Haskell

GHCi

- GHCi is GHC's interactive environment that includes an interactive debugger
- Read Evaluate Print Loop(REPL): interprets the whole line as an expression to evaluate

Commands

$ `ghci` : open GHCi

$ `:?` : the list of commands available

$ `:l file.hs` : load files

$ `:r` : reload files

$ `:{ ~ :}` : multi-line functions

$ `:t function` : returns the type

$ `:fst` : return the first element of length 2 tuple

$ `:scd` : return the second element of length 2 tuple

`show val` : print value in a string

Functional Programming

About

- All functions must take at least one argument and return a value
  - ex. y = f(x)
- All variables are immutable and can not be modified after initialization
  - + Thread safe: no race condition, no need for synchronization
  - + Easier to debug,
  - + Cache-friendly: caches do not need to be frequently updated or invalidated, improving cache hit rates
- All functions are pure and have no side effects
  - calling f(x) with a given x always returns the same value y
  - functions may not use or modify external mutable state/variables
- Functions are just like any other data and can be stored in variables and passed as arguments
  - ex. f(x) = x + 5, y = f, z = g(f)
- The order of execution is of low importance!

Features

- Lazy Evaluation: only evaluates expressions in the order their values are needed
- Left-associative: evaluates function calls from left to right
  - ex. `f g 2` ⇒ `((f g) 2)` ≠ `f (g 2)`

- Always call functions before evaluating operators
  - ex. `f 3 * 10` ⇒ `(f 3)*10`
- Only the IO function can print because printing changes the state of the program's environment

```
main :: IO ()
main = do
  let result = poly 5
  putStrLn ("Result: " ++ show result)
```

Types

Basic Types

- Type Signature
  - `func_name :: param_type -> return_type`
  - `func_name :: param1_type -> param2_type -> return_type`
  - Specify the function's type signature first
  - The last type in the line refers to the return type
  - Type Inference: type is inferred if it's not specified, but it's still type strict
- Types
  - `Int`: machine-sized integer type (32 or 64 bits)
  - `Integer`: a superset of Int - no size limit
  - `Float`: 32-bit floating-point number
  - `Double`: 64-bit floating-point number
  - `Bool`: `True` / `False`
  - `Char`
  - `String` = `[Char]`

Operator

- Infix vs Prefix Notation
  - `x = 3 * x^2` ⇒ `x = (*) 3 x^2`
  - `a == b` ⇒ `(==) a b`
  - `6 - 5` ⇒ `(-) 6 5`
- `/=`: not
- `div`: int division
- `^`: exponential
- Methods
  - `max n1 n2`
  - `min n1 n2`

Expressions

If Expressions

- `if cond then statement else statement`
- In functional programming, "if" is an expression, not a statement!
  - every expression must result in a value
  - so without an "else", "if" would fail being an expression

Guards

- Guards: an alternative to multiple conditions
  - evaluated from top to bottom
  - no equal sign after the function signature!
  - a syntactic sugar to provide simpler syntax

```
func param1 param2
  | condition_1 = action_1 -- if
  | condition_2 = action_2 -- if else
  ...
  | otherwise = action_n -- else
```

Local Bindings

- Local Bindings: define local variables or functions
  - able to define nested functions
- **Let** expression
  - `let binding in expr`
  - can be used anywhere an expression can appear
    - ex. `result = let x = 5 y = 10 in x + y`
  - scoped only to the expression in which it is defined

```
get_nerd_status gpa study_hrs =
  let
    gpa_part = 1 / (4.01 - gpa)
    study_part = study_hrs * 10
    nerd_score = gpa_part + study_part
  in
    if nerd_score > 100 then "good"
    else "bad"
```

- **Where** construct
  - similar to `let` but locates at the end of the function
  - must follow a function definition or guarded expression
  - scoped to the entire function or guarded expressions preceding it

```
get_nerd_status gpa study_hrs =
  if nerd_score > 100 then "good"
  else "bad"
  where
    gpa_part = 1 / (4.01 - gpa)
    study_part = study_hrs * 10
    nerd_score = gpa_part + study_part
```

- `let` vs `where` ?
  - Defining a variable for a single expression, use either one
  - Defining a variable for use across multiple expressions (like guards), use `where`

Data Type

Tuples

- `name = ("str", 1, 3.14)`
- group two or more items of different types into one value in fixed-size
  - ! length of the tuple is 1
- Operations

`fst two_field_tup` : return the first element

`snd two_field_tup` : return the second element

Lists

- `name = [1, 2, 3]`
- hold zero or more items of the same type

```
primes :: [Int]
primes = [2, 3, 5]

lol :: [[Int]]
lol = [[1, 2], [3], [4, 5, 6]]

lot :: [(String, Int)]
lot = [("foo", 1), ("bar", 2)]

mt = []
```

- Operations

  `length ls`

  `head ls` : return the first element of a non-empty list

  `tail ls` : returns all elements of a list except the first one

  `take N ls` : take the first N elements from list (same as `ls[:N]` in Python)

  `drop N ls` : drop the first N elements from list and return the rest (same as `ls[N:]` in Python)

  `init ls` : return all the elements of the list except the last one

  `last ls` : return the last element of the list

  `elem el ls` : check if an element is in the list = `el `elem` ls`

  `sum ls` :

  `maximum ls` : return a max element

  `minimum ls` :

  `or ls` : defaults to False

  `and ls` : defaults to True (ex. `and []` : True)

  `cycle ls` : return an infinite cycle of the list

  `reverse ls` : reverse the list

  `zip ls1 ls2` : pairs two lists and takes the shortest list

    ex. `zip [1, 2] ["A", "B", "C"]` ⇒ [(1, "A"), (2, "B")]

  `all cond ls` : return True if the condition is met for all elements in the list

    ex. `all (odd) [1, 2, 3]` , `all (< 3) [1, 2, 3]`

  `ls1 ++ ls2` : list concatenation

    ex. `reverse_list ls = (reverse_list (tail ls)) ++ [head ls]`

  `el : ls` : consing; prepend a single element from right to left

    ex. `"a" : "b" : []` : [”a”, “b”]

- Ranges
  - `[start..end]`
  - `[start, step..end]` : step is defined as `step - start`
  - ranges are inclusive
  - Infinite range: `[start..]`
    - 🧯 printing an infinite list will hang in an infinite loop! ⇒ should use `take N [start..]` to get a subset

```
one_to_ten = [1..10]
```

```
one_to_ten = [1..10]
odds = [1, 3..10] -- [1, 3, 5, 7, 9]
take 5 [42..] -- take 5 el of inifinit array
cycle [1, 3, 5] -- [1, 3, 5, 1, 3, 5, ...]
```

- List Comprehension
  - `output = [(f x) | x <- input, (guard1 x), (guard2, x), ...]`
    - identical to Set Builder Notation: { x^2 | x ∈ input, x > 5, x < 20 }
  - `output = [(f x y) | x <- input1, y <- input2, (guard1 x), (guard2, y), ...]`
    - works as a nested loop
    - ex. `prods = [x*y | x <- [3, 7], y <- [2, 4, 6]]` ⇒ [6, 12, 18, 14, 28, 42]

```
square_nums1 = [x^2 | x <- input_list]
square_nums2 = [x^2 | x <- input_list, x > 5]
square_nums3 = [x^2 | x <- input_list, x > 5, x < 20]

all_products1 = [x*y | x <- list1, y <- list2]
all_products2 = [x*y | x <- list1, y <- list2, even x, odd y, x*y < 15]

div_by_29 = take 5 [x | x <- [1000..], (mod x 29) == 0]

lies = "nerd EAT rockS!"
truth = [c | c <- lies, not (elem c ['A'..'Z'])]

-- Quicksort
qsort [] = []
qsort ls = (qsort less_eq) ++ [pivot] ++ (qsort greater)
  where
    pivot = head ls
    rest = tail ls
    less_eq = [a | a <- rest, a <= pivot]
    greater = [a | a <- rest, a > pivot]
```

  - ! list of functions using list comprehension

```
inf_list = [\x -> x^z | z <- [1..]] -- infinite list of functions

first_5_funcs = take 5 inf_list
results = map (\f -> f 2) first_5_funcs -- [2, 4, 8, 16, 32]; apply the first 5 functions to the numbe

head inf_list 9 -- 9
```

Generic Types

- `t` : type variable (can be any char)
  - ex. `func :: [t] -> t` : arg and return type has to be matched
  - ex. `func :: t1 -> t2 -> (t2, t1)` : t1 and t2 being the same type is fine

Type classes

- `Ord` : orderable(comparable) type such as number and string
- `Fractional`

```
bigger x y = if x > y then x else y
:t bigger
-- bigger :: Ord a => a -> a -> a

division x = x /5
:t division
-- Fractional a => a -> a
```

Algebraic Data Types

- `data Name = Variant1 | Variant2 | ...`
  - combination of structs, unions, and enum types
  - ADT type name and variant name must start with an uppercase
  - Every time you define a variant, Haskell implicitly creates a new constructor
- `deriving Show`: required for GHCi

```
data Color = Red | Green | Blue

data Shape =
  Circle { radius :: Float, color :: Color } | -- same as `Circle Float Color`
  Rectangle { width :: Float, height :: Float, color :: Color } |
  Shapeless -- nullary variant
  deriving Show

my_circ = Circle { radius = 5.0, color = Blue } -- same as `Circle 5.0 Blue`
```

- ADT with Pattern Matching

```
-- a single function with 4 branches
getArea :: Shape -> Float
getArea (Circle r _) = pi * r^2
getArea (Rectangle w h c) = w * h
getArea (Shapeless) = 0

describe :: Shape -> String
describe (Circle 1 _) = "unit circle"
describe (Circle r Red) = "red circle w/ radius " ++ show r
```

- Linked List using ADT

```
data List =
  Nil |
  Node {
    val :: Integer,
    next :: List
  }
  deriving Show

list1 = Nil
list2 = Node 5 list1 -- Node {val = 5, next = Nil}
l_head = Node 3 (Node 2 (Node 1 Nil))

sumList :: List -> Integer
sumList (Nil) = 0
sumList (Node val next) = val + sumList next
```

❗ Immutability

- To append even a single item at the end of a linked list, you must construct a whole new list!
  - To add one node at position P in a linked list of length N, P+1 nodes have to be newly created, and N-P nodes can be reused
- To add a new node in a balanced BST, we need to generate replacement nodes for the nodes between our new node and the root
  - just need to create O(log n) new nodes!
  - garbage collection reclaims unused values and all other nodes are reused!

Functions

Pattern Matching

- Pattern Matching: define multiple versions of the same function
  - each version of the function must have the same number and types of arguments
  - runs top-to-bottom

```
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * facotrial (n-1)
```

- Pattern matching with Lists
  - `(first:rest)`, `(first:second:rest)`, `...` : get a list as an argument
    - ex. `func (x:[])` : get called only if the arg has only one element, `func (x:xs)`
  - `ls@(first:rest)` : as-pattern; a reference to the entire value

```
fav [] = "no fav"
fav (x:[]) = "fav: " ++ x
fav (x:y:[]) = "two favs: " ++ x ++ "and" ++ y
fav ("apple":xs) = "fav apple"
fav (x:y:_) = "many favs"
```

✨ Preferred: Pattern matching > Guards > If/Then/Else

Tail Recursion

- Tail Recursion: a type of recursion where the recursive call is the final operation in the function, meaning no additional computation occurs after the recursive call returns
  - optimize the recursive function by reusing the same stack frame for each recursive step
  - can handle deeper recursive calls without running into stack overflow issues, making it more efficient in terms of memory usage compared to non-tail-recursive functions

```
factorialTail :: Integer -> Integer
factorialTail n = helper n 1
  where
    helper 0 acc = acc
    helper n acc = helper (n - 1) (acc * n)
```

▼ ! Fibonacci in O(n)

```
fib n = _fib n 0 1 -- n: how many numbers left to compute
  where
    _fib 0 a _ = a
    _fib n a b = _fib (n-1) b (a+b)
```

Fibonacci into list

```
fibonacci :: Int -> [Int]
fibonacci n = _fibonacci 1 1 n
 where
   _fibonacci :: Int -> Int -> Int -> [Int]
   _fibonacci a b count
     | count <= 0 = []
     | otherwise = a : _fibonacci b (a + b) (count - 1)
```

First-class Functions

- First-class Functions: a function is treated like any other data
  - functions can be stored in variables, passed as arguments to other functions, returned as values by functions, and stored in data structures
  - enable functions to generate new functions from scratch

Higher-order functions

- Higher-order functions: accepts another function as an argument or a function that returns another function as its return value
- Passing functions as arguments

```
insult :: String -> String
insult name = name ++ "is cringe"

praise :: String -> String
praise name = name ++ "is dank"

talk_to :: String -> (String -> String) -> String
talk_to name talk_func
  | name == "Carey" = "no commnet"
  | otherwise = (talk_func name)

-- $ talk_to "Devan" prasie
```

- Returning functions from other functions

```
get_pickup_func :: Int -> (String -> String)
get_pickup_func born
  | bron >= 1997 = pickup_genz
  | otherwise = pickup_other
  where
    pickup_genz name = name ++ "genz"
    pickup_other name = name ++ "other"

pickup_fn = get_pickup_func 2003
pickup_fn "Jay" -- "Jay genz"
-- same as..
get_pickup_func 1971 "Carey" -- "Carey other"
```

- Mapper: `map :: (a -> b) -> [a] -> [b]`

```
cube :: Double -> Double
cube x = x^3
map cube [2, 4, 10]

data Tree = Empty | Node Integer [Tree]
max_tree_value Empty = 0
max_tree_value (Node v []) = v
max_tree_value (Node value child) = max value (maximum (map max_tree_value child))
```

- Filter: `filter :: (a -> Bool) -> [a] -> [a]`
- Reducer
  - `foldl :: (a -> b -> a) -> a -> [b] -> a`: process from left to right
    - `foldl (\acc el -> expr) init ls`
  - `foldr :: (a -> b -> b) -> b -> [a] -> b`: process from right to left
    - `foldr (\el acc -> expr) init ls`

```
adder acc x = acc + x
foldl adder 0 [7, -4, 2]
```

```
get_every_nth n ls = foldl (\acc (i, x) -> if i `mod` n == 0 then acc ++ [x] else acc) [] (zip [1..] ls)

longest_run :: [Bool] -> Int -- count the longest consecutive True value
longest_run ls = fst (foldl (\(m, curr) x -> if x then ((max (curr+1) m), curr+1) else (m, 0)) (0, 0) ls)
```

Lambda Functions

- use lambdas in higher-order function
- `\param_1 ... param_n -> expression`
  - ex. `(\x y -> x^3 + y^2) 10 3`
- **Closure**: the combination of a function that we want to run in the future with a list of free variables and their captured values at the time it was created
  - **Free variable**: any variable that is not an explicit parameter to the lambda
  - variables are captured and stored along with the lambda so the lambda knows what values to use in later

```
slope m b = (\x -> m*x + b) -- free variables are m and b, not x since its value is passed into the lambda
twoPlusOne = slope 2 1
-- twoPlusOne closure: {m: 2, b: 1, function: \x -> m*x + b}
twoPlusOne 9 -- uses the captured values in the closure
```

- Hiding

```
foo x y = (\z -> x + y - z)
foo 5 6 7 -- foo 5 6 (\z -> 5 + 6 - z) = (\z -> 5 + 6 - z) 7 = 4

bar x y = (\x -> x + y) -- y is the only free variable
bar 5 6 7
-- bar 5 6 (\x -> x + 6) = (\x -> x + 6) 7 = 13
```

Partial Function

- Idea: when a function is called with less than the full number of args, the function returns a new version of itself that takes just the missing parameters
  - ex. f(x, y, z) = x + y + z ⇒ f′ = f(10, 20) ⇒ f′(z) = 10 + 20 + z

```
product_of x y z = x * y * z

product_5 = product_of 5
product_5 2 3 --  30

product_5_6 = product_of 5 6
product_5_6 2 -- 60

cuber = map (\x -> x^3) -- since map takes in two arguments, this creates a new mapper function that always cu
cuber [2, 3, 5]
```

- **Currying**
  - transforms a function of multiple arguments to a series of nested functions which each takes a single argument
  - every function with two or more parameters can be represented in curried form
    - ex. y = f(a, b, c) ⇒ y = ((f_1 a) b) c ⇒ y = f(a)(b)(c)
  - all the functions with more than one parameter are automatically curried in Haskell
    - ex. `((mult 2) 3) 5)` = `mult 2 3 5`

```
mult3 :: Int -> Int -> Int -> Int
mult3 x y z = x * y * z
```

```
mult3 :: Int -> (Int -> (Int -> Int))
mult3 = \x -> (\y -> (\z -> x * y * z))

f = mult3 5 -- returns closures; f = (\y -> (\z -> 5 * y * z))
```

! more...

`addFive = (+ 5)` = `addFive a = a + 5`

`f = (\g h i -> i h g) ["Carey"] [131]` : g = [”Carey”], h = [131]