FUNCTIONAL PROGRAMMING
NO.7 BASIC SYNTAX

Tatsuya Hagino
hagino@sfc.keio.ac.jp

Slide URL

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

Comment

- One line comment
  - from `--` to the end of line is comment

```
square n = n * n -- square of n
```

- Block comment
  - from `{-` to `-}` is a comment
  - Block comments can be nested.

```
{-
This function is not used now.
square n = n * n {- square of n -}
-}
```

Literate Format

- Puts Haskell programs in a document
  - The extension is .lhs

```
> main = print $ square 5

Function square returns the square of n.

> square :: Int -> Int
> square n = n * n
```

```
\begin{code}
main = print $ square 5
\end{code}

Function square returns the square of n.
\begin{code}
square :: Int -> Int
square n = n * n
\end{code}
```

Layout and Brace Syntax

- Layout syntax groups lines with same indentation.

cat.hs

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

- Using brace syntax with **{ }** and ;, groups expressions without aligning indentation.

```
main = do { cs <- getContents;
            putStr cs }
```

no semicolon at the end

```
main = do { cs <- getContents; putStr cs }
```

```
main = do {
cs <- getContents
; putStr cs }
```

Offside Rule and Continuation

- In layout syntax, multiple expressions are grouped by aligning the indent.
  - The indentation is called offside line.

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

offside line of do expression

offside line of Main module

- If a line has more indentation than the current offside line, it is treated as continuation of the previous line.

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

offside line of do expression

if Expression

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

- ***cond*** is an expression with **`Bool`** type
- If the ***cond*** is **`True`**, it returns exp_1. Otherwise, it return exp_2.
- exp_1 and exp_2 are expressions, not code blocks (groups of expressions)
  - You can write only one expression.

Pattern Match

- Using pattern matching of values, define functions or use in case expressions

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

- Patterns
  - variable pattern
  - wildcard pattern
  - literal pattern
  - tuple pattern
  - list pattern
  - data constructor pattern

Variable and Wildcard Patterns

- Variable patter
  - matches with any value
  - The variable is bound to the matched value.

```
id :: a -> a
id x = x
```

- Wildcard pattern
  - matches with any value
  - The matched value is not bound to any variable.
  - used as a space holder

```
const :: a -> b -> a
const x _ = x

map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

Literal and Tuple Patterns

- Literal Pattern
  - matches with the given literal
  - available for numerical literal, character literal and string literal

```
expandTab :: Char -> Char
expandTap '\t' = '@'
expandTab c    = c
```

- Tuple Pattern
  - matches with a tuple value
  - Each tuple elements are matched.
  - Tuple elements have any pattern.
  - **(*pat*$_1$, *pat*$_2$, *pat*$_3$, ····)**

```
format :: (Int, String) -> String
format (n, line) = rjust 6 (show n) ++ " " ++ line
```

List and Data Constructor Patterns

- List pattern
  - matches with a list
  - [pat_1, pat_2, pat_3, ····]

```
last []     = error "last []"
last [x]    = x
last (_:xs) = last xs
```

- Data constructor pattern
  - Lists are constructed from the empty list `[]` and `:`

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

@ Pattern and Guard

- @ pattern
  - 'as' pattern
  - ***var*@*pat***
  - matches with ***pat*** and the matched value is bound to ***var***

```
lstrip str@(c:cs) = if isSpace c then lstrip cs else str
```

- Guard
  - Bool expression is checked after the pattern match.
  - pat_1 pat_2 pat_3 .... | *guard*

```
joinPath :: String -> String -> String
joinPath a b | null a              = pathSep : b
             | last a == pathSep = a ++ b
             | otherwise           = a ++ pathSepStr ++ b
```

`case` Expression

```
case exp of
  pat_A | guard_A1 -> exp_A1
        | guard_A2 -> exp_A2
    :          :
  pat_B | guard_B1 -> exp_B1
        | guard_B2 -> exp_B2
    :          :
    :          :
```

- pattern match and check guard with ***exp***
  - The first true match $exp_{??}$ value is selected.

```
case str of
  ""     -> ""
  (c:cs) -> toUpper c : cs
```

- **`of`** is followed by a code block.

```
case str of { "" -> ""; (c:cs) -> toUpper c : cs }
```

Function Definition

```
fun pat_A1 pat_A2 .... | guard_A1 = exp_A1
                       | guard_A2 = exp_A2
                            :          :
fun pat_B1 pat_B2 .... | guard_B1 = exp_B1
                       | guard_B2 = exp_B2
                            :          :
```

- defines a function with pattern match
- Function name and variables are identifiers:
  - starts with a small alphabet letter
  - follows small and upper alphabet letters, numbers, underscore and single quote.
- The following identifiers are reserved and cannot be used:
  - **`case`**, **`class`**, **`data`**, **`default`**, **`deriving`**, **`do`**, **`else`**, **`if`**, **`import`**, **`in`**, **`infix`**, **`infixl`**, **`infixr`**, **`instance`**, **`let`**, **`module`**, **`newtype`**, **`of`**, **`then`**, **`type`**, **`where`**, **`-`**

Binary Operator Definition

```
pat1 op pat2 = exp
```

- defines a binary operator like function definition
  - Any combination of symbols are regarded as a binary operator.
  - Any function can be treated as a binary operator by: `` `fun` ``
  - Any binary operator can be treated as a function by: `(op)`

```
(||) :: Bool -> Bool -> Bool
True  || _ = True
False || x = x
```

Priority	Left Associative	Non Associative	Right Associative
9	`!!`		`..`
8			`^` `^^` `**`
7	`*` `/` `` `div` `` `` `mod` `` `` `rem` `` `` `quot` ``		
6	`+` `-`		
5			`:` `++`
4		`==` `/=` `<` `<=` `>` `>=` `` `elem` `` `` `notElem` ``	
3			`&&`
2			`||`
1	`>>` `>>=`		
0			`$` `$!` `` `seq` ``

`let` Expression

```
let def1
    def2
    def3
     :
     :
in exp
```

- let expression allows to define variables and functions which can be used in ***exp***
  - ***exp*** is evaluated under def_1, def_2, ···· are defined.
  - The definitions cannot be referred from outside of ***exp***.

```
f n = let x = n + 1
          y = n + 2
          z = n + 3
      in x * y * z
```

`where` Clause

```
def0 where def1 def2 def3 ....
```

- def_1, def_2, def_3 can be used in def_0
  - def_1, def_2, def_3 can refer parameters of def_0

```
resolverY2K y = base + y where base = 1900
```

```
expandTab :: Int -> String -> String
expandTab width cs = concatMap translate cs
  where
    translate '\t' = replicate width ' '
    translate c    = [c]
```

Exercise 7-1

- Inputs a year and outputs True if it is a leap year.
  - If a year divisible by 4, it is a leap year,
  - but, if it is divisible by 100, it is not a leap year,
  - but, if it is divisible by 400, it is a leap year.

leap.hs

```
import System.Environment

main = do args <- getArgs
          print $ leap $ read $ head args

leap::Int -> Bool
leap y = if ...
```

```
% ghc leap.hs
...
% ./leap 2020
True
% ./leap 2021
False
%
```

Exercise 7-2

- Inputs a year and a month, outputs the number of days in the month.
  - e.g. 2020/2 has 29 days, and 2020/3 has 31 days.
  - If the year is a leap year, its February has 29 days.
  - **`(xs !! n)`** gives **n**th element of **`xs`**

monthday.hs

```
import System.Environment

main = do args <- getArgs
          print $ monthDay (read $ args !! 0) (read $ args !! 1)

monthDay::Int -> Int -> Int
monthDay ... = ...
```

- Use pattern match.

```
% ghc monthday.hs
...
% ./monthday 2020 2
29
% ./monthday 2020 11
30
%
```

Exercise 7-3

- Inputs a year, a month and a day, calculates the number of days from 1/1/1
  - Let 1/1/1 be the first day, 1.

days.hs

```
import System.Environment

main =
  do args <- getArgs
     print $ days (read $ args !! 0) (read $ args !! 1) (read $ args !! 2)

yearDay year = if leap year then 366 else 365

monthDay year month = ...

days year month day = ...
```

```
% ./days 2020 11 24
737753
%
```

- Using this, we can calculate day of the week.
  - 1/1/1 is Monday.

Exercise 7-4

- Given a birthday and outputs the following anniversary dates:
  - 10 days after the birth
  - 100 days after the birth
  - 1000 days after the birth
  - 10000 days after the birth.

anniversary.hs

```
import System.Environment

main = do args <- getArgs
          let year = read $ args !! 0
          let month = read $ args !! 1
          let day = read $ args !! 2
          putStrLn $ yearStr year month (day + 10)
          putStrLn $ yearStr year month (day + 100)
          putStrLn $ yearStr year month (day + 1000)
          putStrLn $ yearStr year month (day + 10000)

yearStr:: Int -> Int -> Int -> String
yearStr year month day = ...
```

```
% ./anniversary 2001 1 1
2001/1/11
2001/4/11
2003/9/28
2028/5/19
%
```

Exercise 7-5

- Inputs a year and a month, outputs the calendar.

cal.hs

```
import System.Environment

main = do args <- getArgs
          purStr $ cal (read $ args !! 0)
                       (read $ args !! 1)

cal::Int -> Int -> String
```

```
% ./cal 2017 11
Su Mo Tu We Th Fr Sa
            1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30
%
```

- Hint
  - Calculate the first day of the month starts which day of the week. (see exercise 7-3)
  - Calculate the number of days in the month. (see exercise 7-2)
  - Format the calendar nicely.
    - e.g. create the list of days and fold for each week.