Lecture 14-16 **24-28.2.2025**

Today's agenda:

Some features in Haskell adopted from LC:

..

Function definition

Example 1

square :: Int -> Int (1)
square x = x * x (2)

Line (1) says that the function named square takes as i/p an integer and returns as o/p an integer; this is called the signature or type of the function; -> denotes maps to.

Line (2) the body of the function is written in the RHS of =. LHS says that the input parameter is x. The notation of function application is employed in the LHS (like f x or sin θ).

In general, the structure of function definition will look like

<function name> :: <type of the function>
<function name> <input parameters> = <body of the function>

Example 2

plus :: Int -> Int -> Int
plus x y = x + y

The function named plus takes as i/p two integers and returns as o/p an integer

List

Example: [2, 3, 7], [ ] , [2]

Empty list: [ ] length of this list is zero

Nonempty list [2, 3, 7] can be written as 2 : [3,7] : constructor for list
It has a head (2)—an item and a tail [3,7]—a list; we will write x : xs to represent a nonempty list

How is a list constructed?

[2, 3, 7] = 2 : [3,7] = 2 : 3 : [7] = 2 : 3 : 7 : [ ] : right associative

The process works internally as per the following steps.

1. Empty list [ ]
2. 7 : [ ] = [7]   (element, list)
3. 3 : [7] = [3,7]   (element, list)
4. 2 : [3,7] = [2,3,7]   (element, list)

The operator : has type a -> [a] -> [a] takes as input an element of type a and a list of elements of type a and returns a list of elements of type a.

Recursion with lists

```
length :: [a] -> Int
length [ ] = 0
length (x : xs) = 1 + length xs
```

computation:

```
length [2,3,5] = length (2 : [3,5])
               = 1 + length [3,5]
               = 1 + length (2 : [5])
               = 1 + 1 + length [5]
               = 1 + 1 + length ( 5 : [ ])
               = 1 + 1 + 1 + length [ ]
               = 1 + 1 + 1 + 0
               = 3
sum :: [Int] -> Int
sum [ ] = 0
sum (x : xs) = x + sum xs
```

sum [2, 3, 5] = sum (2 : [3,5]) = 2 + sum [3,5] = 2 + sum (3 : [5]) = 2+ 3 + sum(5 : [ ]) = 2+3+5 + sum [ ] = 2+3+5+0 = 10

Insertion sort

suppose we have a sorted list, say, in ascending order. we want to insert an element into this list (in the proper position)

```
insert :: int -> [Int] -> [Int]
insert x [ ] = [x]
insert x (y : ys)
|  x <= y  = x : (y : ys)          | denotes guard
|  x > y   = y : insert x xs
```

2: [3, 6] = [2,3,6]
5: [3,6] = 3 : insert 5 [6]
insert 5 [6] = insert 5 (6: []) = [5,6]

combining we get [3,5,6]

inSort :: [Int] -> [Int]
inSort [ ] = [ ]
inSort (x : xs) = insert x (inSort xs)

let us do insertion sort on the list 7: (5:3)

inSort (7: [5,3])
insert 7 (inSort [5,3])
inSort [5,3] = insert 5 (inSort [3])
inSort (3: [ ]) = insert 3 (inSort [ ])
inSort [ ] = [ ]
insert 3 [ ] = [3] first created
then we obtain [3,5]
finally we obtain [3,5,7]

end of lecture 24.2.

HOF

Function Composition (f • g) x = f (g x)
Composition is associative: (f • g) • h = f • (g • h) for all f,g,h

Currying

Multiply :: Int -> Int -> Int -> is right associative
Multiply 2 :: Int -> Int
(Multiply 2) 3 :: Int
Multiply 2 3 :: Int function application is left associative

MultiplyUC :: (Int, Int) -> Int
MultiplyUC (x, y) = x * y

The type Int -> Int -> Int is same as Int -> (Int -> Int) but it is different from (Int -> Int) -> Int

g :: (Int -> Int) -> Int
g h = (h 1) + (h 2)
let h = succ, then g h = 5

map function
map :: (a -> b) -> [a] -> [b] a,b are type variables

map: (Int -> Int) -> [Int] -> [Int] instance of map
map succ [2,4,6] = [3,5,7]
map sqr [2,4,6] = [4,16, 36]

map f [ ] = [ ]
map f (x : xs) = f x : map f xs

filter iseven [2, 3, 6] = [2,6]

iseven n = (mod n 2 == 0)

filter :: (a-> Bool) -> [a] -> [a]
filter p [ ] = [ ]
filter p (x : xs)
| p x = x : filter p xs
| otherwise = filter p xs

Lambda abstraction:

map addone [2,3,4] = [3,4,5]
map (\x -> x+1) [2,3,4] = [3,4,5]

(\x -> x+1) = $\lambda x.\ x + 1$

f x y z = result is same as \x y z -> result is same as $\lambda x.\lambda y.\lambda z.$ result

length :: [a] -> Int
length [ ] = 0
length (_ : xs) = 1 + length xs   since x has no role in the RHS

concatenation operator: ++
[1,2,4] ++ [9,4] = [1,2,4,9,4]

reverse :: [a] -> [a]
reverse [ ] = [ ]
reverse (x : xs) = (reverse xs) ++ [x]

isequal :: Int -> Int -> Bool
isequal x x = True
isequal x y = False   [what is wrong with this code?]   [arguments must be distinct]

isequal x y = (x == y)   == denotes test

can we check if two lists are same?
isequalLists :: [a] -> [a] -> Bool
isequal ListA List B = (ListA == ListB)

give a recursive code that checks element wise.

End of lecture 27.2.25

List comprehension

Let inp = [2, 4, 8]

[3*n | n <- inp] will be [6, 12, 24]   <- is like the set membership operator

"Perform 3*n for all n in the list inp."

```
quicksort :: [Int] -> [Int]
quicksort [ ] = [ ]
quicksort (x : xs) = quicksort [ y | y <- xs, y <= x ] ++ [x] ++ quicksort [ y | y <- xs, y > x]
```

we have simply written the idea behind quicksort.

Lazy evaluation

f x y = x + y   let x = (9-3) y = (f 34 3)

then f x y = f (9-3) (f 34 3) = (9-3) + (f 34 3) = 6 + (34 + 3) = 6 + 37 = 43 [CBN]

Infinite lists

ones = 1 : ones will produce [1, 1, 1, ...] interrupt by ctrl-c

```
addfirsttwo :: [Int] -> Int
addfirsttwo (x:y:zs) = x + y
```

addfirsttwo ones = addfirsttwo (1 : ones) = addfirsttwo (1 : 1 : ones) = 1+1 = 2

in-built [n ..] will give for n=2 [2, 3,4,5,....]
[n, m ..] will give for n=2, m=2 [2,4,6,8,.....]
[n .. m] will give [n, n+1, n+2, ..., m]
will give [ ] if m < n

=

Development of Haskell (or any FPL) from the primitive PL (i.e., LC) that in turn from the primitive recursive functions and computable functions in general.

PRF : 1920s   LC : 1930s   Haskell : 1990s   [60-70 years]

ANN : 1950s   DL : 2000   AI-ML (popularity) : 2010s [60 years]

End of lecture 27.2.25

=