FUNCTIONAL PROGRAMMING
NO.3 TYPE AND HIGHER ORDER FUNCTION

Tatsuya Hagino
hagino@sfc.keio.ac.jp

lecture slide URL

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

Type and Value

- Values are grouped by their types.
  - Each type is a set of values.
- Haskell does static type checking.
  - It checks when it compiles. (vs.. run time type checking)
  - If there is a type mismatch, it reports error and cannot compile.
- Haskell does type inference.
  - No needs to specify types explicitly.
  - The type inference infers types automatically.
  - Sometimes, it is necessary to specify the type information in order to help the type inference (or to avoid ambiguity).

Basic Types

Type	Meaning	Example of Values
`Int`	integer	`-1, 0, 1, 777`
`Char`	character	`'a', 'A', '\n'`
`String`	list of characters	`"abc", "string"`
`Bool`	truth value	`True, False`
`[〇〇]`	list of 〇〇	`[1, 2, 3]`

- Truth value
  - **`True`** and **`False`**
- List
  - **`[Int]`** type for list of integers
  - **`[Char]`** type for list of characters (alias is **`String`**)

Type of Functions

type of first argument `->` type of second argument `->` `····` `->` type of return value

- type of function '**`lines`**'
  - String -> [String]
- type of function '**`unlines`**'
  - [String] -> String
- type of '**`firstNLines`**'
  - firstNLines n cs = unlines $ take n $ lines cs
    - Int -> String -> String

Type Variables

- '**`length`**' function
  - **`length [1, 2, 3]`**
  - **`length ['a', 'b']`**
  - **`length ["abc", "def"]`**
  - '**`length`**' can be applied to a list of any type
  - polymorphic function
  - type of '**`length`**' function
    - **`[a] -> Int`**
    - '**`a`**' is a type variable. '**`a`**' can be any type.

- '**`take`**' function
  - **`take 3 [1, 2, 3]`**
  - type of '**`take`**' function
    - **`Int -> [a] -> [a]`**

To check the type in ghci

```
Prelude> :type take
take :: Int -> [a] -> [a]
```

Type Declaration

```
variable :: type
```

- Specify the type of variable

```
function :: type1 -> type2 -> .... -> type
```

- Specify the type of function
  - Declare it before defining the function
  - The definition is checked against the declaration
  - Helps type inference

```
length  :: [a] -> Int
reverse :: [a] -> [a]
take    :: Int -> [a] -> [a]
words   :: String -> [String]
lines   :: String -> [String]
unlines :: [String] -> String
print   :: (Show a) => a -> IO()
putStr  :: String -> IO()
putStrln:: String -> IO()
```

Higher Order Functions

- Functions are values.
  - Functions can be handled like integer values.

- Higher order functions
  - A function which takes functions are arguments
  - A function which returns functions

- '**`map`**' function
  - **`map :: (a -> b) -> [a] -> [b]`**
  - **`map square [1, 2, 3]`**
    → **`[(square 1), (square 2), (square 3)]`**
    → **`[1, 4, 9]`**
  - where **`square n = n * n`**
  - **`map f xs`**
    - returns a list by applying **`f`** to each element of **`xs`**

expand command

expand.hs

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

expand :: String -> String
expand cs = map translate cs

translate :: Char -> Char
translate c = if c == '\t' then '@' else c
```

- This program replaces tabs with '@'

```
% stack ghc expand.hs
...
% expand < USA-states.txt
AK@Alaska@Juneau
AL@Alabama@Montgomery
...
```

if expression

```
if cond then exp1 else exp2
```

- Not if statement
  - similar to conditional operator '`cond?exp1:exp2`' of C, Java and Javascript
- Value of if expression
  - If the value of ***cond*** is **`True`**, the value is exp_1, otherwise the value is exp_2.

```
translate c = if c == '\t' then '@' else c
```

- If **`c`** is a tab character, the value is **`'@'`**, otherwise the value is **`c`**
  - **`translate '\t'`** → **`'@'`**
  - **`translate 'a'`** → **`'a'`**
  - **`translate '\n'`** → **`'\n'`**

expand function

```
expand :: String -> String
expand cs = map translate cs
```

- **`translate`** convers a tab character to '@'
  - **`translate :: Char -> Char`**

- **`map translate cs`**
  - **`cs`** is a string (a list of Char)
  - **`map`** applyes **`translate`** to each character in **`cs`**
  - **`map translate "abc\tdef\n"`**
    - → `map translate ['a', 'b', 'c', '\t', 'd', 'e', 'f', '\n']`
    - → `[(translate 'a'), (translate 'b'), (translate 'c'), (translate '\t'), (translate 'd'), (translate 'e'), (translate 'f'), (translate '\n')]`
    - → `['a', 'b', 'c', '@', 'd', 'e', 'f', '\n']`
    - → `"abc@def\n"`

== operator

- **==** is a binary operator.
  - **`x == y`**
    - returns **`True`** if **`x`** and **`y`** are equal
    - otherwise returns **`False`**

- **`(==)`** is a function
  - **`(==) :: a -> a -> Bool`**
    - '**`(==) x y`**' is same as '**`x == y`**'

Writing if expression

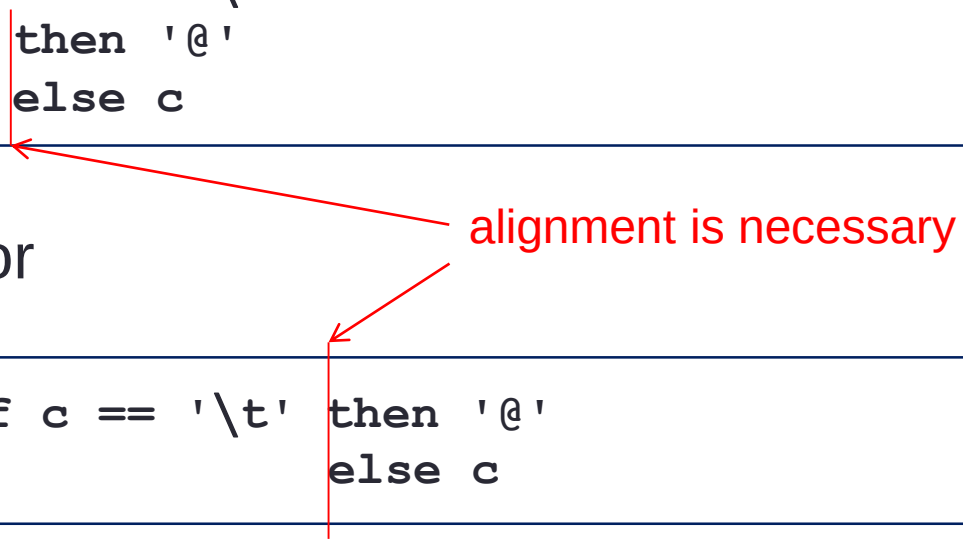
```
if c == '\t' then '@' else c
```

- If expressions of then and else are large, they can be moved to the next line.

```
if c == '\t'
  then '@'
  else c
```

- or

```
if c == '\t' then '@'
             else c
```



Exercise 3-1

- Write a command tab.hs which adds a tab at the beginning of each line.

tab.hs

```
main = do cs <- getContents
          putStr $ unlines $ map addTab $ lines cs

addTab :: String -> String
addTab cs = ...
```

- **`addTab cs`** adds **`'\t'`** at the beginning of **`cs`**
  - **`++`** operator concatenates two lists.
  - **`[1, 2, 3] ++ [4, 5]`** → **`[1,2,3,4,5]`**
  - **`"abc" ++ "def"`** → **`"abcdef"`**

```
% stack ghc tab.hs
...
% tab < USA-states.txt
        AK      Alaska   Juneau
        AL      Alabama  Montgomery
        AR      Arkansas Little Rock
...
```

expand（ver.2）command

expand2.hs

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

expand :: String -> String
expand cs = concat $ map expandTab cs

expandTab :: Char -> String
expandTab c = if c == '\t' then "        " else [c]
```

- The command replaces a tab to eight spaces.

```
% stack ghc expand2.hs
...
% expand2 < USA-states.txt
AK        Alaska          Juneau
AL        Alabama          Montgomery
...
```

concat

```
concat :: [[a]] -> [a]
```

- Given a list of lists, '**concat**' concatenates all the list and returns a single list
  - It convers a double list to a single list (makes it flat)
  - **concat [[1, 2], [3], [4, 5]]** → **[1, 2, 3, 4, 5]**
  - **concat [[1, 2], [], [3]]** → **[1, 2, 3]**
  - **concat ["ab", "c", "de"]** → **"abcde"**
  - **concat ["ab", "", "cd"]** → **"abcd"**

expand（ver.3）command

expand3.hs

```
tabStop = 8

main = do cs <- getContents
          putStr $ expand cs

expand :: String -> String
expand cs = concatMap expandTab cs

expandTab :: Char -> String
expandTab '\t' = replicate tabStop ' '
expandTab c    = [c]
```

- Rewite '**expandTab**' using pattern match.
- '**concatMap**' is the combination of '**concat**' and '**map**'

```
% stack ghc expand3.hs
...
% expand3 < USA-states.txt
AK        Alaska         Juneau
AL        Alabama         Montgomery
...
```

Defining a function using pattern match

```
expandTab :: Char -> String
expandTab '\t' = replicate tabStop ' '
expandTab c    = [c]
```

- Depending on argument value patterns, matched body is evaluated.
  - If the argument is `'\t'`, selects the first definition: `replicate tabStop ' '`
  - otherwise, selects the second definition: `[c]`

$$
\begin{array}{llll}
function & pattern_{11} & pattern_{12} \cdots & = definition_1 \\
function & pattern_{21} & pattern_{22} \cdots & = definition_2 \\
function & pattern_{31} & pattern_{32} \cdots & = definition_3 \\
\vdots & \vdots & \vdots & \vdots \quad \vdots \\
\vdots & \vdots & \vdots & \vdots \quad \vdots
\end{array}
$$

concatMap and replicate functions

- **`concatMap`** function
  - **`concatMap :: (a -> [b]) -> [a] -> [b]`**
  - **`concat`** and **`map`** are combined.
  - **`concatMap f xs = concat $ map f xs`**

- **`replicate`** function
  - **`replicate :: Int -> a -> [a]`**
  - **`replicate n x =`** returns a list which contains **`n`** number of **`x`**
  - **`replicate 3 True → [True, True, True]`**
  - **`replicate 3 77 → [77, 77, 77]`**
  - **`replicate 3 'a' → "aaa"`**
  - **`replicate 0 True → []`**

Exercise 3-2

lower.hs

```
main = do cs <- getContents
          putStr $ map lower cs

lower :: Char -> Char
lower 'A' = 'a'
...
lower c = ...
```

- Convert capital letters into small letters.

```
% stack ghc lower.hs
...
% ./lower < USA-states.txt
ak       alaska   juneau
al       alabama  montgomery
ar       arkansas little rock
...
```

Exercise 3-3

sfc.hs

```
main = do cs <- getContents
          putStr ...
```

- Write a program sfc.hs which replaces all the occurrence of '**S**' to "Shonan ", 'F' to "Fujisawa " and 'C' to "Campus " in a given input..
  - **'S' → "Shonan "**
  - **'F' → "Fujisawa "**
  - **'C' → "Campus "**

```
% stack ghc sfc.hs
...
% ./sfc
SFC
Shonan Fujisawa Campus
Super Car
Shonan upper Campus ar
...
```

input

output

Summary of functions

function	example	description
`map`	`map f xs`	returns a list by applying `f` to elements in list `xs`
`concat`	`concat xs`	concatenates elements of list `xs` and returns a single list
`concatMap`	`concatMap f xs`	`concat(map f xs)`
`replicate`	`replicate n x`	returns a list of `x` with `n` times
`(==)`	`x == y`	returns `True` when `x` and `y` are equal
`(++)`	`xs ++ ys`	concatenates list `xs` and list `ys`
`show`	`show x`	converts value `x` to a string