Principles of Programming Languages (19CSE313)

Semester: VI

Unit-1
Sample Haskel Programs

Prepared By:

Dr. Amulyashree S
Assistant Professor,
Amrita School of Computing,
Amrita Vishwa Vidyapeetham,
Bengaluru Campus

Dr. Daddala Yasoomkari
Assistant Professor,
Amrita School of Computing,
Amrita Vishwa Vidyapeetham,
Bengaluru Campus

Sample Programs in Haskell

General Instructions:

- Save the program with extension ".hs" to indicate that it is a Haskell file

Program-1: print hello world

```
module Main (main) where          -- not needed in interpreter, is the default in a module file

main :: IO ()                -- the compiler can infer this type definition
main = putStrLn "Hello, World!"
```

Program-2
Write a Haskell function to check if a year is leap year or not

Note: The key to determining whether a given year is a leap year is to know whether the year is evenly divisible by 4, 100, and 400.

Approach-1: Using logical expression

```
isLeapYear :: Integer -> Bool
isLeapYear year = divisibleBy 4 && (not (divisibleBy 100) ||
divisibleBy 400)
  where
    divisibleBy d = year `mod` d == 0
```

Approach 2: use guards

```
isLeapYear :: Integer -> Bool
isLeapYear year
  | indivisibleBy 4   = False
  | indivisibleBy 100 = True
  | indivisibleBy 400 = False
  | otherwise         = True
  where
    indivisibleBy d = year `mod` d /= 0
```

Approach 3: conditional expression

```
isLeapYear :: Integer -> Bool
isLeapYear year =
  if divisibleBy 100
    then divisibleBy 400
    else divisibleBy 4
  where
    divisibleBy d = year `mod` d == 0
```

Program-3
Write a Haskell function to reverse a string

Approach-1

```
module ReverseString (reverseString) where
reverseString :: String -> String
reverseString [] = []
reverseString (x:xs) = reverseString xs ++ [x]
```

Approach-2

```
module ReverseString (reverseString) where
reverseString :: String -> String
reverseString = foldl (flip (:)) []
```

Program-4
Write a Haskell function that returns the earned points in a single toss of a Darts game.

Darts is a game where players throw darts at a target.

In our instance of the game, the target rewards 4 different amounts of points, depending on where the dart lands:

Our dart scoreboard with values from a complete miss to a bullseye

If the dart lands outside the target, player earns no points (0 points).
If the dart lands in the outer circle of the target, player earns 1 point.
If the dart lands in the middle circle of the target, player earns 5 points.
If the dart lands in the inner circle of the target, player earns 10 points.
The outer circle has a radius of 10 units (this is equivalent to the total radius for the entire target), the middle circle a radius of 5 units, and the inner circle a radius of 1. Of course, they are all centered at the same point — that is, the circles are concentric defined by the coordinates (0, 0).

Write a function that given a point in the target (defined by its Cartesian coordinates x and y, where x and y are real), returns the correct amount earned by a dart landing at that point.

```
module Darts (score) where
score :: Float -> Float -> Int
score x y
 | distance <= 1 = 10
 | distance <= 5 = 5
 | distance <= 10 = 1
 | otherwise     = 0
 where distance = sqrt (x^2 + y^2)
```

<u>Program-5</u>
Write a Haskell function that returns your age in space

Given an age in seconds, calculate how old someone would be on:

- Mercury: orbital period 0.2408467 Earth years
- Venus: orbital period 0.61519726 Earth years
- Earth: orbital period 1.0 Earth years, 365.25 Earth days, or 31557600 seconds
- Mars: orbital period 1.8808158 Earth years
- Jupiter: orbital period 11.862615 Earth years
- Saturn: orbital period 29.447498 Earth years
- Uranus: orbital period 84.016846 Earth years
- Neptune: orbital period 164.79132 Earth years

So if you were told someone were 1,000,000,000 seconds old, you should be able to say that they're 31.69 Earth-years old

```
module SpaceAge (Planet(..), ageOn) where
data Planet = Mercury
            | Venus
            | Earth
            | Mars
            | Jupiter
            | Saturn
            | Uranus
            | Neptune
ageOn :: Planet -> Float -> Float
```

```
ageOn Earth seconds   = seconds / 31557600
ageOn Mercury seconds = ageOn Earth seconds / 0.2408467
ageOn Venus seconds   = ageOn Earth seconds / 0.61519726
ageOn Mars seconds    = ageOn Earth seconds / 1.8808158
ageOn Jupiter seconds = ageOn Earth seconds / 11.862615
ageOn Saturn seconds  = ageOn Earth seconds / 29.447498
ageOn Uranus seconds  = ageOn Earth seconds / 84.016846
ageOn Neptune seconds = ageOn Earth seconds / 164.79132
```

Program-6
Write a Haskell function to check if a sentence is a pangram

A pangram is a sentence using every letter of the alphabet at least once. It is case insensitive, so it doesn't matter if a letter is lower-case (e.g. k) or upper-case (e.g. K).

```
module Pangram (isPangram) where
import Data.Char (toLower)
isPangram :: String -> Bool
isPangram text = all (`elem` map toLower text) ['a'..'z']
```

Program-7
Write a Haskell function to identify the RNA complement of a given DNA sequence

```
module DNA (toRNA) where
toRNA :: String -> Either Char String
toRNA = traverse fromDNA
  where
    fromDNA :: Char -> Either Char Char
    fromDNA 'G' = pure 'C'
    fromDNA 'C' = pure 'G'
    fromDNA 'T' = pure 'A'
    fromDNA 'A' = pure 'U'
    fromDNA c = Left c
```

Program-8
Given a string representing a DNA sequence, write a Haskell function count how many of each nucleotide is present.

For example:

"GATTACA" -> 'A': 3, 'C': 1, 'G': 1, 'T': 2
"INVALID" -> error

```
module DNA (nucleotideCounts, Nucleotide(..)) where
import Data.Map (Map, fromList)
```

```
data Nucleotide = A | C | G | T deriving (Eq, Ord, Show)
isValid :: String -> Bool
isValid [] = True
isValid s = check s
 where
  check (x:xs)
   | x /= 'A' && x /= 'C' && x /= 'G' && x /= 'T' = False
   | otherwise = isValid xs
nucleotideCounts :: String -> Either String (Map Nucleotide Int)
nucleotideCounts xs = if not (isValid xs) then Left "error" else
 let
  a = sum [ 1 | ch <- xs, ch == 'A']
  c = sum [ 1 | ch <- xs, ch == 'C']
  g = sum [ 1 | ch <- xs, ch == 'G']
  t = sum [ 1 | ch <- xs, ch == 'T']
 in Right (fromList [(A, a), (C, c), (G, g), (T, t)])
```

Program-9

Your task is to write a code that calculates the energy points that get awarded to players when they complete a level.

The points awarded depend on two things:

The level (a number) that the player completed.
The base value of each magical item collected by the player during that level.
The energy points are awarded according to the following rules:

For each magical item, take the base value and find all the multiples of that value that are less than the level number.
Combine the sets of numbers.
Remove any duplicates.
Calculate the sum of all the numbers that are left.
Let's look at an example:

The player completed level 20 and found two magical items with base values of 3 and 5.

To calculate the energy points earned by the player, we need to find all the unique multiples of these base values that are less than level 20.

Multiples of 3 less than 20: {3, 6, 9, 12, 15, 18}
Multiples of 5 less than 20: {5, 10, 15}
Combine the sets and remove duplicates: {3, 5, 6, 9, 10, 12, 15, 18}
Sum the unique multiples: 3 + 5 + 6 + 9 + 10 + 12 + 15 + 18 = 78
Therefore, the player earns 78 energy points for completing level 20 and finding the two magical items with base values of 3 and 5.

```
module SumOfMultiples (sumOfMultiples) where
```

```
import Data.List

sumOfMultiples :: [Integer] -> Integer -> Integer
sumOfMultiples factors limit = sum $ nub [x * y | x <- factors, y <- [1 ..
limit], x * y < limit]
```

Program-10

Write a Haskell function to calculate the number of grains of wheat on a chessboard given that the number on each square doubles. There are 64 squares on a chessboard (where square 1 has one grain, square 2 has two grains, and so on).

Write code that shows:

how many grains were on a given square, and
the total number of grains on the chessboard

```
module Grains (square, total) where
square :: Integer -> Maybe Integer
square n
    | n > 0 && n <= 64 = Just $ 2 ^ (n-1)
    | otherwise = Nothing
total :: Integer
total = 2 ^ 64 - 1
```

Program-11

Write a Haskell function to calculate the number of grains of wheat on a chessboard given that the number on each square doubles. There are 64 squares on a chessboard (where square 1 has one grain, square 2 has two grains, and so on).

```
module Grains (square, total) where
square :: Integer -> Maybe Integer
square n
    | n > 0 && n <= 64 = Just $ 2 ^ (n-1)
    | otherwise = Nothing
total :: Integer
total = 2 ^ 64 - 1
```

Program 12

Implement a clock that handles times without dates. You should be able to add and subtract minutes to it. Two clocks that represent the same time should be equal. It's a 24-hour clock going from "00:00" to "23:59". To complete this exercise, you need to define the data type Clock, add an Eq instance, and implement the functions:

addDelta
fromHourMin
toString

```
module Clock (addDelta, fromHourMin, toString) where

import Text.Printf (printf)

data Clock = Clock { hours :: Int, mins :: Int }
  deriving (Read, Show, Eq)

fromHourMin :: Int -> Int -> Clock
fromHourMin hour min =
  let modMins = min `mod` 60
      modHours = (hour + min `div` 60) `mod` 24
  in Clock { hours = modHours, mins = modMins }

toString :: Clock -> String
toString clock = printf "%02d:%02d" (hours clock) (mins clock)

addDelta :: Int -> Int -> Clock -> Clock
addDelta hour min clock =
  let totalMins = mins clock + min
      totalHours = hours clock + hour
      modMins = totalMins `mod` 60
      modHours = (totalHours + totalMins `div` 60) `mod` 24
  in Clock { hours = modHours, mins = modMins }
```

Program 13

Write a Haskell function to determine if a number is perfect, abundant, or deficient based on Nicomachus' (60 - 120 CE) classification scheme for positive integers.

Perfect: aliquot sum = number
6 is a perfect number because (1 + 2 + 3) = 6
28 is a perfect number because (1 + 2 + 4 + 7 + 14) = 28
Abundant: aliquot sum > number
12 is an abundant number because (1 + 2 + 3 + 4 + 6) = 16
24 is an abundant number because (1 + 2 + 3 + 4 + 6 + 8 + 12) = 36
Deficient: aliquot sum < number
8 is a deficient number because (1 + 2 + 4) = 7
Prime numbers are deficient

```
module PerfectNumbers (classify, Classification(..)) where

data Classification = Deficient | Perfect | Abundant deriving (Eq, Show)
```

```
classify :: Int -> Maybe Classification
classify int
    | int < 1   = Nothing
    | otherwise = Just $ case aliquotSum int `compare` int of
                              LT -> Deficient
                              EQ -> Perfect
                              GT -> Abundant
  where
    aliquotSum n = sum [x | x <- [1..div n 2], mod n x == 0]
```

Program 14

Write a Haskell function to Insert and search for numbers in a binary tree.

```
module BST
    ( BST
    , bstLeft
    , bstRight
    , stale
    , empty
    , fromList
    , insert
    , singleton
    , toList
    ) where

data BST a = Empty | Node a (BST a) (BST a)  deriving (Eq, Show)

bstLeft :: BST a -> Maybe (BST a)
bstLeft Empty = Nothing
bstLeft (Node _ l _) = Just l

bstRight :: BST a -> Maybe (BST a)
bstRight Empty = Nothing
bstRight (Node _ _ r) = Just r

bstValue :: BST a -> Maybe a
bstValue Empty = Nothing
bstValue (Node x _ _) = Just x

empty :: BST a
empty = Empty

fromList :: Ord a => [a] -> BST a
fromList = foldl (flip insert) Empty

insert :: Ord a => a -> BST a -> BST a
insert x Empty = singleton x
insert x (Node y l r)
  | x <= y = Node y (insert x l) r
  | otherwise = Node y l (insert x r)
```

```
singleton :: a -> BST a
singleton x = Node x Empty Empty

toList :: BST a -> [a]
toList Empty = []
toList (Node x l r) = toList l ++ [x] ++ toList r
```

Program-15

Given the position of two queens on a chess board, write a Haskell function to indicate whether they are positioned so that they can attack each other (N-queens problem, here N=8).

```
module Queens
    ( boardString
    , canAttack
    )
where

boardString :: Maybe (Int, Int) -> Maybe (Int, Int) -> String
boardString white black = unlines
    [ unwords [ board i j | j <- [0 .. 7] ] | i <- [0 .. 7] ]
  where
    board i j | Just (i, j) == white = "W"
              | Just (i, j) == black = "B"
              | otherwise            = "_"

canAttack :: (Int, Int) -> (Int, Int) -> Bool
canAttack (iA, jA) (iB, jB) =
    iDiff == 0 || jDiff == 0 || abs iDiff == abs jDiff
  where
    iDiff = iA - iB
    jDiff = jA - jB
```

Program-16

Write a robot simulator using Haskell function. A robot factory's test facility needs a program to verify robot movements. The robots have three possible movements:

- turn right
- turn left
- advance

Robots are placed on a hypothetical infinite grid, facing a particular direction (north, east, south, or west) at a set of {x,y} coordinates, e.g., {3,8}, with coordinates increasing to the north and east. The robot then receives a number of instructions, at which point the testing facility verifies the robot's new position, and in which direction it is pointing.

The letter-string "RAALAL" means:

- Turn right
- Advance twice

- Turn left
- Advance once
- Turn left yet again

Say a robot starts at {7, 3} facing north. Then running this stream of instructions should leave it at {9, 4} facing west.

To complete this exercise, you need to create the data type Robot, and implement the following functions:

- bearing
- coordinates
- mkRobot
- move

```
module Robot
    ( Bearing(East,North,South,West)
    , bearing
    , coordinates
    , mkRobot
    , move
    ) where

data Bearing = North
             | East
             | South
             | West
             deriving (Eq, Show)

instance Enum Bearing where
    toEnum 0 = North
    toEnum 1 = East
    toEnum 2 = South
    toEnum 3 = West
    toEnum n = toEnum $ n `mod` 4
    fromEnum North = 0
    fromEnum East = 1
    fromEnum South = 2
    fromEnum West = 3

data Robot = Robot { bearing :: Bearing
                   , coordinates :: (Integer, Integer)
                   } deriving (Show)

mkRobot :: Bearing -> (Integer, Integer) -> Robot
mkRobot direction coords = Robot {bearing=direction, coordinates=coords}

move :: Robot -> String -> Robot
move robot [] = robot
move robot (c : cs) = move (mkRobot direction coords) cs
    where
        (x, y) = coordinates robot
        direction
```

```
            | c == 'R' = succ $ bearing robot
            | c == 'L' = pred $ bearing robot
            | otherwise = bearing robot
        coords
            | c /= 'A' = (x, y)
            | direction == North = (x, y+1)
            | direction == East = (x+1, y)
            | direction == South = (x, y-1)
            | direction == West = (x-1, y)
            | otherwise = (x, y)
```

Program-17

Your task is to determine which items to take so that the total value of his selection is maximized, considering the knapsack's carrying capacity. Items will be represented as a list of items. Each item will have a weight and value. All values given will be strictly positive. Bob can take only one of each item.

For example:

```
Items: [
  { "weight": 5, "value": 10 },
  { "weight": 4, "value": 40 },
  { "weight": 6, "value": 30 },
  { "weight": 4, "value": 50 }
]
Knapsack Maximum Weight: 10
```

Write a Haskell function to solve the problem.

```haskell
module Knapsack (maximumValue) where

maximumValue :: Int -> [(Int, Int)] -> Int
maximumValue = go
  where
    go _ [] = 0
    go limit ((w, v) : xs)
      | w > limit = go limit xs
      | otherwise = max (v + go (limit - w) xs) (go limit xs)
```

Program-18

Implement run-length encoding and decoding using Haskell functions.
Run-length encoding (RLE) is a simple form of data compression, where runs (consecutive data elements) are replaced by just one data value and count.
For simplicity, you can assume that the unencoded string will only contain the letters A through Z (either lower or upper case) and whitespace. This way data

to be encoded will never contain any numbers and numbers inside data to be decoded always represent the count for the following character.

```
module RunLength (decode, encode) where

import Data.Char

decode :: String -> String
decode [] = []
decode xs
    | null eqs = decodeStrip difs 1
    | otherwise = decodeStrip difs (read eqs::Int)
    where (eqs, difs) = span isNumber xs
          decodeStrip (y:ys) n = (replicate n y)++(decode ys)


encode :: String -> String
encode [] = []
encode (x:xs) = (encodeStrip eqs) ++ (encode difs)
    where (eqs, difs) = span (== x) xs
          encodeStrip [] = [x]
          encodeStrip ys = show ((length ys) + 1) ++ [x]
```

Program-19

Write a Haskell prototype of the music player application. For the prototype, each song will simply be represented by a number. Given a range of numbers (the song IDs), create a singly linked list. Given a singly linked list, you should be able to reverse the list to play the songs in the opposite order.

```
module LinkedList
    ( LinkedList
    , datum
    , fromList
    , isNil
    , new
    , next
    , nil
    , reverseLinkedList
    , toList
    ) where

data LinkedList a = LinkedList {datum :: a, next :: LinkedList a} | Nil
  deriving (Eq, Show)

fromList :: [a] -> LinkedList a
fromList [] = Nil
```

```
fromList (x:xs) = LinkedList x (fromList xs)

isNil :: LinkedList a -> Bool
isNil (LinkedList _ _) = False
isNil Nil = True

new :: a -> LinkedList a -> LinkedList a
new x linkedList = LinkedList x linkedList

nil :: LinkedList a
nil = Nil

_reverse :: LinkedList a -> LinkedList a -> LinkedList a
_reverse Nil Nil = Nil
_reverse Nil as = as
_reverse as bs = _reverse (next as) (LinkedList (datum as) bs)

reverseLinkedList :: LinkedList a -> LinkedList a
reverseLinkedList as = _reverse as Nil

toList :: LinkedList a -> [a]
toList Nil = []
toList (LinkedList d n) = d : toList n
```

Program-20

Correctly determine the fewest number of coins to be given to a customer such that the sum of the coins' value would equal the correct amount of change
For example: An input of 15 with [1, 5, 10, 25, 100] should return [5, 10]

```
module Change (findFewestCoins) where

import Data.List (tails)
import Data.Maybe (listToMaybe)

draw :: [a] -> Int -> [[a]]
draw _ 0 = [[]]
draw xs n = [t0 : rest | t@(t0:_) <- tails xs, rest <- draw t (n-1)]

findFewestCoins :: Integer -> [Integer] -> Maybe [Integer]
findFewestCoins target coins = listToMaybe (solutions choices)
  where
    solutions = filter (\cs -> sum cs == target)
    choices = draw coins =<< [0..fromIntegral (target `div` minimum
coins)]
```

Program-21

Given an input integer N, find all Pythagorean triplets for which a + b + c = N. For example, with N = 1000, there is exactly one Pythagorean triplet for which a + b + c = 1000: {200, 375, 425}.

```
module Triplet (tripletsWithSum) where

tripletsWithSum :: Int -> [(Int, Int, Int)]
tripletsWithSum sum = [(a, b, c) | a <- [1..sum], b <- [a+1..sum], let
c = sum-a-b, a^2 + b^2 == c^2]
```

<u>Program-22</u>

Create an implementation of the rotational cipher, also sometimes called the Caesar cipher. The Caesar cipher is a simple shift cipher that relies on transposing all the letters in the alphabet using an integer key between 0 and 26. Using a key of 0 or 26 will always yield the same output due to modular arithmetic. The letter is shifted for as many values as the value of the key.

The general notation for rotational ciphers is ROT + <key>. The most commonly used rotational cipher is ROT13.

A ROT13 on the Latin alphabet would be as follows:

Plain: abcdefghijklmnopqrstuvwxyz
Cipher: nopqrstuvwxyzabcdefghijklm

```
module RotationalCipher (rotate) where

import Data.Char

cipherLower x n = chr ( ord 'a' + mod (ord x - ord 'a' + n) 26)
cipherUpper x n = chr ( ord 'A' + mod (ord x - ord 'A' + n) 26)

rotate :: Int -> String -> String
rotate n [] = []
rotate n (x:xs)
      | isLower x = cipherLower x n:(rotate n xs)
      | isUpper x = cipherUpper x n:(rotate n xs)
      | otherwise = x:rotate n xs
```

<u>Program-23</u>

<u>Implement a Haskell function that determines the state of a tic-tac-toe game played on a 3x3 grid</u>

```
module StateOfTicTacToe (gameState, GameState (..)) where

import Data.List (transpose)
```

```
data GameState = WinX | WinO | Draw | Ongoing | Impossible deriving (Eq,
Show)

gameState :: [String] -> GameState
gameState board
  | isImpossible = Impossible
  | winX = WinX
  | winO = WinO
  | isDraw = Draw
  | otherwise = Ongoing
  where
    winX :: Bool
    winX = win 'X'

    winO :: Bool
    winO = win 'O'

    win :: Char -> Bool
    win player =
      winHorizontally player
        || winVertically player
        || winDiagonally player

    isDraw :: Bool
    isDraw = countX + countO == 9

    isImpossible :: Bool
    isImpossible =
      win 'X' && win 'O' || wentTwice || oStarted
      where
        wentTwice = countX - countO > 1
        oStarted = countO > countX

    countPlayer :: Char -> Int
    countPlayer player = length . filter (== player) . concat $ board

    countX :: Int
    countX = countPlayer 'X'

    countO :: Int
    countO = countPlayer 'O'

    winHorizontally :: Char -> Bool
    winHorizontally player = winsRow player board

    winVertically :: Char -> Bool
    winVertically player = winsRow player (transpose board)

    winsRow :: Char -> [String] -> Bool
    winsRow player board' =
      any (\row -> [player, player, player] == filter (== player) row)
```

```
board'

    winDiagonally :: Char -> Bool
    winDiagonally player =
      [board !! 0 !! 0, board !! 1 !! 1, board !! 2 !! 2] == [player,
player, player]
        || [board !! 0 !! 2, board !! 1 !! 1, board !! 2 !! 0] == [player,
player, player]
```