

CptS 355- Programming Language Design

Functional Programming in Haskell Part-1

Instructor: Sakire Arslan Ay
Spring 2020



World Class. Face to Face.

Functional Programming - A Brief History

- Based on *lambda calculus* by *Alonzo Church* (1930s)
- First real language: **Lisp** by *John McCarthy* (1950s)
- Popularized by many, especially *John Backus*
- **ML** developed by *Robin Milner* at Edinburgh (1973)
- **Miranda** and **Haskell** in late 1980s

Haskell

- Haskell is:
 - pure functional programming language – there are no side effects
 - lazy evaluation – values are only computed on demand, allowing the implementation of infinite data structures
 - type system – statically typed, no type declarations needed; polymorphic
- Benefits:
 - allows for concise programs -- Haskell makes code easier to understand and maintain
 - easier to parallelize – Haskell can parallelize non-concurrent code without changing semantics
 - much cleaner mathematically
 - strong typing catches many bugs at compile time
 - functional code permits better testing methodologies

Getting started with Haskell

- Install the Haskell Platform, which includes the GHC compiler.
 - Windows: <https://www.haskell.org/platform/>
 - Mac and Linux: <https://www.haskell.org/downloads/>

- Create a file called `hello.hs` with the following contents:

```
main = putStrLn "Hello, world!"
```

- Compile your program to a native executable like this:

```
$ ghc --make hello
[1 of 1] Compiling Main                ( hello.hs, hello.o )
Linking hello ...
$ ./hello
Hello, world!
```

- Or run it in the GHCi interpreter like this:

```
$ ghci hello.hs
GHCi, version 7.0.3: http://www.haskell.org/ghc/  :? for help
...
Ok, one module loaded.
Prelude Main> main
Hello, world!
Prelude Main>
```

Getting started with Haskell

- `main` is the entry point to your program. That's where the execution starts and ends.
- If you need multiple I/O actions in one expression, you can use a `do` block

```
main = do putStrLn "What is 4 * 5?"
        x <- readLn
        if x == 20
        then putStrLn "You're right!"
        else putStrLn "You're wrong!"
```

- The indentation is significant.

Getting started with Haskell (cont.)

- Or run Haskell interpreter and load the file within Haskell :

```
$ ghci
Prelude > :load hello
Ok, one module loaded.
Prelude Main> main
Hello, world!
Prelude Main>
```

OR

```
Prelude > :l hello
```

- If you've subsequently edited the file with an external editor, use:

```
Main> :reload
```

OR

```
Main> :r
```

- To change the prompt:

```
Prelude> :set prompt "ghci> "
```

- To quit:

```
Prelude> :quit
```

OR

```
Prelude> :q
```

In Haskell every value, expression, and function has a type

- **Some basic types:**

- `Bool` - either `True` or `False`
- `Char` - a unicode code point (i.e., a character)
- `Int` - fixed-size integer
- `Integer` - an arbitrary-size integer
- `Double` - an IEEE double-precision floating-point number
- `String` – which is an alias for `[Char]`
- `type1 → type2` - a function from `type1` to `type2`
- `(type1, type2, ..., typeN)` - a tuple
- `[type1, type1, ..., type1]` - a list

- You can declare the type of a symbol or expression with `::`

```
x :: Integer
x = (1 :: Integer) + (1 :: Integer) :: Integer
```

- `::` has lower precedence than any function operators (including `+`)

Bindings

- Haskell uses the "`=`" sign to declare bindings:

```
x = 2           -- Two hyphens introduce a comment
y = 3           -- ...that continues to end of line.
main = let z = x + y  -- let introduces local bindings
      in print z     -- program will print 5
```

- Bound names cannot start with upper-case letters
- Bindings are separated by "`;`", which is usually auto-inserted by a layout rule

- A binding may declare a function of one or more arguments

- Function and arguments are separated by spaces (when defining or invoking them)

```
addOne arg = 1 + arg  -- defines function addOne
four = addOne 3        -- invokes function addOne
```

```
add arg1 arg2 = arg1 + arg2  -- defines function add
five = add 2 3               -- invokes function add
```

- Parentheses can wrap compound expressions, must do so for arguments

```
bad = print add 2 3      -- error! (print should have only 1 argument)
main = print (add 2 3)   -- ok, calls print with 1 argument, 5
```


Functions

- Functions map input values to output values

```
      head      body
add arg1 arg2 = arg1 + arg2  -- defines function add
five = add 2 3              -- invokes function add
```

```
exclaim sentence = sentence ++ "!"  -- defines function exclaim
s = exclaim "Hello"                 -- invokes function exclaim
```

Type Signatures (of functions)

- A complete function definition (with type signature) appears as follows:

```
double :: Integer -> Integer    -- type signature
double x = 2*x                  -- function expression
```

- Function type signatures are optional, but it's good form to declare them.
 - They provide documentation for other programmers and help the Haskell system to spot type errors.
- Haskell will infer the types of most things:

```
double x = 2.0 * ( x :: Double)
:t double

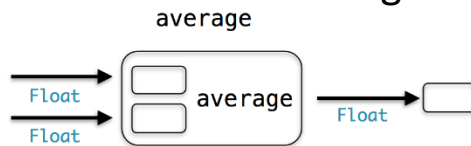
double :: Double -> Double
```

- `:type` (or `:t`) retrieves the type of a binding, expression, or function.

More on Functions

```
average :: Float -> Float -> Float
average a b = (a + b) / 2.0
a = average 3.0 4.0
```

- The type of a function with more than one argument separates the arguments with an arrow (\rightarrow)
- Function application happens one argument at a time (a.k.a. "currying")
 - You can view a function with two arguments, such as `average`, as a box with two free slots:



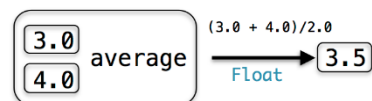
- Once the function is applied to an argument of type `Float`, the first slot is filled, and it results in a new function which maps a value `b` provided as an argument to $(3.0 + b) / 2.0$.



Will return a function of type `Float -> Float`

- Only when the second argument is provided, and all slots are filled, can the function be fully evaluated and return the result value of type `Float`:

`average 3.0 4.0`



Will return `Float`

- So the type of function “`average`” is `Float -> (Float -> Float)`

More on Functions

- Function application happens one argument at a time (a.k.a. "*currying*")

```
average :: Float -> Float -> Float
average a b = (a + b) / 2.0
a = average 3.0 4.0
```

- So `average 3.0 4.0` is equivalent to `(average 3.0) 4.0`
- `(average 3.0)` takes `4.0` returns `3.5`, so `(average 3.0)` has type `Float -> Float`
- Functions of multiple arguments that can be applied to their arguments one at a time are called *curried* functions
 - after the mathematician *Haskell B. Curry* — the Haskell language was named after him as well).
 - In Haskell, all functions of multiple arguments are curried by default.

Infix and Prefix Application of Functions

Infix notation

```
3 + 4  
2.5 * 4.0
```

Prefix notation

```
(+) 3 4  
(* ) 2.5 4.0
```

Infix notation

```
3.0 `average` 4.0  
3 `add` 4
```

Prefix notation

```
average 3.0 4.0  
add 3 4
```

Note that the
function name is in
backquotes

- Binary functions in backquotes are, by default, left associative, i.e.,

```
6.5 `average` 7.2 `average` 6.2
```

is equivalent to:

```
(6.5 `average` 7.2) `average` 6.2
```

Haskell is a pure functional language

- Unlike variables in imperative languages, Haskell bindings are:
 - *immutable* - can only bind a symbol once in a given scope (We still call bound symbols "variables" though)

```
x = 5
x = 6      -- error, cannot re-bind x
```

- *order-independent* - order of bindings in source code does not matter
- *lazy* - definitions of symbols are evaluated only when needed

```
safeDiv x y =
  let q = div x y      -- safe as q never evaluated if y == 0
  in if y == 0 then 0 else q
main = print (safeDiv 1 0)  -- prints 0
```

- *recursive* - the bound symbol is in scope within its own definition

```
x = 5      -- this x is not used in main

main = let x = 6  -- introduces new x
      in print x
```

Role of variable

- In Haskell variables are immutable
- A variable just maps to a value that it is bound to.
- There is no “assignment statement” in Haskell for changing what a variable maps to

- In imperative languages, a variable is a *location* that can hold a value, and which can be changed through an assignment.

`x = x + 1;`

How to program without mutable variables?

- In C, we use mutable variables to create loops:

```
long factorial (int n)
{
    long result = 1;
    while (n > 1)
        result *= n--;
    return result;
}
```

- In Haskell, can use recursion to "*re-bind*" argument symbols in new scope

```
factorial n = if n > 1
              then n * factorial (n-1)
              else 1
```

- Recursion often fills a similar need to mutable variables
- But the above Haskell factorial is inferior to the C one--why?

Parameterized types

- Types can have parameters sort of the way functions do:

```
myNum :: Num p => p  
myNum = 3
```

```
pi :: Floating a => a  
pi = 3.141592653589793
```

```
double :: Num a => a -> a  
double x = x * 2
```

```
equal :: Eq a => a -> a -> Bool  
equal x y = (x == y)
```

```
bigger :: Ord a => a -> a -> Bool  
bigger x y = (x > y)
```

- `Num`, `Eq`, `Ord` are all *type classes*; `p` and `a` are *type variables*.

```
concat x y = x ++ y
```

```
first a b = a
```

Parameterized types

- Here is an overview of some frequently used type classes, and some overloaded operations on these type classes.
- ❑ Typeclass **Show**
 - functions: `show :: Show a => a -> String`: convert the given value into a string.
 - member types: almost all predefined types, excluding function types.
- ❑ Typeclass **Eq**
 - functions: `(==), (/=) :: Eq a => a -> a -> Bool`: equality and inequality.
 - member types: almost all predefined types, excluding function types.
- ❑ Typeclass **Ord**
 - functions: `(<), (>), (<=), (>=) :: Ord a => a -> a -> Bool`: less than, greater than, less or equal, greater or equal
 - member types: almost all predefined types, excluding function types.
 - all types in **Ord** are already in **Eq**, so if you are using both `=` and `<` on a value, it is sufficient to require it to be in **Ord**.
- ❑ Typeclass **Num**
 - functions: `(+), (-), (*) :: Num a => a -> a -> a`: arithmetic operations.
 - member types: **Float**, **Double**, **Int**, **Integer**
- ❑ Typeclass **Integral**
 - functions: `div, mod :: Integral a => a -> a -> a`: division.
 - member types: **Int** (fixed precision), **Integer** (arbitrary precision)
- ❑ Typeclass **Fractional**
 - functions: `(/) :: Fractional a => a -> a -> a`: division.
 - member types: **Float**, **Double**

Haskell Tuples

- A tuple combines multiple components into one compound value.
 - The values in a tuple can be of different types.
 - The values in a tuple has a specific order.

```
myTuple :: (Bool, Integer, String)
myTuple = (True, 1, "one")
```

```
nestedTuple :: (Bool, (Integer, String), Double)
nestedTuple = (True, (2, "two"), 2.0)
```

- Decomposing values of a pair (a 2-tuple):

```
fst (True, (2, "two")) -- returns the first element : True
snd (True, (2, "two")) -- returns the second element (2, "two")
```

Haskell Tuples – cont.

- Example functions taking tuple as argument:

```
swap :: (Integer, String) -> (String, Integer)
swap (x,y) = (y,x)

swap (2, "two")      -- will return ("two", 2)
swap ("2", "two")    -- will give a type error
```

- Example functions returning tuples:

```
strPair :: Integer -> (Integer, String)
strPair x = (x, show x)

strPair 5  -- will return (5,"5")
```

Haskell Lists

- Haskell lists can be of arbitrary size. They can have values of various types, but all elements must be the same type.

```
tenPrimes :: [Integer]
tenPrimes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 27]
```

- We don't need to explicitly write every single element if our list elements are just a sequence of consecutive numbers — or any type whose values can be enumerated:

```
oneToTwenty :: [Integer]
oneToTwenty = [1..20]
```

```
-- return all positive odd numbers up to maxNumber
oddNumbers :: Integer -> [Integer]
oddNumbers maxNumber = [1, 3..maxNumber]
```

```
oddNumbers 10 - will return [1, 3, 5, 7, 9]
```

- The difference between tuples and lists can be seen by comparing their types:

```
(1, 2, "green") :: (Integer, Integer, String)
```

```
[1, 2, 3, 4] :: [Integer]
```

Haskell Lists – cont.

- Haskell lists can be nested or may include composite values (tuples, functions, etc.) :

```
[[1,2,3],[1,2],[3,4],[]]  
[(1,"one"),(2,"two"), (3,"three")]
```

```
> :type []  
[] :: [a] -- Polymorphic type - a is a type variable
```

- The below won't work.

```
[[1,2,3],[“1”,“2”],[3,4]]  
[(1, 2, "one"),(2, "two")]
```

- A String is just a list of Char, so `['a', 'b', 'c'] == "abc"`

Haskell List Processing

- The `:` operator appends an item to the head of an already existing list.
 - “`:`” is pronounced “cons”
 - It takes a value and a list and returns a list where the value is added to the beginning of the list.
- Examples:
 - `"blue" : []` \Rightarrow `["blue"]`
 - `"yellow":["red","green","blue"]` \Rightarrow `["yellow","red","green","blue"]`
 - `"yellow":"red":"green":"blue":[]` \Rightarrow `["yellow","red","green","blue"]`
 - `["red","green","blue"] : "yellow"` \Rightarrow *Error!*
- The cons-operator is another example of a *polymorphic* function, as it works on lists of any type.
 - The only restriction is that the element we are adding is of the same type as the elements of the list we pass as the second argument:

```
(:) :: a -> [a] -> [a]
```

Some basic list functions

- Appending two lists:

```
(++) :: [a] -> [a] -> [a]
```

```
[1,2,3] ++ [4,5,6,7] ⇒ [1,2,3,4,5,6,7]
```

```
["red","green","blue"] ++ ["yellow"] ⇒ ["red","green","blue","yellow"]
```

- Extract the element at a specific index position out of a list

```
(!!) :: [a] -> Int -> a
```

```
["zero","one","two","three","four","five","six"] !! 5 ⇒ "five"
```

```
"CptS355" !! 4 ⇒ '3'
```

- Split a list into its first element and the rest

```
head :: [a] -> a
```

```
head [0, 1, 2, 3] ⇒ 0
```

```
head "mouse" ⇒ 'm'
```

```
tail :: [a] -> [a]
```

```
tail [0, 1, 2, 3] ⇒ [1, 2, 3]
```

```
tail "mouse" ⇒ "ouse"
```


Some basic list functions – cont.

- Length of a list:

```
length :: [a] -> Integer  
length [0, 1, 2, 3] ⇒ 4
```

- Check if an item is contained in a list

```
elem :: Eq a => a -> [a] -> Bool  
elem 3 [0, 1, 2, 3] ⇒ True  
elem 6 [1, 2, 3, 4] ⇒ False  
elem 't' "CptS" ⇒ True
```

- Add up or multiply the elements of a list

```
maximum :: Ord a => [a] -> a  
minimum :: Ord a => [a] -> a  
sum :: Num a => [a] -> a  
product :: Num a => [a] -> a  
  
sum [0, 1, 2, 3] ⇒ 6  
product [1, 2, 3, 4] ⇒ 24
```

Conditionals : if/else

- Examples of *if/else* statements:

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

```
signum :: (Ord a, Num a) => a -> Integer
signum x = if x < 0 then -1 else if x == 0 then 0 else 1
```

Conditionals: *guards*

- Cascading conditional expressions are difficult to read; *guards* provide an easier syntax:

```
signum :: (Ord a, Num a) => a -> Int
signum x = if x < 0 then -1 else if x == 0 then 0 else 1
```

```
signum :: (Ord a, Num a) => a -> Int
signum x | x < 0 = -1
         | x == 0 = 0
         | x > 0 = 1
```

- The *guards* are checked in the order they are listed
- Usually, the last *guard* should catch all the cases not covered before.
 - We use the special guard `otherwise`, which always evaluates to `True`

```
signum :: (Ord a, Num a) => a -> Int
signum x | x < 0 = -1
         | x == 0 = 0
         | otherwise = 1
```

Patterns

- In ML we can access components of lists (or tuples) directly by using patterns. The context in which the identifier appears tells us the part of the structure it references.
- Examples:

```
x :: (Integer,Integer)
x = (1,2)
(h,t) = x           --h will be assigned to 1 and t will be assigned to 2

myList :: [Integer]
myList = [1,2,3]
[v1,v2,v3] = myList --v1, v2, and v3 will be assigned to 1,2, and 3, respectively.
[1,v4,3] = myList   --v4 will be assigned to 2

[1,ys] = myList      -- This won't work. Why?
(x:ys) = myList       --x will be assigned to 1 and ys will be assigned to [2,3]
```

Patterns – cont.

- An underscore (`_`) may be used as a “wildcard” or “don’t care” symbol. It matches part of a structure without defining a new binding.

```
y:_ = ['c','a','t'] -- y will be assigned to 'c'  
_:xs = ['c','a','t'] -- xs will be assigned to ['a','t'] or 'at'
```

- Patterns can be nested too:

```
x :: (Integer, Double, Integer)  
nestedTuple = (1, 3.0, 5)  
  
((_, y), _) = nestedTuple -- y will be assigned to 3.0
```

Pattern matching in functions : *cons*

- We use pattern matching to decompose lists into their first element and the rest of the list
- Head of a list (*head* in *Prelude*):

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

Partial function

This parenthesis is necessary

```
head' :: [a] -> a
head' (x:_) = x
head' [] = error "head: empty list"
```

- Tail of a list (*tail* in *Prelude*):

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

Partial function

```
tail' :: [a] -> [a]
tail' (_:xs) = xs
tail' [] = error "tail: empty list"
```

- If your functions don't cover all possible cases, you may get a run-time "Match" exception.

```
partial :: (Ord a, Num a, Num p) => a -> p
partial x | x < 0 = -1
         | x > 0 = 1
```

Pattern matching in functions : *cons*

- Check if a list is empty (*null* in *Prelude*):

```
isNull :: Eq a => [a] -> Bool  
isNull x = if x==[] then True else False
```

VS

```
isNull :: [a] -> Bool  
isNull [] = True  
isNull (x:xs) = False
```

- What is the difference between the above *isNull* definitions? (Note the difference in type)

Recursive Functions in Haskell

- Add first n natural numbers:

```
natSum :: (Num a, Ord a) => a -> a
natSum 0 = 0
natSum n | n > 0 = n + natSum (n - 1)
         | otherwise = error "natSum: Input value is negative!"
```

- Length of a list:

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

n = length [1,2,2,3]
```


Recursive Functions in Haskell

- Last element of a list:

```
last' :: [a] -> a
last' []      = error "last': Input list is empty."
last' [x]     = x
last' (x:xs)  = (last xs)

last' [1,2,3,4]
```

➤ Caution!

- Patterns are checked in order and order matters. The first matching pattern is evaluated. In the function , if you have specified the last pattern before the middle, it would not work.
- If you use the cons (:) patterns, you need to use parenthesis around it.
 - For example: `last' x:xs = (last xs)` will give an error.

Recursive Functions in Haskell

- n^{th} element: Return the n th element in a list. Assume ($n > 0$)

```
nthElement [] n = error "nthElement': The input list is too short."  
nthElement (x:xs) 1 = x  
nthElement (x:xs) n = (nthElement xs (n-1))
```

Recursive Functions in Haskell

- **Mapping**: applying an operation to every element of a list:

- Compute the square of each element in the list:

```
allSquares [x1, x2,... , xn] = [x1 * x1, x2 * x2,... , xn * xn]
```

```
allSquares :: Num a => [a] -> [a]
allSquares [] = []
allSquares (x : xs) = x * x : allSquares xs
```

- Make all letters in a string uppercase:

```
strToUpper "Cpts355" = "CPTS355"
```

```
import Data.Char

strToUpper :: String -> String
strToUpper [] = []
strToUpper (chr : xs) = (toUpper chr) : (strToUpper xs)
```

Recursive Functions in Haskell

- **Filtering**: removing elements from a list
 - Filter out the values smaller than a given value.

```
filterSmaller [] v = []  
filterSmaller (x:xs) v | (x >= v) = x:(filterSmaller xs v)  
                        | otherwise = (filterSmaller xs v)
```

- Extract digits:
 extractDigits "CptS355" = "355"

```
import Data.Char  
  
extractDigits :: String -> String  
extractDigits [] = []  
extractDigits (chr : xs) | isDigit chr = chr : extractDigits xs  
                        | otherwise = extractDigits xs
```

Recursive Functions in Haskell

- Reductions: combining the elements of a list.
- Add-up all the elements of a list:

```
addup :: Num p => [p] -> p
addup []      = 0
addup (x:xs) = x + (addup xs)

sum1 = addup [1,2,3,4,5]           -- evaluates to 15
sum2 = addup [1.0,2.0,3.0,4.0,5.0] -- evaluates to 15.0
```

- Multiply the elements in a list:

```
mul :: Num p => [p] -> p
mul []      = 1
mul (x:xs) = x * (mul xs)

p1 = mul [1,2,3,4,5]           -- evaluates to 120
```

Recursive Functions in Haskell

- Reductions: combining the elements of a list.
 - List is a recursive structure: all lists are constructed from `[]` with `cons (:)` operator.
 $[x_1, x_2, \dots, x_n] = (x_1 : (x_2 : \dots : (x_n : [])) \dots)$
 - When we combine the elements of a list, we simply replace the `cons (:)` with another operator. For example:
 $\text{mul } (x_1 : (x_2 : \dots : (x_n : [])) \dots) = (x_1 * (x_2 * \dots * (x_n * [])) \dots)$

 $\begin{aligned} \text{mul } (3:(5:(6:[]))) &\Rightarrow 3 * \text{mul } (5:(6:[])) \\ &\Rightarrow 3 * (5 * \text{mul } (6:[])) \\ &\Rightarrow 3 * (5 * (6 * \text{mul } [])) \\ &\Rightarrow 3 * (5 * (6 * 1)) \\ &\Rightarrow 90 \end{aligned}$
 - If we generalize this:
 $\text{op } (x_1 : (x_2 : \dots : (x_n : [])) \dots) = (x_1 \text{ `op` } (x_2 \text{ `op` } \dots \text{ `op` } (x_n \text{ `op` } [])) \dots)$

Recursive Functions in Haskell

- Reductions: combining the elements of a list.
 - Minimum/maximum value in a list:

```
minList :: [Int] -> Int
minList []      = maxBound
minList (x:xs) = x `min` minList xs

m = minList [2,6,1,4,3]    -- evaluates to 1
```

- `maxBound` is *Prelude* constant for the maximum `Int` value.

Recursive Functions in Haskell

- Reverse:

- We can express reverse as a reduction as well

```
reverse [x1, x2,... , xn] ⇒ [xn,... , x2, x1]
```

```
reverse (x1 : (x2 : ... : (xn : []))...) ⇒ ((([xn] ++ ..) ++ [x2]) ++ [x1])
```

- Consider the function snoc:

```
snoc x xs = xs ++ [x]
```

```
reverse (x1 : (x2 : ... : (xn : []))...) ⇒  
  (x1 `snoc` (x2 `snoc` ... `snoc` (xn `snoc` []))...)
```

- Now we can implement reverse using snoc: (we call it reverse' since reverse is already defined in Prelude)

```
reverse' :: [a] -> [a]  
reverse' [] = []  
reverse' (x:xs) = x `snoc` (reverse' xs)
```

OR

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


Recursive Functions in Haskell

- Reverse:

- What is the time complexity of `reverse'`?

```
snoc x xs = xs ++ [x]
```

```
reverse' :: [a] -> [a]
reverse' [] = []
reverse' (x:xs) = x `snoc` (reverse' xs)
```

- Later we will give a more efficient definition of `reverse`.

Recursive Functions in Haskell

- **Append:**

- We append the first list to the second.
- Do we need to capture the case where the second list is empty?

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

- If patterns overlap and if some patterns are redundant, you may get a warning from the compiler. For example:

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

 redundant pattern

let expression and *where* clause

- All the assignment statements in a Haskell module are “top-level” assignments.
- However, often one needs to make assignments inside other code in order to avoid repeated computation of values or simply to make the implementation clearer.
- Haskell offers two alternatives:
 - *let* ... *in* statement , and
 - *where* clause

let expression

- Syntax:
 - Each ***bi*** is any *binding* and ***e*** is any *expression*

```
let  b1 and
    b2 and
    ...
    bn
in   e
```
- Type-checking: Type-check each ***bi*** and ***e*** in a static environment that includes the previous bindings.
 - Type of whole let-expression is the type of ***e***.
- Evaluation: Evaluate each ***bi*** and ***e*** in a dynamic environment that includes the previous bindings.
 - Result of whole let-expression is result of evaluating ***e***.
- A let-expression is ***just an expression***, so we can use it ***anywhere*** an expression can go

let expression and *where* clause

- Consider the `reverse` function we defined before. `reverse` calls `snoc`:

```
snoc x xs = xs ++ [x]

reverse' :: [a] -> [a]
reverse' [] = []
reverse' (x:xs) = x `snoc` (reverse' xs)
```

- let ... in** introduces a variable/function before it can be used;

```
reverse' :: [a] -> [a]
reverse' [] = []
reverse' (x:xs) = let
                    snoc x xs = xs ++ [x]
                  in x `snoc` (reverse' xs)
```

- whereas **where** assigns a value to a variable after it has been used.

```
reverse' :: [a] -> [a]
reverse' [] = []
reverse' (x:xs) = x `snoc` (reverse' xs)
                  where snoc x xs = xs ++ [x]
```

let expression and *where* clause

- So far we have used both constructs interchangeably. However, there is one significant difference that is important to us.
 - A variable bound with ***let*** has a so called *scope*. That is, it is only “visible” after the ***in*** in the context of a computational block.
 - A variable bound with ***where*** is “visible” anywhere in the body of a function preceding the declaration.
 - Example:

- ***let ... in***

```
basic n = if even n then
          let two = 2
          in two * n
        else
          two * two * n
```

Will yield an
out of scope
error for ***two***.

- ***where***

```
basic n = if even n then two * n
          else two * two * n
          where two = 2
```