Lecture 2 – Haskell Primitives

- Constants (and operators)
- Variables
- Tuples
- Lists
- Functions
- Pattern matching

Constants

Integer, Float, Double, Bool, Char, String

```
    1 :: Integer
    1.2 :: Float
    1.2 :: Double
    True :: Bool
    False :: Bool
    'a' :: Char
    "a" :: String
```

Operators on constants

Most operators are Java-like except not equal (/=) and logical negation (not). String is in fact a list of characters and is combined with list concatenation operator ++.

```
1 + 2           -- 3
1 + 2.3         -- 3.3
1 / 2           -- 0.5
1 == 2          -- False
1 < 2           -- True
1 /= 2          -- True
"ab" ++ "cd"    -- "abcd"
['a', 'b'] ++ ['c'] -- "abc"
True && False -- False
True || False -- True
not False       -- True
```

Variables

Variables can be defined globally or locally. Local variable is defined in let expression of the form below, where x is visible from where it is defined to the end of the let expression.

```
let x = e1
    y = e2      -- x is visible
    z = e3      -- x, y are visible
in exp          -- x,y,z are visible
```

For example

```
let a = 1
    b = a + 1
    c = a + b + 1
in  (a, b, c)          -- evaluates to (1, 2, 4)
```

Variables

Global variable is visible within current scope. Let variable is visible within the let expression.
For example

```
x = 2                     -- x is in global scope

let y = 1                 -- y is in local scope
in  y + x                 -- evaluates to 3

z = let y = 1 in y + x -- let expression evaluates to 3
                          -- z = 3
```

Local variable can also be introduced by ‘where’ – more on this later.
Inside GHCi console, you may use ‘let’ to introduce a variable without using ‘in’.

Tuples

A tuple is a fixed collection of values (of possibly different types).

```
point = (1, 2)
color_point = (point, "red")

fst point                        -- evaluates to 1
snd point                        -- evaluates to 2

fst color_point                  -- evaluates to (1,2)
snd(fst color_point)             -- evaluates to 2
snd color_point                  -- evaluates to "red"
```

Other than ‘fst’ and ‘snd’ functions, tuple values are accessed through pattern matching (match tuple patterns with tuple values).

```
(x, y) = point                   -- x = 1, y = 2
((x, y), c) = color_point        -- x = 1, y = 2, c = "red"
```

Lists

A list is an extensible collection of values of the same type.

```
x = [1,2,3]
y = [4,5,6]

a = head x        -- a = 1
b = tail x        -- b = [2,3]

z = x ++ y        -- z = [1,2,3,4,5,6]

w = 0 : x         -- w = [0,1,2,3]

u = y !! 0        -- u = 4
v = y !! 1        -- v = 5

null x            -- False
null []           -- True
```

Functions

A function applies to an argument and evaluates to a value.

```
f x = x * x    -- square function

f 10           -- 100
f 1.5          -- 2.25
```

A function may have multiple cases.

```
lstsum [] = 0                         -- sum up a list
lstsum (a:b) = a + lstsum b

lstsum [1,2,3]                            -- evalautes to 6
lstsum [1.1, 2.2, 3.3]                    -- evaluates to 6.6
```

Functions

```haskell
lstsum [] = 0                                    -- sum up a list
lstsum (a:b) = a + lstsum b
```

The lstsum function may be implemented using if-then-else (but not as readable).

```haskell
lstsum x = if null x
             then 0
             else head x + lstsum (tail x)
```

Pattern matching

Tuples and lists should be accessed using pattern matching.

```
t = (1, 'A', "one")

(a, b, c) = t          -- a = 1, b = 'A', c = "one"

(a, _, _) = t          -- a = 1, _ means don't care

middle (_, x, _) = x

middle t               -- evaluates to 'A'
```

Pattern matching

Tuples and lists should be accessed using pattern matching.

```
t = [1,2,3,4]

[w,x,y,z] = t       -- w = 1, x = 2, y = 3, z = 4

a:b = t             -- a = 1, b = [2,3,4]

a:b:c = t           -- a = 1, b = 2, c = [3,4]

a:_ = t             -- a = 1

_:b = t             -- b = [2,3,4]

_:_:c = t           -- c = [3,4]

_:b:_ = t           -- b = 2
```

Constant pattern matches constant exactly. For example, 0 only matches 0 and [] only matches [].

Example

Split a list into two halves.

```
split [] = ([], [])
split [a] = ([a], [])
split (a:b:c) = let (x,y) = split c
                in (a:x, b:y)

split [1,2,3,4,5,6]       -- evaluates to ([1,3,5], [2,4,6])
```

Merge two lists by increasing order

```
merge x [] = x
merge [] x = x
merge (x1:r1) (x2:r2) =
  if x1<x2
    then x1 : merge(r1, x2:r2)
    else x2 : merge(x1:r1, r2)

merge [1,3,5] [2,4,6]     -- evaluates to [1,2,3,4,5,6]
```

Example

```
split [] = ([], [])
split [a] = ([a], [])
split (a:b:c) = let (x,y) = split c
                in (a:x, b:y)
```

```
merge x [] = x
merge [] x = x
merge (x1:r1) (x2:r2) =
   if x1<x2
     then x1 : merge r1 (x2:r2)
     else x2 : merge (x1:r1) r2
```

Mergesort function sorts lists using ‘split’ and ‘merge’ functions.

```
mergesort [] = []
mergesort [a] = [a]
mergesort x =
   let (a, b) = split x
   in merge (mergesort a) (mergesort b)

mergesort [5,2,1,9,11,73,0]    -- evaluates to [0,1,2,5,9,11,73]
```

Example

Local functions (or variables) can be defined using 'where' keyword.

```
mergesort [] = []
mergesort [a] = [a]
mergesort x =
   let (a, b) = split x
   in merge (mergesort a) (mergesort b)

 where
   split [] = ([], [])
   split [a] = ([a], [])
   split (a:b:c) = let (x,y) = split c
                   in (a:x, b:y)

   merge x [] = x
   merge [] x = x
   merge (x1:r1) (x2:r2) =
      if x1<x2
        then x1 : merge r1 (x2:r2)
        else x2 : merge (x1:r1) r2
```

Example

The split function can be simplified.

```
mergesort [] = []
mergesort [a] = [a]
mergesort x =
   let (a, b) = split x
   in merge (mergesort a) (mergesort b)

 where
   split x = (take n x, drop n x)    -- take/drop first n elements
     where n = length x `div` 2      -- `div` is integer division

   merge x [] = x
   merge [] x = x
   merge (x1:r1) (x2:r2) =
      if x1<x2
        then x1 : merge r1 (x2:r2)
        else x2 : merge (x1:r1) r2
```

Example

Merge function can be improved a little using *as patterns*.

```
mergesort [] = []
mergesort [a] = [a]
mergesort x =
   let (a, b) = split x
   in merge (mergesort a) (mergesort b)

 where
   split x = (take n x, drop n x)    -- take/drop first n elements
     where n = length x `div` 2      -- `div` is integer division

   merge x [] = x
   merge [] x = x
   merge l1@(x1:r1) l2@(x2:r2) =
      if x1<x2
        then x1 : merge r1 l2
        else x2 : merge l1 r2
```

Example

Quicksort can be implemented using pattern matching as well.

```
quicksort [] = []
quicksort [x] = [x]
quicksort (pivot:x) =
   let split [] = ([], [])
       split (a:b) = if a < pivot
                        then (a:left, right)
                        else (left, a:right)
          where (left, right) = split b

       (lower, upper) = split x

   in quicksort lower ++ (pivot : quicksort upper)
```

Example

Selection sort works well for short lists.

```
selectsort [] = []
selectsort [a] = [a]
selectsort x = f x []
  where
    f [] sofar = sofar
    f l sofar = f rest (max:sofar)

       where (max, rest) = select l

             select [a] = (a, [])
             select (a:b) = if a > c then (a, b) else (c, a:d)
                 where (c, d) = select b
```

Example

Quicksort can switch to selection sort for shorter lists.

```
quicksort x =
  if length x < 10
   then selectsort x
   else
     let pivot:rest = x
         split [] = ([], [])
         split (a:b) = if a < pivot
                       then (a:left, right)
                       else (left, a:right)
           where (left, right) = split b

         (lower, upper) = split rest

     in quicksort lower ++ (pivot : quicksort upper)
```

Example

If-then-else in quicksort can be replaced by *guards*.

```
quicksort x
  | length x < 10 = selectsort x
  | otherwise =
     let pivot:rest = x
         split [] = ([], [])
         split (a:b) = if a < pivot
                       then (a:left, right)
                       else (left, a:right)
             where (left, right) = split b

         (lower, upper) = split rest

     in quicksort lower ++ (pivot : quicksort upper)
```

Run tests

It is easier to write your Haskell program in an IDE or edit your program a text editor and then run it through GHCi. Below is a main program for testing the sorting functions.

```haskell
import System.Random  -- this is the first line of the program

-- definitions of the sorting functions are placed here

main :: IO ()
main = do
        g <- getStdGen    -- random number generator
        let n = 20
        -- generate a list of 20 integers between 0 and 100
        let x = take n $ (randomRs (0, 100) g :: [Integer])
        print (quicksort x)
        print (mergesort x)
        print (selectsort x)
```

Run tests

We can also use QuickCheck, which is an automatic test generator.

```haskell
import Test.QuickCheck -- this is the first line of the program

-- definitions of the sorting functions are placed here

main :: IO ()
main = do
       let ordered :: [Integer] -> Bool  -- check if list ordered
           ordered xs = and (zipWith (<=) xs (drop 1 xs))
       let prop_q x = ordered (quicksort x)  -- boolean function
       let prop_m x = ordered (mergesort x)  -- boolean function
       let prop_s x = ordered (selectsort x) -- boolean function
       quickCheck prop_q                     -- run 100 tests
       quickCheck prop_m                     -- run 100 tests
       quickCheck prop_s                     -- run 100 tests
```

Install missing packages

Some packages such as System.Random are missing from GHCi. To install them, run the following commands in GHCi console and then restart it.

```
import System.Process
system "cabal update"
system "cabal install Random"     -- installs System.Random
system "cabal install QuickCheck" -- installs Test.QuickCheck
```