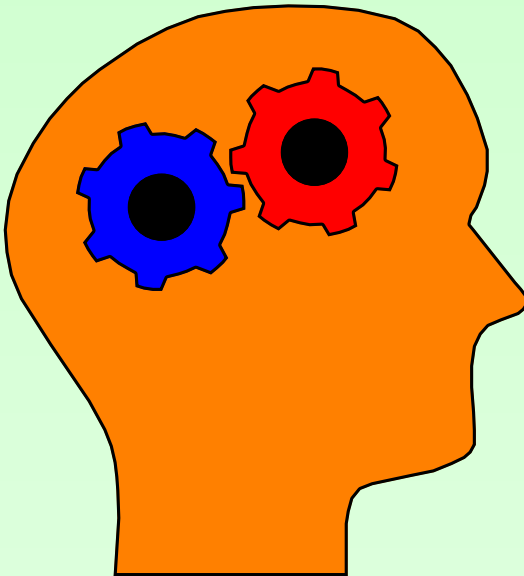




# CS2104: Programming Languages Concepts

## Lecture 2 : **Functions, Binders and Layout Rule**



*“FP, Binders and Layout Mechanism  
with Haskell”*

Lecturer : Chin Wei Ngan

Email : chinwn@comp.nus.edu.sg

Office : COM2 4-32

# *Topics*

- Declarative Programming Concepts
- Writing Functions
- Local Bindings
- Typeful Programming

## Advanced Language -



- Strongly-typed with polymorphism
- Higher-order functions
- Pure and Lazy Language.
- Algebraic data types + records
- Exceptions
- Type classes, Monads, Arrows, etc
- Advantages : concise, abstract, pure
- Why use Haskell ?

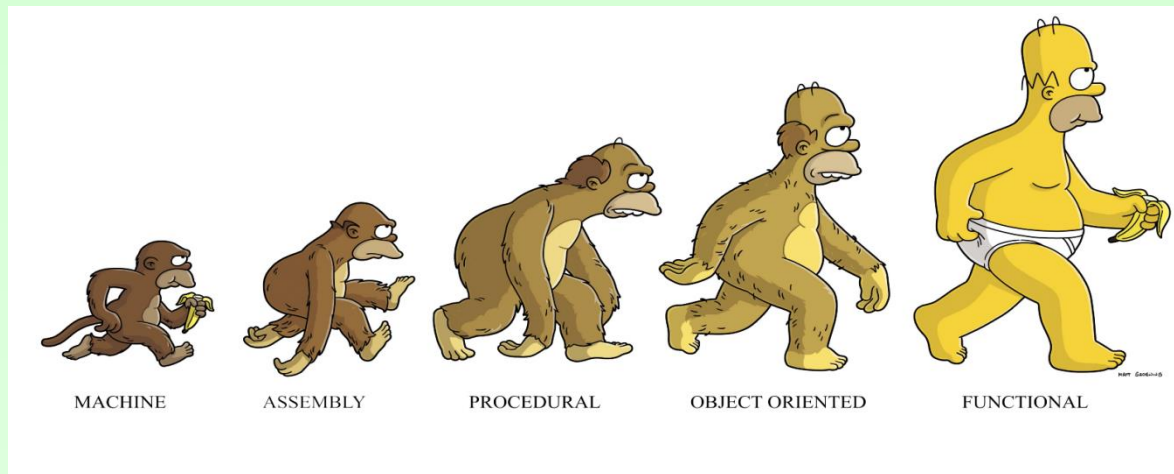
**Cool, clean  
and Pure**

# *What is Purely Functional?*

No updates and side-effects.



Easier debugging and better abstraction



# ***Declarative Programming Model Philosophy***

- Ideal of declarative programming
  - say **what** you want to compute
  - let computer find **how** to compute it
- More pragmatically
  - let the computer provide more support
  - free the programmer from some burden

# *Properties of Declarative Models*

- Focus on functions (or relations) which compute when given some data structures as inputs
- Widely used
  - functional languages: LISP, Scheme, ML, Haskell, ...
  - logic languages: Prolog, Mercury, ...
  - representation languages: XML, XSL, ...
- Stateless programming
  - no update of data structures
  - Simple data transformer

# *Pure Functions*

- Pure functions are *mathematical* functions whose outputs depend and solely on its inputs.
- Declarative Haskell
  - One of the purest language.
  - data structures is immutable and moreover is lazy by default.

# Reference

- Haskell home-page  
[www.haskell.org/documentation](http://www.haskell.org/documentation)

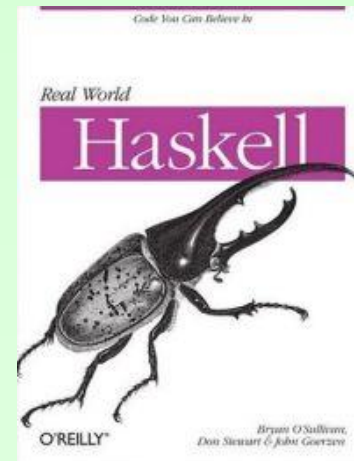
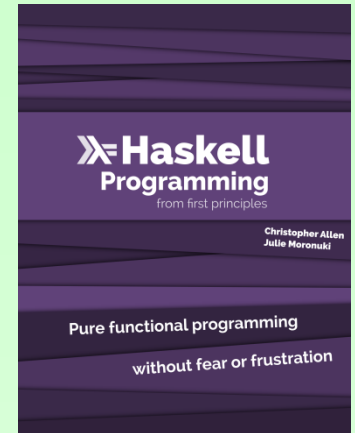
- A fast Haskell tutorial

<https://www.schoolofhaskell.com/school/starting-with-haskell/haskell-fast-hard>

- A gentle and comprehensive Haskell book at [haskellbook.com](http://haskellbook.com)

- Real World Haskell

<http://book.realworldhaskell.org/>





# ***Three Basic PL Concepts***

- Writing Functions
- Local Bindings
- Typeful Programming

## ***Example - Haskell Program***

- Computing Factorial.

-- written using conditional construct

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

-- written using pattern-matching

```
fact1 :: Int -> Int
fact1 0 = 1
fact1 n = n * fact1 (n-1)
```

-- Int denotes bounded integer (with possible overflow)

```
*Main> fact 20
2432902008176640000
*Main> fact 30
-8764578968847253504
```

## *Example - Haskell Program*

- Factorial with Infinite Precision

-- Integer denotes integer of arbitrary precision

```
fact2 :: Integer -> Integer
fact2 0 = 1
fact2 n = n * fact2 (n-1)
```

-- No more overflows

```
*Main> fact2 20
2432902008176640000
*Main> fact2 30
265252859812191058636308480000000
*Main> fact2 40
815915283247897734345611269596115894272000000000
```

## *Example - Haskell Program*

- Mixed Precision

```
fact3 :: Int -> Integer
fact3 0 = 1
fact3 n = n * fact3(n-1)
```

-- However, we get a type error. Why?

```
lec1.hs:16:11:
  Couldn't match expected type `Integer' with actual type `Int'
  In the first argument of `(*)', namely `n'
  In the expression: n * fact3 (n - 1)
  In an equation for `fact3': fact3 n = n * fact3 (n - 1)
```

## ***Example - Haskell Program***

- Fix problem using type conversion

```
-- fromIntegral :: Int -> Integer
```

```
fact3 :: Int -> Integer
```

```
fact3 0 = 1
```

```
fact3 n = (fromIntegral n) * fact2(n-1)
```

```
-- Type in Haskell is actually much more general ...more details later.
```

```
*Main> :t fromIntegral
```

```
fromIntegral :: (Integral a, Num b) => a -> b
```

```
-- For example, can also convert to floating point numbers ...
```

```
*Main> (fact 20)
```

```
2432902008176640000
```

```
*Main> (fromIntegral (fact 20)) :: Double
```

```
2.43290200817664e18
```

## Example - Haskell Program

- Finite and infinite lists.

a type variable

```
data List a = Nil | Cons a (List a)
```

```
finite_list :: List Int
```

```
finite_list = Cons 1 (Cons 2 (Cons 3 Nil))
```

*finite list of three elements*

```
infint :: Int -> List Int
```

```
infint n = Cons n (infint (n+1))
```

*infinite list starting from  $[n, n+1, n+2, \dots]$*



# Local Binding

- Binders are a special class of constructs which declare **new identifiers** and their possible values.
- Each binder has some lexical scope where the identifier is visible.
- Values of the binders may be either *immutable* or *mutable*.
- Examples of name binders
  - Local declarations
  - Method declaration (parameters)
  - Pattern-Matching

# Local Binder in Haskell

- Local variable binders in Haskell are immutable

```
let x = 3::Integer  
in x*2
```

- where  $x$  is a local immutable variable denoting a value with a fixed scope.
- In general:

```
let v = e1  
in e2
```

For Haskell, scope of  $v$  is in both  $e1$  and  $e2$ .



# Local Binder in Haskell

- Let binding in Haskell is recursive

```
let x = 1:x in  
in x
```

which is an infinite list of `[1,1,1,1,...]`

- Can write recursive function:

```
let f = \ n ->  
    if n==0 then 1  
    else n * (f (n-1))  
in f
```

- Be careful to avoid infinite loop:

```
let x = 1+x  
in x*2
```

## Local Binder in OCaml

- Let binder in OCaml is non-recursive by default

```
let x = 1
in let x = x+2
in x
```

which will return 3.

- For OCaml:

```
let v = e1
in e2
```

The scope of **v** is only in **e2**.

- Recursive let requires an extra keyword :

```
let rec v = e1
in e2
```

The scope of **v** is now in both **e1** and **e2**.

## Local Binder in C

- Local binder in C is *mutable* and *non-recursive*

scope of  
local variable {  
    { ...;  
    <type> <id>;  
    ...  
}

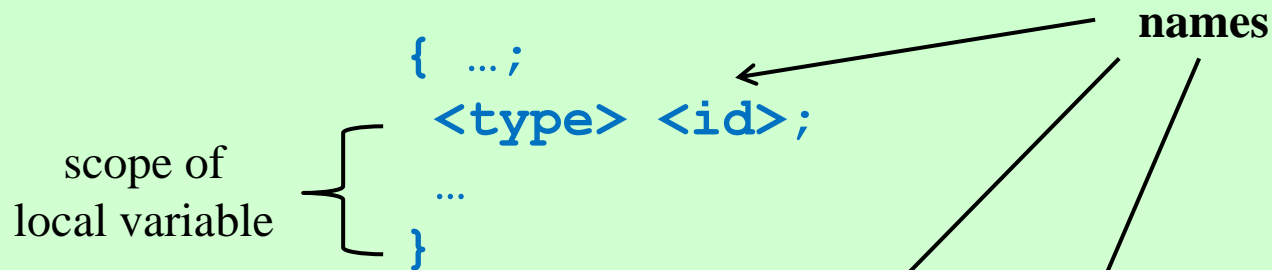
- Declare variable without a value. Assign variable to a new value with the assignment statement.

```
{ ...;  
  int x;  
  x = 1;  
  x = x + 3  
  ...  
}
```

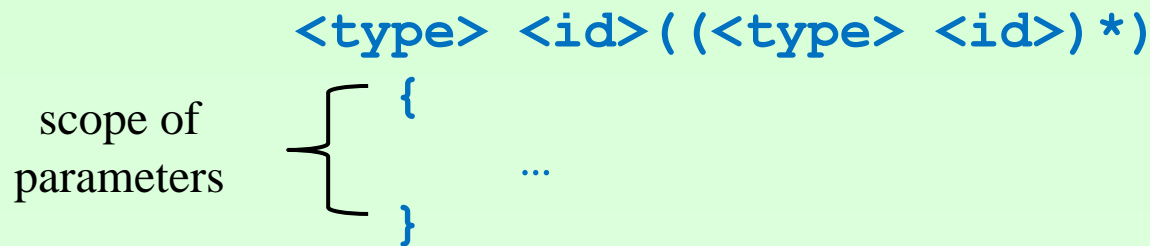
- Here, red **x** denotes previous value of **x**

# ***Binders (in C)***

- Block Declaration



- Method Declaration



# *Typeful Programming in Haskell*

- Every expression has a type.
- We use ascription  $(e :: t)$  to force an expression  $e$  to have a given type  $t$

```
let x = 3 :: Integer
```

```
let y = [1,2,3,4] :: [Int]
```

```
let t = ("Tom", 2, 4.5)  
-- type of t above will be inferred  
-- as ([Char], Integer, Double)
```

# *Typing Comparison*

- Typeful programming also in OCaml which has both object and module types, but lack type classes mechanism of Haskell.
- Both type inference and type checking possible.
- Weak typing in C with unsafe casting.
- Dynamic typing in Javascript and Python

# Functions

- Functions are also values

```
let add x y = x + y
    double x = (add x x)
    quad x = (double x)+(double x)
```

- Layout rule supports multiple and recursive declarations
- Declarations may be nested:

```
let quad x =
    let add x y = x + y
        double x = add x x
    in (double x)+(double x)
```

Here, `add` and `double` are local functions, while `quad` is a global declaration.

# *Layout Rules*

- Uses indentation to structure code
- Minimise use of separators
- Interesting MSc thesis proposal

<http://www.cse.chalmers.se/~abela/master/layout-parsing.html>



## Layout Rule

- Haskell uses two dimensional syntax to help reduce syntactic separators where declarations are “aligned”

```
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.
- See <https://en.wikibooks.org/wiki/Haskell/Indentation>

# ***Supporting Recursion***

- By default in Haskell
- Explicitly in OCaml

# Recursive Functions in Haskell

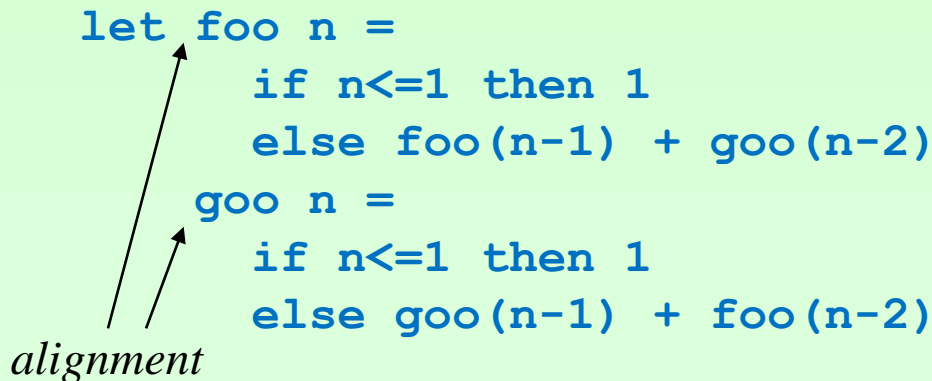
- Declarations are mutual recursive by default.

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

- Mutual-recursive functions can be defined and are aligned based on the layout rule

```
let foo n =  
    if n<=1 then 1  
    else foo(n-1) + goo(n-2)  
goo n =  
    if n<=1 then 1  
    else goo(n-1) + foo(n-2)
```

*alignment*



- Note that `goo n-1` is parsed as `(goo n) - 1`

# Recursive Methods *in OCaml*

- Declarations are *non-recursive* by default. Add **rec** keyword to capture recursive declaration.

```
let rec fact n =  
    if n=1 then 1  
    else n * fact(n-1);;
```

- Mutual-recursive functions can be defined simultaneously with the help of the **and** keyword

```
let rec foo n =  
    if n<=1 then 1  
    else foo(n-1) + goo(n-2)  
  
and goo n =  
    if n<=1 then 1  
    else goo(n-1) + foo(n-2);;
```

- Note that **goo n-1** is parsed as **(goo n) - 1**