this
47 always5 :: Int
48 always5 = 5
49
50 -- ----- MATH -----
51 -- Something crazy to start
52 sumOfVals = sum [1..1000]
53
54 addEx = 5 + 4
55 subEx = 5 - 4
56 multEx = 5 * 4
57 divEx = 5 / 4
58
59 -- mod is a prefix operator
60 modEx = mod 5 4
61
62 -- With back ticks we can use it as an infix operator
63 modEx2 = 5 `mod` 4
64
65 -- Negative numbers must be surrounded with parentheses
66 negNumEx = 5 + (-4)
67
68 -- If you define an Int you must use fromIntegral to use it with sqrt
69 -- :t sqrt shows that it returns a floating point number

```

```

70 num9 = 9 ::Int
71 sqrtOf9 = sqrt (fromIntegral num9)
72
73 -- Built in math functions
74 piVal = pi
75 ePow9 = exp 9
76 logOf9 = log 9
77 squared9 = 9 ** 2
78 truncateVal = truncate 9.999
79 roundVal = round 9.999
80 ceilingVal = ceiling 9.999
81 floorVal = floor 9.999
82
83 -- Also sin, cos, tan, asin, atan, acos, sinh, tanh, cosh, asinh, atanh, acosh
84
85 trueAndFalse = True && False
86 trueOrFalse = True || False
87 notTrue = not(True)
88
89 -- Remember you use :t in the terminal to get the data type (:t value)
90 -- You can also see how functions use data types with :t
91
92 -- :t (+) = Num a => a -> a -> a
93 -- Type a is in the type class num, we receive 2 of them and return 1
94
95 -- :t truncate = (RealFrac a, Integral b) => a -> b
96
97 -- ----- LISTS -----
98 -- Lists are singly linked and you can only add to the front of it
99
100 -- Lists store many elements of the same type
101 primeNumbers = [3,5,7,11]
102
103 -- Concatenate lists (Can be slow if your using a large list)
104 morePrimes = primeNumbers ++ [13,17,19,23,29]
105
106 -- You can use the cons operator to construct a list
107 favNums = 2 : 7 : 21 : 66 : []
108
109 -- You can make a list of lists
110 multList = [[3,5,7],[11,13,17]]
111
112 -- Quick way to add 1 value to the front of a list
113 morePrimes2 = 2 : morePrimes
114
115 -- Get number of elements in the list
116 lenPrime = length morePrimes2
117
118 -- Reverse the list
119 revPrime = reverse morePrimes2
120
121 -- return True if list is empty
122 isEmptyList = null morePrimes2
123
124 -- Get the number in index 1
125 secondPrime = morePrimes2 !! 1
126
127 -- Gets the 1st value in a list
128 firstPrime = head morePrimes2
129
130 -- Gets the last value
131 lastPrime = last morePrimes2
132
133 -- Gets everything but the first value
134 primeTail = tail morePrimes2
135
136 -- Gets everything but the last value
137 primeInit = init morePrimes2
138

```

```

139 -- Get specified number of elements from the front of a list
140 first3Primes = take 3 morePrimes2
141
142 -- Return values left after removing specified values
143 removedPrimes = drop 3 morePrimes2
144
145 -- Check if value is in list
146 is7InList = 7 `elem` morePrimes2
147
148 -- Get max value
149 maxPrime = maximum morePrimes2
150
151 -- Get minimum value
152 minPrime = minimum morePrimes2
153
154 -- Sum values in list
155 sumPrimes = sum morePrimes2
156
157 -- Get product of values in list (Value all can evenly divide by)
158 newList = [2,3,5]
159 prodPrimes = product newList
160
161 -- Create list from 0 to 10
162 zeroToTen = [0..10]
163
164 -- Create list of evens by defining the step between the first 2 values
165 evenList = [2,4..20]
166
167 -- You can use letters as well
168 letterList = ['A','C'..'Z']
169
170 -- You can generate an infinite list and Haskell will only generate what you
171 -- need
172 infinPow10 = [10,20..]
173
174 -- repeat repeats a value a defined number of times
175 many2s = take 10 (repeat 2)
176
177 -- replicate generates a value a specified number of times
178 many3s = replicate 10 3
179
180 -- cycle replicates the values in a list indefinitely
181 cycleList = take 10 (cycle [1,2,3,4,5])
182
183 -- You could perform operations on all values in a list
184 -- Cycle through the list storing each value in x which is multiplied by 2 and
185 -- then stored in a new list
186 listTimes2 = [x * 2 | x <- [1..10]]
187
188 -- We can filter the results with conditions
189 listTimes3 = [x * 3 | x <- [1..20], x*3 <= 50]
190
191 -- Return all values that are divisible by 13 and 9
192 divisBy9N13 = [x | x <- [1..500], x `mod` 13 == 0, x `mod` 9 == 0]
193
194 -- Sort a list
195 sortedList = sort [9,1,8,3,4,7,6]
196
197 -- zipWith can combine lists using a function
198 sumOfLists = zipWith (+) [1,2,3,4,5] [6,7,8,9,10]
199
200 -- Filter returns a list of items that match a condition
201 listBiggerThen5 = filter (>5) sumOfLists
202
203 -- takeWhile returns list items until the condition is false
204 evensUpTo20 = takeWhile (<=20) [2,4..]
205
206 -- foldl applies the operation on each item of a list
207 -- foldr applies these operations from the right

```

```

208 multOfList = foldl (*) 1 [2,3,4,5]
209
210 ----- LIST COMPREHENSION -----
211
212 -- We can generate a list from 1 to 10 to the power of 3
213 pow3List = [3^n | n <- [1..10]]
214
215 -- We can filter the results to only show values divisible by 9
216 pow3ListDiv9 = [3^n | n <- [1..10], 3^n `mod` 9 == 0]
217
218 -- Generate a multiplication table by multiplying x * y where y has the values
219 -- 1 through 10 and where x does as well
220 multTable = [[x * y | y <- [1..10]] | x <- [1..10]]
221
222 ----- TUPLES -----
223 -- Stores list of multiple data types, but has a fixed size
224
225 randTuple = (1,"Random tuple")
226
227 -- A tuple pair stores 2 values
228 bobSmith = ("Bob Smith",52)
229
230 -- Get the first value
231 bobsName = fst bobSmith
232
233 -- Get the second value
234 bobsAge = snd bobSmith
235
236 -- zip can combine values into tuple pairs
237 names = ["Bob","Mary","Tom"]
238 addresses = ["123 Main","234 North","567 South"]
239
240 namesNAddress = zip names addresses
241
242 ----- FUNCTIONS -----
243 -- ghc --make haskell tut compiles your program and executes the main function
244
245 -- Functions must start with lowercase letters
246
247 -- We can define functions and values in the GHCi with let
248 -- let num7 = 7
249 -- let getTriple x = x * 3
250
251 -- getTriple num7 = 21
252
253 -- main is a function that can be called in the terminal with main
254 main = do
255     -- Prints the string with a new line
256     putStrLn "What's your name: "
257
258     -- Gets user input and stores it in name
259     -- <- Pulls the name entered from an IO action
260     name <- getLine
261
262     putStrLn ("Hello " ++ name)
263
264 -- Create function addMe
265 -- x is a parameter and the operation follows the equals sign
266 -- The data type passed in will work if it makes sense
267 -- Every function must return something
268 -- A function name can't begin with a capital letter
269 -- A function that doesn't receive parameters is called a definition or name
270
271 -- You can define a type declaration for functions
272 -- funcName :: param1 -> param2 -> returnType
273 addMe :: Int -> Int -> Int
274
275 -- funcName param1 param2 = operations (Returned Value)
276 -- Execute with : addMe 4 5

```

```

277 addMe x y = x + y
278
279 -- Without type declaration you can add floats as well
280 sumMe x y = x + y
281
282 -- You can also add tuples : addTuples (1,2) (3,4) = (4,6)
283 addTuples :: (Int, Int) -> (Int, Int) -> (Int, Int)
284 addTuples (x, y) (x2, y2) = (x + x2, y + y2)
285
286 -- You can perform different actions based on values
287 whatAge :: Int -> String
288 whatAge 16 = "You can drive"
289 whatAge 18 = "You can vote"
290 whatAge 21 = "You're an adult"
291
292 -- The default
293 whatAge x = "Nothing Important"
294
295 -- Define that we expect an Int in and out
296 factorial :: Int -> Int
297
298 -- If 0 return a 1 (Recursive Function)
299 factorial 0 = 1
300 factorial n = n * factorial (n - 1)
301
302 -- 3 * factorial (2) : 6
303 -- 2 * factorial (1) : 2
304 -- 1 * factorial (0) : 1
305
306 -- You could also use product to calculate factorial
307 productFactorial n = product [1..n]
308
309 -- We can use guards that provide different actions based on conditions
310 isOdd :: Int -> Bool
311 isOdd n
312     -- if the modulus using 2 equals 0 return False
313     | n `mod` 2 == 0 = False
314
315     -- Else return True
316     | otherwise = True
317
318 -- This could be shortened to
319 isEven n = n `mod` 2 == 0
320
321 -- Use guards to define the school to output
322 whatGrade :: Int -> String
323 whatGrade age
324     | (age >= 5) && (age <= 6) = "Kindergarten"
325     | (age > 6) && (age <= 10) = "Elementary School"
326     | (age > 10) && (age <= 14) = "Middle School"
327     | (age > 14) && (age <= 18) = "High School"
328     | otherwise = "Go to college"
329
330 -- The where clause keeps us from having to repeat a calculation
331 batAvgRating :: Double -> Double -> String
332 batAvgRating hits atBats
333     | avg <= 0.200 = "Terrible Batting Average"
334     | avg <= 0.250 = "Average Player"
335     | avg <= 0.280 = "Your doing pretty good"
336     | otherwise = "You're a Superstar"
337     where avg = hits / atBats
338
339 -- You can access list items by separating letters with : or get everything but
340 -- the first item with xs
341 getListItems :: [Int] -> String
342 getListItems [] = "Your list is empty"
343 getListItems (x:[]) = "Your list contains " ++ show x
344 getListItems (x:y:[]) = "Your list contains " ++ show x ++ " and " ++ show y
345 getListItems (x:xs) = "The first item is " ++ show x ++ " and the rest are "

```

```

346     ++ show xs
347
348 -- We can also get values with an As pattern
349 getFirstItem :: String -> String
350 getFirstItem [] = "Empty String"
351 getFirstItem all@(x:xs) = "The first letter in " ++ all ++ " is "
352     ++ [x]
353
354 -- ----- HIGHER ORDER FUNCTIONS -----
355 -- Passing of functions as if they are variables
356
357 times4 :: Int -> Int
358 times4 x = x * 4
359
360 -- map applies a function to every item in the list
361 listTimes4 = map times4 [1,2,3,4,5]
362
363 -- Let's make map
364 multBy4 :: [Int] -> [Int]
365 multBy4 [] = []
366
367 -- Takes the 1st value off the list x, multiplies it by 4 and stores it in the
368 -- new list
369 -- xs is then passed back into multBy4 until there is nothing left of the list -- to process (
370 multBy4 (x:xs) = times4 x : multBy4 xs
371
372 -- Check if strings are equal with recursion
373 areStringsEq :: [Char] -> [Char] -> Bool
374 areStringsEq [] [] = True
375 areStringsEq (x:xs) (y:ys) = x == y && areStringsEq xs ys
376 areStringsEq _ _ = False
377
378 -- PASSING A FUNCTION INTO A FUNCTION
379 -- (Int -> Int) says we expect a function that receives an Int and returns an
380 -- Int
381 doMult :: (Int -> Int) -> Int
382
383 -- We receive the function and pass 3 into it
384 doMult func = func 3
385
386 -- We pass in the function that multiplies by 4
387 num3Times4 = doMult times4
388
389 -- RETURNING A FUNCTION FROM A FUNCTION
390 getAddFunc :: Int -> (Int -> Int)
391
392 -- We can pass in the values to the function
393 getAddFunc x y = x + y
394
395 -- We could also get a function that adds 3 for example
396 adds3 = getAddFunc 3
397
398 fourPlus3 = adds3 4
399
400 -- We could use this function with map as well
401 threePlusList = map adds3 [1,2,3,4,5]
402
403 -- ----- LAMBDA -----
404 -- How we create functions without a name
405 -- \ represents lambda then you have the arguments -> and result
406
407 dbl11To10 = map (\x -> x * 2) [1..10]
408
409 -- ----- CONDITIONALS -----
410
411 -- Comparison Operators : < > <= >= == /=
412 -- Logical Operators : && || not
413
414 -- Every if statement must contain an else

```

```

415 doubleEvenNumber y =
416     if (y `mod` 2 /= 0)
417         then y
418         else y * 2
419
420 -- We can use case statements
421 getClass :: Int -> String
422 getClass n = case n of
423     5 -> "Go to Kindergarten"
424     6 -> "Go to elementary school"
425     _ -> "Go some place else"
426
427 ----- MODULES -----
428 -- You can group functions into modules. I showed previously how to load them
429 -- You can create your own module by creating a file that contains all your
430 -- functions and then list the functions at the top like this
431 -- module SampFunctions (getClass, doubleEvenNumber) where
432 -- They can then be imported with import SampFunctions
433
434 ----- ENUMERATION TYPES -----
435 -- Used when you want a list of possible types
436 -- Provide name, a list and then Show converts into a String for printing
437
438 data BaseballPlayer = Pitcher
439                     | Catcher
440                     | Infield
441                     | Outfield
442                     deriving Show
443
444 barryBonds :: BaseballPlayer -> Bool
445 barryBonds Outfield = True
446
447 barryInOF = print(barryBonds Outfield)
448
449 ----- CUSTOM TYPES -----
450 -- You can store multiple values sort of like a struct to create custom types
451 data Customer = Customer String String Double
452               deriving Show
453
454 -- Define Customer and its values
455 tomSmith :: Customer
456 tomSmith = Customer "Tom Smith" "123 Main St" 20.50
457
458 -- Define how we'll find the right customer (By Customer) and the return value
459 getBalance :: Customer -> Double
460 getBalance (Customer _ _ b) = b
461
462 tomSmithBal = print (getBalance tomSmith)
463
464 -- We can define a type with all possible values
465 data RPS = Rock | Paper | Scissors
466
467 shoot :: RPS -> RPS -> String
468 shoot Paper Rock = "Paper Beats Rock"
469 shoot Rock Scissors = "Rock Beats Scissors"
470 shoot Scissors Paper = "Scissors Beat Paper"
471 shoot Scissors Rock = "Scissors Loses to Rock"
472 shoot Paper Scissors = "Paper Loses to Scissors"
473 shoot Rock Paper = "Rock Loses to Paper"
474 shoot _ _ = "Error"
475
476 -- We could define 2 versions of a type
477 -- First 2 floats are center coordinates and then radius for Circle
478 -- First 2 floats are for upper left hand corner and bottom right hand corner
479 -- for the Rectangle
480 data Shape = Circle Float Float Float | Rectangle Float Float Float Float
481            deriving (Show)
482
483 -- :t Circle = Float -> Float -> Float -> Shape

```

```

484
485 -- Create a function to calculate area of shapes
486 area :: Shape -> Float
487 area (Circle _ _ r) = pi * r ^ 2
488 area (Rectangle x y x2 y2) = (abs (x2 - x)) * (abs (y2 - y))
489
490 -- Could also be area (Rectangle x y x2 y2) = (abs $ x2 - x) * (abs $ y2 - y)
491 -- $ means that anything that comes after it will take precedence over anything
492 -- that comes before (Alternative to adding parentheses)
493
494 -- The . operator allows you to chain functions to pass output on the right to
495 -- the input on the left
496 -- sumValue = putStrLn (show (1 + 2)) becomes
497 sumValue = putStrLn . show $ 1 + 2
498
499 -- Get area of shapes
500 areaOfCircle = area (Circle 50 60 20)
501 areaOfRectangle = area $ Rectangle 10 10 100 100
502
503 ----- TYPE CLASSES -----
504 -- Num, Eq, Ord and Show are type classes
505 -- Type classes correspond to sets of types which have certain operations
506 -- defined for them.
507 -- Polymorphic functions, which work with multiple parameter types, define
508 -- the types it works with through the use of type classes
509 -- For example (+) works with parameters of the type Num
510 -- :t (+) = Num a => a -> a -> a
511 -- This says that for any type a, as long as a is an instance of Num, + can take
512 -- 2 values and return an a of type Num
513
514 -- Create an Employee and add the ability to check if they are equal
515 data Employee = Employee { name :: String,
516                             position :: String,
517                             idNum :: Int
518                           } deriving (Eq, Show)
519
520 samSmith = Employee {name = "Sam Smith", position = "Manager", idNum = 1000}
521 pamMarx = Employee {name = "Pam Marx", position = "Sales", idNum = 1001}
522
523 isSamPam = samSmith == pamMarx
524
525 -- We can print out data because of show
526 samSmithData = show samSmith
527
528 -- Make a type instance of the typeclass Eq and Show
529 data ShirtSize = S | M | L
530
531 instance Eq ShirtSize where
532     S == S = True
533     M == M = True
534     L == L = True
535     _ == _ = False
536
537 instance Show ShirtSize where
538     show S = "Small"
539     show M = "Medium"
540     show L = "Large"
541
542 -- Check if S is in the list
543 smallAvail = S `elem` [S, M, L]
544
545 -- Get string value for ShirtSize
546 theSize = show S
547
548 -- Define a custom typeclass that checks for equality
549 -- a represents any type that implements the function areEqual
550 class MyEq a where
551     areEqual :: a -> a -> Bool
552

```



```

553 -- Allow Booleans to check for equality using areEqual
554 instance MyEq ShirtSize where
555     areEqual S S = True
556     areEqual M M = True
557     areEqual L L = True
558     areEqual _ _ = False
559
560 newSize = areEqual M M
561
562 -- ----- I/O -----
563
564 sayHello = do
565     -- Prints the string with a new line
566     putStrLn "What's your name: "
567
568     -- Gets user input and stores it in name
569     name <- getLine
570
571     -- $ is used instead of the parentheses
572     putStrLn $ "Hello " ++ name
573
574 -- File IO
575 -- Write to a file
576 writeFile = do
577
578     -- Open the file using WriteMode
579     theFile <- openFile "test.txt" WriteMode
580
581     -- Put the text in the file
582     hPutStrLn theFile ("Random line of text")
583
584     -- Close the file
585     hClose theFile
586
587 readFromFile = do
588
589     -- Open the file using ReadMode
590     theFile2 <- openFile "test.txt" ReadMode
591
592     -- Get the contents of the file
593     contents <- hGetContents theFile2
594     putStr contents
595
596     -- Close the file
597     hClose theFile2
598
599 -- ----- EXAMPLE : FIBONACCI SEQUENCE -----
600
601 -- Calculate the Fibonacci Sequence
602 -- 1, 1, 2, 3, 5, 8, ...
603
604 -- 1 : 1 : says to add 2 1s to the beginning of a list
605 -- | for every (a, b) add them
606 -- <- stores a 2 value tuple in a and b
607 -- tail : get all list items minus the first
608 -- zip creates pairs using the contents from 2 lists being the lists fib and the
609 -- list (tail fib)
610
611 fib = 1 : 1 : [a + b | (a, b) <- zip fib (tail fib) ]
612
613 -- First time through fib = 1 and (tail fib) = 1
614 -- The list is now [1, 1, 2] because a: 1 + b: 1 = 2
615
616 -- The second time through fib = 1 and (tail fib) = 2
617 -- The list is now [1, 1, 2, 3] because a: 1 + b: 2 = 3
618
619 fib300 = fib !! 300 -- Gets the value stored in index 300 of the list
620
621 -- take 20 fib returns the first 20 Fibonacci numbers

```