FUNCTIONAL PROGRAMMING
NO.6 BASIC VALUES

Tatsuya Hagino
hagino@sfc.keio.ac.jp

Slide URL

https://vu5.sfc.keio.ac.jp/slide/

Basic Values and Types

- Haskell has the following basic values and types:
  - Boolean values
    - `Bool`
  - Numerical values
    - `Int`, `Integer`, `Float`, `Double`
  - Character values
    - `Char`
  - Character string values
    - `String` = `[Char]`
  - Tuple values
    - `(a,b)`
  - Unit value
    - `()`
  - List
    - `[a]`
  - Function
    - `a -> b`

Boolean Values

- **`Bool`**

- There are only two Boolean values
  - **`True`**
  - **`False`**

- Functions for **`Bool`**:

Function	Usage	Meaning
`not::Bool -> Bool`	`not x`	If **`x`** is **`True`**, it returns **`False`**. If **`x`** is **`False`**, it returns **`True`**.
`(&&)::Bool->Bool->Bool`	`x && y`	If both **`x`** and **`y`** are **`True`**, it returns **`True`**. Otherwise, it returns **`False`**.
`(||)::Bool->Bool->Bool`	`x || y`	If **`x`** or **`y`** are **`True`**, it returns **`True`**. Otherwise, it returns **`False`**.

Exercise 6-1

- Define **`not`** using **`if`** function.

```
not x = if x then False else True
```

- Define it with pattern match.

```
not True = ...
not False = ...
```

- Define **`(&&)`** and **`(||)`** using **`if`** function.
- Define them with pattern match.

logic.hs

```
x && y = if x then ...
x || y = if x then ...

True && True = ...          True || True = ...
True && False = ...         True || False = ...
False && True = ...         False || True = ...
False && Flase = ...        False || False = ...
```

Numerical Values

- Integer values
  - **`Int`** small integer numbers with sign
  - **`Integer`** unlimited integer
- Integer literals
  - decimal **`5`**, **`999`**, **`123456789012345678901`**
  - octal **`0o644`**
  - hex **`0x1f`**
  - **`Int`** or **`Integer`** depends on the context.
  - Can be specified explicitly: **`(16::Int)`**
- Floating point numbers
  - Float single precision floating point number
  - Double double precision floating point number
- Floating point literals
  - **`1.5`**
  - **`3.141592`**
  - **`0.1543e+2`**
  - **`1343e-3`**
  - **`Float`** or **`Double`** depends on the context.
  - **`(1.5::Double)`**

Numerical Operations

Usage	Meaning
`x + y`	`x` add `y`
`x - y`	`x` subtract `y`
`x * y`	`x` multiply by `y`
`x / y`	`x` divide by `y` (for floating points only)
`x `div` y`	`x` divide by `y` (for integer only, round toward negative infinity)
`x `quot` y`	`x` divide by `y` (for integer only, round toward zero)
`x `mod` y`	reminder of `(x `div` y)` (for integer only)
`x `rem` y`	reminder of `(x `quot` y)` (for integer only)
`x ^ y`	`x` power of `y`
`-x`	negate `x`
`negate x`	same as `-x`
`subtract x y`	same as `(y - x)`
`abs x`	absolute value of `x`
`odd x`	`True` if `x` is an odd number
`even x`	`True` if `x` is an even number

Numerical Value Conversion

- Convert integer values to other typed values.

Usage	Meaning
`toInteger x`	convert `Int` value to `Integer` value
`fromInteger x`	convert `Integer` value to numerial value (actual type depends on the context)
`fromIntegral x`	convert `Int` or `Integer` value to numerial value (actual type depends on the context)

- Convert floating point numbers to integer values

Usage	Meaning
`ceiling x`	the smallest integer which is greater than or equal to `x`
`floor x`	the largest integer which is less than or equal to `x`
`truncate x`	the closest integer to `x` which is between `x` and `0` (including `x` itself)
`round x`	the closest integer to `x` (if there are two such integers, choose even number)

Exercise 6-2

- Given a price without VAT, calculate the price with VAT.
  - Vat in Japan is 10%.
  - Round the fraction.
    - 10.5 → 11
    - 10.4 → 10

vat.hs

```
import System.Environment

main = print $ vat $ read $ head args

vat :: Integer -> Integer
vat x =
```

Characters

- **`Char`**
  - a unicode character

- Character literals:
  - `'a'`
  - `'*'`
  - `' '`

- Escape sequences (special characters):

Usage	Meaning
`'\t'`	tab
`'\n'`	new line
`'\r'`	carriage return
`'\v'`	vertical tab
`'\f'`	next page
`'\a'`	bell
`'\b'`	backspace

Usage	Meaning
`'\NNN'`	character code with decimal **`NNN`**
`'\oNN'`	character code with octal **`NN`**
`'\xNN'`	character code with hex **`NN`**
`'\^X'`	control **`X`**
`'\''`	single quote
`'\"'`	double quote
`'\\'`	backslash (or \ symbol)

Character Strings

- **`String`**
  - a list of characters
  - same as **`[Char]`**

- String literals:
  - **`"string"`**
  - **`"Meisei"`**
  - **`"abc\ndef\n\"hello\65\tend"`**

Functions for Characters (1)

- Checking the kind of characters（type is Char -> Bool)
- Need to import **`Data.Char`**

Usage	Meaning
`isAlpha c`	if **`c`** is a unicode alphabet, then **`True`**
`isLower c`	if **`c`** is a unicode small alphabet, then **`True`**
`isUpper c`	if **`c`** is a unicode upper alphabet, then **`True`**
`isAlphaNum c`	if **`c`** is a unicode alphabet or number, then **`True`**
`isDigit c`	if **`c`** is a unicode number (**`0`**～**`9`**), then **`True`**
`isHexDigit c`	if **`c`** is a unicode hex number (**`0`**～**`9`**, **`a`**～**`f`**, **`A`**～**`F`**), then **`True`**
`isOctDigit c`	if **`c`** is a unicode number (**`0`**～**`7`**), then **`True`**
`isSpace c`	if **`c`** is a unicode space (space, tab, new line, etc.), then **`True`**
`isAscii c`	if **`c`** is an ascii alphabet (**`'\0'`**～**`'\127'`**), then **`True`**
`isLatin1 c`	if **`c`** is a Latin 1 alphabet (**`'\0'`**～**`'\255'`**), then **`True`**
`isPrint c`	if **`c`** is a unicode printable character, then **`True`**
`isControl c`	if **`c`** is not a unicode printable character, then **`True`**

Functions for Characters (2)

- Convert small letters to upper letters, and vice vasa
- Need to import **`Data.Char`**

Function	Usage	Meaning
`toLower :: Char -> Char`	`toLower c`	If `c` is an upper alphabet, returns its lower alphabet. Otherwise, it returns `c` itselft.
`toUpper :: Char -> Char`	`toUpper c`	If `c` is an lower alphabet, returns its upper alphabet. Otherwise, it returns `c` itselft.

- Convert characters to codes, and vice vasa

Function	Usage	Meaning
`ord :: Char -> Int`	`ord c`	returns the character code of `c`
`chr :: Int -> Char`	`chr n`	returns the character of which code is `n`

Tuples

- Tuples
  - Combine several elements with different types.
  - Tuple types depends on the number of elements and their order.

- Example of tuples
  - `(3, "string")` `:: (Int, String)`
  - `("lucky", 7)` `:: (String, Int)`
  - `(1, "string", [5, 4, 3])` `:: (Int, String, [Int])`
  - `('a', "string", (1, 3))` `:: (Char, String, (Int, Int))`

- **`Unit`**
  - 0 element tuple
  - `()` `:: ()`

Functions for Tuples

- **`fst :: (a, b) -> a`**
  - returns the first element from a two element tuple (a pair).
  - **`fst (1, 2)`** → **`1`**
  - **`fst ("key", "value")`** → **`"key"`**

- **`snd :: (a,b) -> b`**
  - returns the second element from a two element tuple.
  - **`snd (1, 2)`** → **`2`**
  - **`snd ("key", "value")`** → **`"value"`**

- **`zip :: [a] -> [b] -> [(a, b)]`**
  - **`zip xs ys`** returns a list of pairs taking elements from two lists **`xs`** and **`ys`**
  - **`zip [1, 2, 3] [4, 5, 6]`** → **`[(1, 4), (2, 5), (3, 6)]`**
  - **`zip [1, 2, 3] ["a", "b"]`** → **`[(1, "a"), (2, "b")]`**

- **`unzip :: [(a, b)] -> ([a], [b])`**
  - reverse of **`zip`** function
  - transforms a list of pairs to a list of first elements and a list of second element.
  - **`unzip [(1, 4), (2, 5), (3, 6)]`** → **`([1, 2, 3], [4, 5, 6])`**
  - **`unzip [(1, "a"), (2, "b")]`** → **`([1, 2], ["a", "b"])`**

List

- a list of values from the same type.
  - cannot mix different type values.
  - one direction list
  - can only traverse from head to tail, not from tail to head.

- List types:
  - `[a]`
  - `[Char]`
  - `[Int]`

- List examples:

`[]`	`::[a]`
`[1, 2, 3]`	`::[Int]`
`['a', 'b', 'c']`	`::[Char]=String`
`["aa", "bb", "cc"]`	`::[String]`
`[['a', 'a'], ['b', 'b'], ['c', 'c']]`	`::[[Char]]`

- a character list can be written as a string literal
  - `"Hello, World\n"`

: Operator

- `(:) :: a -> [a] -> [a]`
  - `x : xs`
  - returns a list by adding `x` in front of `xs`

- Example:
  - `1 : [2, 3]` → `[1, 2, 3]`
  - `'a' : "bc"` → `"abc"`

- `:` operator is right associative (evaluate right to left)
  - `1:2:[] = 1:(2:[])`
  - `[1, 2, 3] = 1 : 2 : 3 : []`

`elem` Function

- **`elem :: a -> [a] -> Bool`**
  - **`elem x xs`**
  - checks whether **`x`** appears in **`xs`** or not.

- Example:
  - **`elem 3 [2, 3, 5]`** → **`True`**
  - **`elem 3 [2, 4, 6]`** → **`False`**
  - **`3 `elem` [2, 3, 5]`** → **`True`**

- **`notElem :: a -> [a] -> Bool`**
  - the nagation of **`elem`**.

- **`elem`** can be defined as follows:

```
elem a []     = False
elem a (x:xs) = if a == x then True else elem a xs
```

Arithmetic Sequences

- Special syntax for arithmetic sequences.
  - list of numbers or characters.
  - `[1..7] = [1, 2, 3, 4, 5, 6, 7]`
  - `['a'..'e'] = ['a', 'b', 'c', 'd', 'e']`

- Change the increment (or decrement) value
  - `[1,3..11] = [1, 3, 5, 7, 9, 11]`
  - `[10,8..1] = [10, 8, 6, 4, 2]`

- Infinite lists
  - `[1..] = [1, 2, 3, 4, 5, 6, 7, 8, ····]`
  - `[1,3..] = [1, 3, 5, 7, 9, 11, ····]`

Functions for Lists

Function	Usage	Meaning
`length::[a]->Int`	`length xs`	returns the length of list `xs`
`take::Int->[a]->[a]`	`take n xs`	returns the prefix of `xs` of length `n`
`reverse::[a]->[a]`	`reverse xs`	returns the reverse list of `xs`
`(++)::[a]->[a]->[a]`	`xs ++ ys`	returns the concatenated list of `xs` and `ys`
`concat::[[a]]->[a]`	`concat xs`	concatenates all the elements in `xs`
`replicate::Int->a->[a]`	`replicate n x`	returns a list of `n` elements of `x`
`lines::String->[String]`	`lines cs`	returns a list of strings by separating `cs` by lines
`unlines::[String]->String`	`unlines xs`	concatenates strings in `xs` by adding new lines
`words::String->[String]`	`words cs`	returns a list of strings by separating `cs` by words
`unwords::[String]->String`	`unwords xs`	concatenates strings in `xs` by adding spaces
`map::(a->b)->[a]->[b]`	`map f xs`	returns a list by applying `f` to elements of `xs`
`concatMap::(a->[b])->[a]->[b]`	`concatMap f xs`	applies `f` to elements of `xs` and concatenates the result lists
`filter::(a->Bool)->[a]->[a]`	`filter f xs`	returns a list by selecting elements from `xs` which satisfies `f`
`any::(a->Bool)->[a]->Bool`	`any f xs`	checks whether there is an element in `xs` which satisfies `f`
`head::[a]->a`	`head xs`	returns the head element of `xs`
`tail::[a]->[a]`	`tail xs`	returns the list of removing the head element of `xs`
`null::[a]->Bool`	`null xs`	checks whether `xs` is empty or not
`elem::a->[a]->Bool`	`x `elem` xs`	checks whether `x` appears in `xs` or not

Data.List Module Function	Usage	Meaning
`tails::[a]->[[a]]`	`tails xs`	`[xs, (tail xs), (tail(tail xs)), ...]`
`isPrefixOf::(Eq a)=>[a]->[a]->Bool`	`xs `isPrefixOf` ys`	checks whether `xs` is the prefix of `ys`
`sort::(Ord a)=>[a]->[a]`	`sort xs`	sorts the elements in `xs`
`group::(Eq a)=>[a]->[[a]]`	`group xs`	groups the consecutive same elements

List Comprehensions

- Collects elements which satisfies the given condition.
  - similar to **`filter`** function
  - **`filter :: (a -> Bool) -> [a] -> [a]`**

- Examples:
  - **`[abs x | x <- xs ]`**
    - for each element in **`xs`**, collects **`(abs x)`**
  - **`[(x, y) | x <- [1, 2, 3], y <- ['a', 'b', 'c']]`**
    - for each element **`x`** in **`[1, 2, 3]`** and each element **`y`** in **`['a', 'b', 'c']`**, collects **`(x, y)`**
    - **`[(1, 'a'), (1, 'b'), (1, 'c'), (2, 'a'), (2, 'b'), (2, 'c'), (3, 'a'), (3,'b'), (3, 'c')]`**

- Quick sort function:

```
qsort []     = []
qsort (p:xs) = qsort lt ++ [p] ++ qsort gteq
               where
                 lt   = [x | x <- xs, x < p]
                 gteq = [x | x <- xs, x >= p]
```

Exercise 6-3

- Calculate the sum of square of odd numbers from 1 to 99.
  - $1^2+3^2+5^2+\cdots+99^2$

os1.hs

```
square n = n * n
main = print $ sum ...
```

- Use list comprehension

os2.hs

```
main = print $ sum [ ... | ... ]
```

- Write it without using built in list functions, but defining your own recursive function.

os3.hs

```
main = print $ os 99

os n = if n == 1 then ... else ...
```

cat -n Command

- UNIX cat command adds line numbers if -n option is specified.
- Write a similar command catn.hs which adds line number to a file.

catn.hs

```
main = do cs <- getContents
          putStr $ numbering cs

numbering :: String -> String
numbering cs = unlines $ map format $ zipLineNumber $ lines cs

zipLineNumber :: [String] -> [(Int, String)]
zipLineNumber xs = zip [1..] xs

format :: (Int, String) -> String
format (n, line) = rjust 4 (show n) ++ " " ++ line

rjust :: Int -> String -> String
rjust width s = replicate (width - length s) ' ' ++ s
```

catn.hs

- **`numbering cs`**
  - divides **`cs`** into lines, add line numbers, and concatenates them
- **`zipLineNumber xs`**
  - assigns line numbers to each line
  - **`zip [1..] xs`**
  - zips an infinite list of numbers starting from **`1`** and **`xs`**
- **`format (n, line)`**
  - adds the line number **`n`** in front of **`line`**
  - needs to pad spaces to make each number 4 characters length
- **`rjust width s`**
  - pads spaces in front of **`s`** to create a string of **`width`** length
  - right justification
- **`show`**
  - **`show :: (Show a) => a -> String`**
  - **`show x`**
  - convers value **`x`** to a string

Exercise 6-4

- Modify fgrep.hs to adds line number in front of matched lines.
  - The line number should be the line number in the input file, not the line number of output.

fgrepn.hs

```
import System.Environment
import Data.List

main = do args <- getArgs
          cs <- getContents
          putStr $ fgrep (head args) cs

fgrep :: String -> String -> String
fgrep pattern cs = unlines $ filter match $ lines cs
  where
    match :: String -> Bool
    match line = any prefixp $ tails line

    prefixp :: String -> Bool
    prefixp line = pattern `isPrefixOf` line
```

This is fgrep.hs
no line number version

```
% ./fgrepn ar < USA-states.txt
0007 CT  Connecticut      Hartford
0008 DE  Delaware Dover
0020 MD  Maryland Annapolis
...
```