CS331
Haskell Tutorial 02

A. Sahu
Dept of Comp. Sc. & Engg.
Indian Institute of Technology Guwahati

Outline

- Functional programming
- Let constructs
- Writing Fact in multiple ways
- Currying, Composition
- Lazy Evaluation

Expression-Oriented

- An example function:

```
fact      :: Integer -> Integer
fact n   = if n=0 then 1
             else n * fact (n-1)
```

- Can use pattern-matching instead of conditional

```
fact 0              = 1
fact n              = n * fact (n-1)
```

- Alternatively:

```
fact n              = case n of
                   0 -> 1
                   a -> a * fact (a-1)
```

Writing multiline function

- Space and indentation is important in writing code
- Use space instead of Tab
- Writing multiline function; Start with :{ and end with :} , spacing and newline is must

```
Prelude> :{
Prelude| fact n = if n==0 then 1
Prelude|              else n * fact (n-1)
Prelude| :}
```

Notation

- We can abbreviate repeated left hand sides

```
absolute x | x >= 0 = x
absolute x | x < 0   = -x
```

```
absolute x | x >= 0 = x
           | x < 0   = -x
```

- Haskell also has **if then else**

absolute x = **if** x >= 0 **then** x **else** -x

```
Prelude> :{
Prelude| absolute x | x>=0 = x
Prelude|            | x<0  = -x
Prelude| :}
Prelude> absolute (-24)
24
```

Loading from HS file

- Loading Haskell script (source code) from file
- Suppose fact.hs contents this : **H**askell **S**cript

```
fact n = if n==0 then 1
         else n * fact (n-1)
```

- Any module it say as Main : from file

```
Prelude> :load fact.hs
[1 of 1] Compiling Main      ( fact.hs, interpreted )
Ok, one module loaded.
*Main> fact 4
24
*Main> :m - Main
Prelude>
```

Conditional → Case Construct

- Conditional;

```
if e1 then e2 else e3
```

- Can be translated to

```
case e1 of
  True -> e2
  False -> e3
```

- Case also works over data structures (without any extra primitives)

```
length xs = case xs of
              [] -> 0;
              y:ys -> 1+(length ys)
```

$\nwarrow$ Locally bound variables

Lexical Scoping

- Local variables can be created by let construct to give nested scope for the name space.

Example:

```
let       y  = a+b
        f x = (x+y)/y
in f c + f d
```

```
Prelude> :{
Prelude> | myf c d =
Prelude> |          let y = 7+3
Prelude> |              f x = (x+y)/2
Prelude> |          in f c + f d
Prelude> |:}
Prelude> myf 20 30
Prelude> 35.0
```

Layout Rule

- Haskell uses two dimensional syntax that relies on declarations being "lined-up columnwise"

```
let  y     = a+b
     f x   = (x+y)/y
in f c + f d
```

is being parsed as:

```
let  { y   = a+b
     ; f x = (x+y)/y }
in f c + f d
```

- Rule : Next character after keywords `where`/`let`/`of`/`do` determines the starting columns for declarations. Starting *after* this point continues a declaration, while starting *before* this point terminates a declaration.

Lexical Scoping

- For scope bindings over guarded expressions, we require a where construct instead:

```
f x y | x>z     = …
      | y==z    = …
      | y<z     = … .
    where z=x*x
```

Where example : write like math Statement

```haskell
roots (a,b,c) = (x1, x2) where
   x1 = e + sqrt d / (2 * a)
   x2 = e - sqrt d / (2 * a)
   d = b * b - 4 * a * c
   e = - b / (2 * a)
main = do
   putStrLn "The roots of our Polynomial equation are:"
   print (roots(1,-8,6))
```

```
Prelude> :load WhereExample.hs
*Main> main
The roots of our Polynomial equation are:
(7.1622777,0.8377223)
```

Expression Evaluation

- Expression can be computed (or evaluated) so that it is reduced to a value. This can be represented as:

  $e \rightarrow .. \rightarrow v$

- We can abbreviate above as:

  $e \rightarrow^{*} v$

- A concrete example of this is:

  ```
  inc (inc 3) → inc (4) → 5
  ```

- Type preservation theorem says that:

  if $e :: t$ Æ $e \rightarrow v$, it follows that $v :: t$

Polymorphic Types

- This polymorphic function can be used on list of any type..

```
length [1,2,3]                )    2
length [‘a’, ’b’, ‘c’]        )    3
length [[1],[],[3]]           )    3
```

- More examples :

```
head        :: [a] -> a
head (x:xs)      = x

tail        :: [a] -> [a]
tail (x:xs)      = xs
```

- Note that *head/tail* are partial functions, while *length* is a total function?

Functions and its Type

- Method to increment its input

```
inc x=  x+1
```

- Or through lambda expression (anonymous functions)

```
(\ x -> x+1)
```

- They can also be given suitable function typing:

```
inc            :: Num a => a -> a
(\x -> x+1)      :: Num a => a -> a
```

- Types can be *user-supplied* or *inferred*.

Anonymous Functions

- Anonymous functions are used often in Haskell, usually enclosed in parentheses
- `\x y -> (x + y) / 2`
  - the `\` is pronounced "lambda"
    - It's just a convenient way to type λ
  - the `x` and `y` are the formal parameters
- Functions are first-class objects and can be assigned
  - `avg = \x y -> (x + y) / 2`

Functions and its Type

- Some examples

```
(\x -> x+1) 3.2  →

(\x -> x+1) 3  →

Prelude> (\x -> x+1) 3
```

- User can restrict the type, e.g.

```
inc    :: Int -> Int
```

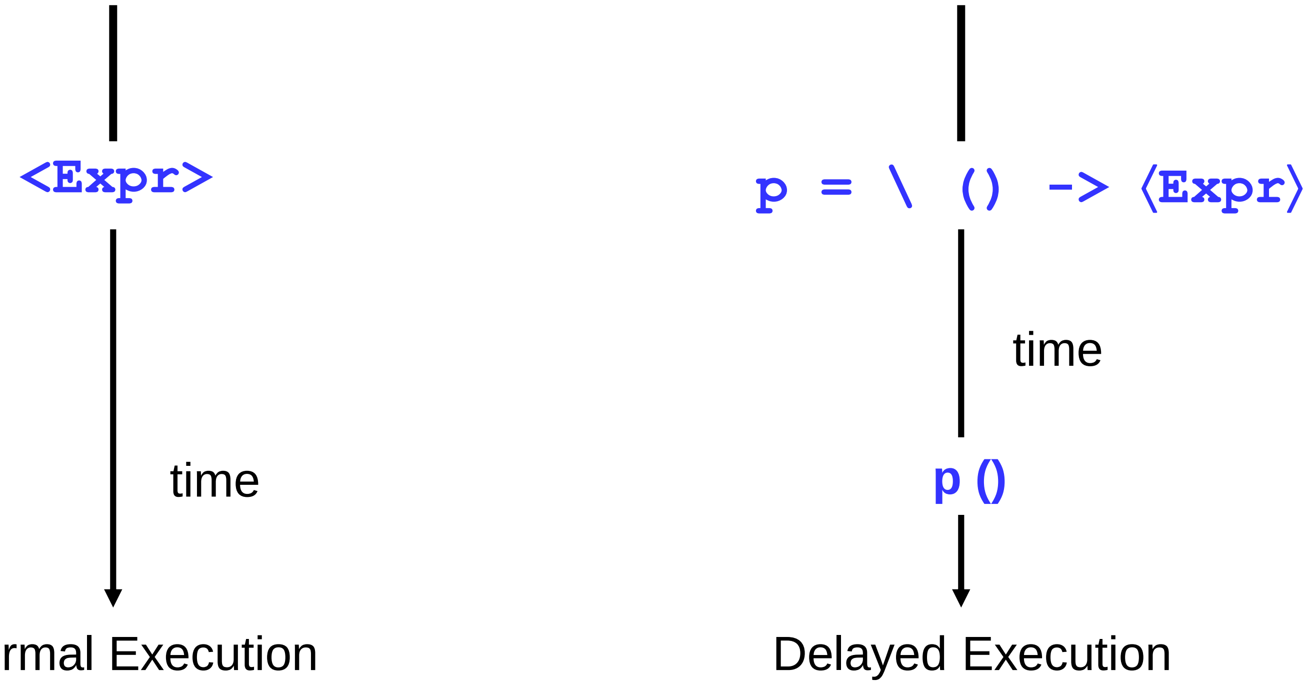
- In that case, some examples may be wrongly typed.

```
inc 3.2  →

inc 3  →
```

Function Abstraction

- Function abstraction is the ability to convert any expression into a function that is evaluated at a later time.



Higher-Order Functions

- **Higher-order programming** treats functions as first-class,
  - Allowing them to be passed as parameters, returned as results or stored into data structures.
- This concept supports generic coding,
  - and allows programming to be carried out at a more abstract level.
- Genericity can be applied to a function
  - by letting specific operation/value in the function body to become parameters.

Higher order Functions

- Functions can be written in two main ways:

```
add x y        = x+y
add2(x,y)      = x+y
```

- The first version allows a function to be returned as result after applying a single argument.

```
inc   =   add 1
```

```
Prelude> add x y = x+y
Prelude> inc = add 1
Prelude > inc 5
6
Prelude>
```

Higher order Functions

- The second version needs all arguments. Same effect requires a lambda abstraction:

```
add2(x,y)      = x+y
inc    = \x -> add2(x,1)
```

```
Prelude> add2 (x+y) = x+y
Prelude> inc =  \x -> add2(x, 1)
Prelude > inc 5
6
Prelude>
```

Functions

- Functions can also be passed as parameters. Example:

```
map              :: (a->b) -> [a] -> [b]
map f []         = []
map f (x:xs)     = (f x) : (map f xs)
```

- Such higher-order function aids code reuse.

```
map (add 1) [1, 2, 3]    ) [2, 3, 4]
map add [1, 2, 3]        ) [add 1, add 2, add 3]
```

- Alternative ways of defining functions:

```
add            = \ x -> \ y -> x+y
add            = \ x y -> x+y
```

Haskell Brooks Curry

- Haskell Brooks Curry (September 12, 1900 – September 1, 1982)
- Developed Combinatorial Logic, the basis for Haskell and many other functional languages

Currying

- **Technique named after: logician Haskell Curry**
- **Currying** absorbs an argument into a function, returning a new function that takes one fewer argument
- `f a b = (f a) b`, where `(f a)` is a curried function
  - For example, if `avg = \x y -> (x + y) / 2` then `(avg 6)` returns a function
    - This new function takes one argument (y) and returns the average of that argument with 6
  - Consequently, we can say that in Haskell, every function takes exactly one argument

Currying

- For example, if `avg = \x y -> (x + y) / 2` then `(avg 6)` returns a function
  - This new function takes one argument (y) and returns the average of that argument with 6

```
Prelude> avg = \x  y  ->  (x  +  y)  /  2
Prelude> (avg 6) 20
```

Currying example

- "And", `&&`, has the type `Bool -> Bool -> Bool`
- `x && y` can be written as `(&&) x y`
- If `x` is `True`,
  `(&&)x` is a function that returns the value of `y`
- If `x` is `False`,
  `(&&)x` is a function that returns `False`
  - It accepts `y` as a parameter, but doesn't use its value

Slicing

- `negative = (< 0)`

```
Main> negative 5
False
Main> negative (-3)
True
Main> :type negative
negative :: Integer -> Bool
Main>
```

Factorial I

```
fact n =
   if n == 0 then 1
   else n * fact (n - 1)
```

This is an extremely conventional definition.

Factorial II

```
fact n
    | n == 0          = 1
    | otherwise = n * fact (n - 1)
```

Each `|` indicates a “guard.”

Notice where the equal signs are.

Factorial III

```
fact n = case n of
   0 -> 1
   n -> n * fact (n - 1)
```

This is essentially the same as the last definition.

Factorial IV

You can introduce new variables with

`let` ***declarations*** `in` ***expression***

```
fact n
   | n == 0    = 1
   | otherwise = let m = n - 1 in n * fact m
```

Factorial V

You can also introduce new variables with

expression `where` *declarations*

```
fact n
   | n == 0        = 1
   | otherwise = n * fact m
   where m = n - 1
```

List

- List creation/declaration

```
myData = [1,2,3,4,5,6,7]
```

Operations on Lists I

head	[a] -> a	First element
tail	[a] -> [a]	All but first
:	a -> [a] -> [a]	Add as first
last	[a] -> a	Last element
init	[a] -> [a]	All but last
reverse	[a] -> [a]	Reverse

Operations on Lists II

`!!`	`[a] -> Int -> a`	Index (from 0)
`take`	`Int -> [a] -> [a]`	First n elements
`drop`	`Int -> [a] -> [a]`	Remove first n
`nub`	`[a] -> [a]`	Remove duplicates
`length`	`[a] -> Int`	Number of elements

Operations on Lists III

`elem, notElem`	`a -> [a] -> Bool`	Membership
`concat`	`[[a]] -> [a]`	Concatenate lists

Operations on Tuples

`fst (a, b) -> a`	First of two elements
`snd (a, b) -> b`	Second of two elements

…and nothing else, really.

Finite and Infinite Lists

`[a..b]`	All values a to b	`[1..4] = [1, 2, 3, 4]`
`[a..]`	All values a and larger	`[1..]` = positive integers
`[a, b..c]`	a step (b-a) up to c	`[1, 3..10] = [1,3,5,7,9]`
`[a, b..]`	a step (b-a)	`[1, 3..] =` positive odd integers

List Comprehensions-0

- Notation for constructing new lists from old:

```
myData = [1,2,3,4,5,6,7]

twiceData = [2 * x | x <- myData]
-- [2,4,6,8,10,12,14]

twiceEvenData = [2 * x| x <- myData, x `mod` 2 == 0]
-- [4,8,12]
```

- Similar to “set comprehension”

$$\{ x \mid x \in \text{Odd} \wedge x > 6 \}$$

List Comprehensions I

- `[` ***expression_using_x*** `| x <-` ***list*** `]`
  - read: <expression> where x is in <list>
  - `x <-` ***list*** is called a generator
- Example: `[ x * x | x <- [1..] ]`
  - This is the list of squares of positive integers
- `take 5 [x * x | x <- [1..]]`
  - `[1,4,9,16,25]`

List Comprehensions II

- `[` ***expression_using_x_and_y*** `| x <-` ***list***`, y <-` ***list***`]`
- `take 10 [x*y | x <- [2..], y <- [2..]]`
  - `[4,6,8,10,12,14,16,18,20,22]`
- `take 10 [x * y | x <- [1..], y <- [1..]]`
  - `[1,2,3,4,5,6,7,8,9,10]`
- `take 5 [(x,y) | x <- [1,2], y <- "abc"]`
  - `[(1,'a'),(1,'b'),(1,'c'),(2,'a'),(2,'b')]`

List Comprehensions III

- [***expression_using_x*** | ***generator_for_x***, ***test_on_x***]

- `take 5 [x*x | x <- [1..], even x]`
  - `[4,16,36,64,100]`

List Comprehensions IV

- `[x+y | x <- [1..5], even x, y <- [1..5], odd y]`
  - `[3,5,7,5,7,9]`
- `[x+y | x <- [1..5], y <- [1..5], even x, odd y]`
  - `[3,5,7,5,7,9]`
- `[x+y | y <- [1..5], x <- [1..5], even x, odd y]`
  - `[3,5,5,7,7,9]`

Set Comprehensions

In mathematics, the <u>comprehension</u> notation can be used to construct new sets from old sets.

$\{x^2 \mid x \in \{1...5\}\}$

The set {1,4,9,16,25} of all numbers x^2 such that x is an element of the set {1...5}.

Lists Comprehensions

In Haskell, a similar comprehension notation can be used to construct new lists from old lists.

```
[x^2 | x ← [1..5]]
```

The list [1,4,9,16,25] of all numbers x^2 such that x is an element of the list [1..5].

Note: ## Lists Comprehensions

- The expression x ← [1..5] is called a <u>generator</u>, as it states how to generate values for x.

- Comprehensions can have <u>multiple</u> generators, separated by commas. For example:

```
> [(x,y) | x ← [1,2,3], y ← [4,5]]

[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
```

Lists Comprehensions

- Changing the order of the generators changes the order of the elements in the final list:

```
> [(x,y) | y ← [4,5], x ← [1,2,3]]

[(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)]
```

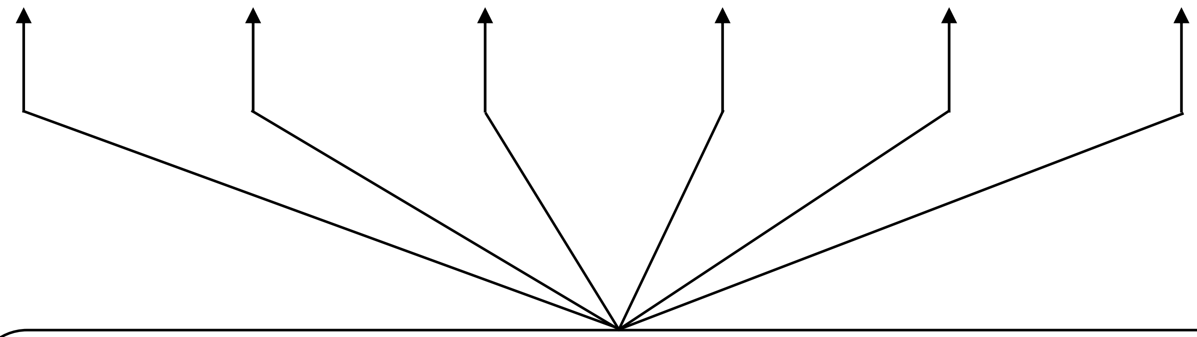
- Multiple generators are like nested loops, with later generators as more deeply nested loops whose variables change value more frequently.

Lists Comprehensions

- For example:

```
> [(x,y) | y ← [4,5], x ← [1,2,3]]

[(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)]
```



x ← [1,2,3] is the last generator, so the value of the x component of each pair changes most frequently.

Factorial VI : Revisited

```
product [] = 1
product (a:x) = a * product x

fact n = product [1..n]
```

Dependent Generators

Later generators can depend on the variables that are introduced by earlier generators.

```
[(x,y) | x ← [1..3], y ← [x..3]]
```

The list [(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)] of all pairs of numbers (x,y) such that x,y are elements of the list [1..3] and $y \geq x$.

Guards

List comprehensions can use guards to restrict the values produced by earlier generators.

```
[x | x ← [1..10], even x]
```

The list [2,4,6,8,10] of all numbers x such that x is an element of the list [1..10] and x is even.

Guards

Using a guard we can define a function that maps a positive integer to its list of factors:

```
factors  :: Int → [Int]
factors n =
   [x | x ← [1..n], n `mod` x == 0]
```

For example:

```
> factors 15

[1,3,5,15]
```

Guards

A positive integer is prime if its only factors are 1 and itself. Hence, using factors we can define a function that decides if a number is prime:

```
prime  :: Int → Bool
prime n = factors n == [1,n]
```

For example:

```
> prime 15
False

> prime 7
True
```

Using a guard we can now define a function that returns the list of all primes up to a given limit:

```
primes   :: Int → [Int]
primes n = [x | x ← [2..n], prime x]
```

For example:

```
> primes 40

[2,3,5,7,11,13,17,19,23,29,31,37]
```

Thanks