Modules

Thumrongsak Kosiyatrakul
tkosiyat@pitt.edu

Modules

- A Haskell program is a collection of **modules**
- A **module** is simply a file that defines a collection of functions, types, and type classes
- A module can have multiple functions and types
- We can specify which functions, types, or constructors will be visible to the outside
    - Compare to classes vs inner classes
    - Compare to public and private
- General modules can be used by multiple programs (code reuse)
- Code is easy to manage when we split to multiple parts

Import Modules

- Additional modules other than standard `Prelude` can be imported
- `Data.List` contains a number of additional functions for working with lists
- To import the `Data.List` module into a `GHCi` session, perform one of these methods:
  1. At the beginning of a Haskell source file:

     ```
     import Data.List
     ```

     and load the file into the `GHCi` as usual
  2. In the `GHCi`:

     ```
     ghci> :m + Data.List
     ```

Import Modules

- To import multiple modules like `Data.List` and `Data.Map`, perform one of these methods:
  1. At the beginning of a Haskell source file:

     ```
     import Data.List
     import Data.Map
     ```

     and load the file into the `GHCi` as usual
  2. In the `GHCi`:

     ```
     ghci> :m + Data.List Data.Map
     ```

- To import **only** `nub` and `sort` functions from `Data.List` module:

  ```
  import Data.List (nub, sort)
  ```

- To import all functions from `Data.List` module **except** the `num` function:

  ```
  import Data.List hiding (nub)
  ```

Import Modules

- Function names of an imported module may be the same as those already imported
- `Data.Map` module contain functions `filter` and `null`

```
ghci> :t filter
filter :: (a -> Bool) -> [a] -> [a]
ghci> :m + Data.Map
ghci> :t filter

<interactive>:1:1: error:...
```

- To solve this problem, we need to use **qualified imports**

```
import qualified Data.Map
```

- To use the `filter` function from `Data.Map` module, simply use `Data.Map.filter`:

```
ghci> :t filter
filter :: (a -> Bool) -> [a] -> [a]
ghci> :t Data.Map.filter
Data.Map.filter :: (a -> Bool) -> Data.Map.Map k a -> Data.Map.Map k a
```

Import Modules

- We can make a qualified import name shorter using the `as` keyword:

```
import qualified Data.Map as M
```

- To use the `filter` function from `Data.Map` module, simply use `Data.Map.filter`:

```
ghci> :t M.filter
M.filter :: (a -> Bool) -> M.Map.Map k a -> M.Map.Map k a
```

Module Functions

- We will look at some useful functions provided by modules
- The `words` functions from `Data.List` converts a string into a list of strings

```
ghci> :t words
words :: String -> [String]
ghci> words "Haskell is cool and fun to play with"
["Haskell","is","cool","and","fun","to","play","with"]
```

- The `group` function from `Data.List` groups identical adjacent elements into sublists:

```
ghci> :t group
group :: Eq a => [a] -> [[a]]
ghci> group [3,3,5,5,5,1,2,2,2,2,4,4,4,4,4,7,6,1,1]
[[3,3],[5,5,5],[1],[2,2,2,2],[4,4,4,4,4],[7],[6],[1,1]]
```

- Note that `[1]` and `[1,1]` are two separated group since they are not adjacent to each other

Module Functions

- The `sort` function from `Data.List` can be used to sort lists:

```
ghci> :t sort
sort :: Ord a => [a] -> [a]
ghci> sort [3,3,5,5,5,1,2,2,2,2,4,4,4,4,4,7,6,1,1]
[1,1,1,2,2,2,2,3,3,4,4,4,4,4,5,5,5,6,7]
```

- Now we can group duplicate elements together into one group

```
ghci> (group . sort) [3,3,5,5,5,1,2,2,2,2,4,4,4,4,4,7,6,1,1]
[[1,1,1],[2,2,2,2],[3,3],[4,4,4,4,4],[5,5,5],[6],[7]]
```

- What if we want to turn the original list into a list of pairs of element and the number of that element on the list like the following:

```
[(1,3),(2,4),(3,2),(4,5),(5,3),(6,1),(7,1)]
```

- Simply `map` the previous result as follows:

```
ghci> (map (\xs -> (head xs, length xs)) . group . sort)
      [3,3,5,5,5,1,2,2,2,2,4,4,4,4,4,7,6,1,1]
[(1,3),(2,4),(3,2),(4,5),(5,3),(6,1),(7,1)]
```

Substring (as List)

- We know that `"put"` is a substring of `"Computer"`
- Since a string in Haskell is simply a list of character, let's call this property **sub-list** (just for this discussion)
- So, `[1,2]` is a sub-list of `[3,5,1,2,4]`
- However, `[3,2]` is not a sublist of `[3,5,1,2,4]`
- How to check that a list is a sub-list of another list?
- The `tails` functions in `Data.List` takes a list and successively applies the `tail` function to that list:

```
ghci> :t tails
tails :: [a] -> [[a]]
ghci> tails "Computer"
["Computer","omputer","mputer","puter","uter","ter","er","r",""]
```

Substring (as List)

- The `isPrefixOf` function from `Data.List` takes two lists and tells us if the second one starts with the first one:

```
ghci> :t isPrefixOf
isPrefixOf :: Eq a => [a] -> [a] -> Bool
ghci> isPrefixOf "put" "puter"
True
ghci> isPrefixOf "put" "Computer"
False
ghci> isPrefixOf [3,5,1] [3,5,1,2,4]
True
ghci> isPrefixOf [5,1] [3,5,1,2,4]
False
```

- If we `map` the `isPrefixOf "put"` function to the `tails "Computer"`, we get the list of boolean:

```
ghci> map (isPrefixOf "put") (tails "Computer")
[False,False,False,True,False,False,False,False,False]
```

Substring (as List)

- Now, we can simply use the `or` function

```
isSubList xs ys = or (map (isPrefixOf xs) (tails ys))
```

- Here is a couple tests

```
ghci> isSubList "put" "Computer"
True
ghci> isSubList "pt" "Computer"
False
ghci> isSubList [5,1,2] [3,5,1,2,4]
True
```

- The `any` function from `Data.List` takes a predicate and a list and tells us if any element from the list satisfies the predicate

```
ghci> :t any
any :: Foldable t => (a -> Bool) -> t a -> Bool
ghci> any (==5) [3,5,1,2,4]
True
ghci> any (>6) [3,5,1,2,4]
False
```

- We can use the `any` function instead of `or` and `map`

```
isSubList' xs ys = any (isPrefixOf xs) (tails ys)
```

`chr` and `ord` Function

- The `ord` function from `Data.Char` converts a given character into its corresponding number:

```
ghci> :t ord
ord :: Char -> Int
ghci> ord 'H'
72
ghci> ord '!'
33
ghci> ord 'k'
107
```

- The `chr` function from `Data.Char` converts a given number into its corresponding character:

```
ghci> :t chr
chr :: Int -> Char
ghci> chr 72
'H'
ghci> chr 33
'!'
ghci> chr 107
'k'
```

Caesar Cipher

- Caesar Cipher encodes a message by shifting every character in the message by a fix magnitude
- For example, if we shift every character in `"Haskell"` by positive 5, we get `"Mfxpjqq"`
- Let's define the function `encode` that takes a number and a message, and turn the message into a new message according to the Caesar cipher
  - Need to turn a character into a number $\leadsto$ `ord`
  - Need to shift the number by `n` $\leadsto$ `(+n)`
  - Need to turn a new number back to a character $\leadsto$ `chr`
  - The function (for each character) is simply `chr . (+n) . ord`
- Now we have

```
encode :: Int -> [Char] -> [Char]
encode n xs = map (chr . (+n) . ord) xs
```

Caesar Cipher

- To decode a Caesar cipher, we simply need to know the amount of shifting
- We can use the `encode` function to decode a message by negating the amount of shift
- Now, we have

```
decode :: Int -> [Char] -> [Char]
decode n xs = encode (negate n) xs
```

- Let's do some tests:

```
ghci> encode 10 "I love Haskell. It is fun."
"S*vy\128o*Rk}uovv8*S~*s}*p\DELx8"
ghci> decode 10 "S*vy\128o*Rk}uovv8*S~*s}*p\DELx8"
"I love Haskell. It is fun."
```

Left Fold Problem

- Consider the following expressions:

```
ghci> foldl (+) 0 (replicate 10000000 1)
10000000
ghci> foldl (+) 0 (replicate 50000000 1)
*** Exception: stack overflow
```

- You may only see `Killed` and exit without seeing stack overflow exception
- Recall the definition of `foldl`

```
foldl f z [] = z
foldl f z (x:xs) = foldl f (f z x) xs
```

- Let's trace and see what is going on:

```
foldl (+) 0 [1,2,3]
foldl (+) (0 + 1) [2,3]
foldl (+) ((0 + 1) + 2) [3]
foldl (+) (((0 + 1) + 2) + 3) []
((0 + 1) + 2) + 3
(1 + 2) + 3
3 + 3
6
```

Left Fold Problem

- Because of the lazy evaluation, Haskell needs to keep the intermediate expression
- The expression keeps getting longer and longer which require more and more memory
- The `foldl'` in `Data.List` is the same as `foldl` but it does not defer the computation

```
foldl' (+) 0 [1,2,3]
foldl' (+) (0 + 1) [2,3]
foldl' (+) 1 [2,3]
foldl' (+) (1 + 2) [3]
foldl' (+) 3 [3]
foldl' (+) (3 + 3) []
foldl' (+) 6 []
6
```

- No stack overflow:

```
ghci> foldl' (+) 0 (replicate 50000000 1)
50000000
```

The `digitToInt` Function

- The `digitToInt` from `Data.Char` takes a digit and returns an `Int`

```
ghci> :t digitToInt
digitToInt :: Char -> Int
ghci> digitToInt '6'
6
ghci> digitToInt 'f'
15
ghci> digitToInt 'C'
12
ghci> digitToInt 'r'
*** Exception: Char.digitToInt: not a digit 'r'
```

- It supports hexadecimal digits as well
- Given an integer, find the sum of all digits...
  - Turn an integer into its string representation $\rightsquigarrow$ `show`
  - For each digit, turn it into its value $\rightsquigarrow$ `digitToInt`
  - Sum all digits $\rightsquigarrow$ `sum`
- The function is `(sum . map digitToInt . show)`

The `find` Function

- The `find` function from `Data.List` takes a predicate and a list, and returns the first element on the given list that satisfies the predicate

```
ghci> :t find
find :: Foldable t => (a -> Bool) -> t a -> Maybe a
ghci> find (<2) [3,5,1,2,4]
Just 1
ghci> find (\x -> div x 37 == 2) [1..]
Just 74
ghci> find (>6) [3,5,1,2,4]
Nothing
```

- Note that the return type is `Maybe a`
- If no element satisfies the predicate, it returns `Nothing`

Exercise

- Find the first natural number where its digits sum up to a given number `n`
  - The list of all natural numbers

    ```
    [1..]
    ```

  - Find the first number x where $f(x)$ is n

    ```
    find (\x -> f x == n) [1..]
    ```

    where `f` turn each number into sum of its digit
  - Recall the function that returns the sum of all digits of a given number:

    ```
    sum . map digitToInt . show
    ```

- Final Function:

  ```
  firstEqual n = find (\x -> (sum . map digitToInt . show) x == n) [1..]
  ```

Key/Value Pair

- We generally use a directionary data structure for key/value mapping
- One way to do this in Haskell is to use a list of pairs of key and value

```
phonebook =
   [("Charley", "782-808-4631"),
    ("Maisy", "607-895-7258"),
    ("Josie", "679-537-2084"),
    ("Lana", "931-674-5053"),
    ("Louisa", "658-851-0163"),
    ("Jane", "554-883-8260"),
    ("Patricia", "916-809-9262"),
    ("Hana", "308-599-3830")]
```

Key to Value

- Define a function that search for a key and return its value should be straightforward

```
findKey key [] = Nothing
findKey key ((k,v):xs) = if key == k
                         then Just v
                         else findKey key xs
```

- Can we do this we a fold?

```
findKey' key xs = foldr
                  (\(k,v) acc -> if k == key then Just v else acc)
                  Nothing
                  xs
```

- A couple tests:

```
ghci> findKey "Maisy" phonebook
Just "607-895-7258"
ghci> findKey' "Jane" phonebook
Just "554-883-8260"
```

Association List with Data.Map

- Data.Map module offers association lists that are much faster with some utility functions
- Note that lots of functions name in Data.Map are the same as in Prelude
- Do not forget to use **qualified** import

```
import qualified Data.Map as Map
```

- The fromList function takes an association list and returns a map

```
ghci> :t Map.fromList
Map.fromList :: Ord k => [(k, a)] -> Map.Map k a
ghci> Map.fromList [("CS0401", 70), ("CS0441", 45), ("CS0445", 50)]
fromList [("CS0401",70),("CS0441",45),("CS0445",50)]
ghci> Map.fromList [("Dog", 4), ("Dog", 6), ("Cat", 2), ("Dog", 1)]
fromList [("Cat",2),("Dog",1)]
```

- The output is just a formList (ignore)
- Note that uplicate keys are discarded
- Items are sorted as well

Association List with `Data.Map`

- Let's use our phonebook with `Map.fromList`

```
import qualified Data.Map as Map

phonebook = Map.fromList
    [("Charley", "782-808-4631"),
     ("Maisy", "607-895-7258"),
     ("Josie", "679-537-2084"),
     ("Lana", "931-674-5053"),
     ("Louisa", "658-851-0163"),
     ("Jane", "554-883-8260"),
     ("Patricia", "916-809-9262"),
     ("Hana", "308-599-3830")]
```

- The `lookup` function from `Data.Map` takes a key and a map, and returns the correspond value (if exists)

```
ghci> :t Map.lookup
Map.lookup :: Ord k => k -> Map.Map k a -> Maybe a
ghci> Map.lookup "Lana" phonebook
Just "931-674-5053"
ghci> Map.lookup "John" phonebook
Nothing
```

Association List with `Data.Map`

- The `insert` function from `Data.Map` takes a key, a value, and a map, and it returns a new map with inserted key/value pair

```
ghci> :t Map.insert
Map.insert :: Ord k => k -> a -> Map.Map k a -> Map.Map k a
ghci> newphonebook = Map.insert "John" "555-525-5525" phonebook
ghci> newphonebook
fromList [("Charley","782-808-4631"),("Hana","308-599-3830"),
("Jane","554-883-8260"),("John","555-525-5525"),
("Josie","679-537-2084"),("Lana","931-674-5053"),
("Louisa","658-851-0163"),("Maisy","607-895-7258"),
("Patricia","916-809-9262")]
ghci> phonebook
fromList [("Charley","782-808-4631"),("Hana","308-599-3830"),
("Jane","554-883-8260"),("Josie","679-537-2084"),
("Lana","931-674-5053"),("Louisa","658-851-0163"),
("Maisy","607-895-7258"),("Patricia","916-809-9262")]
```

- To get the size of a map, use the `size` function:

```
ghci> Map.size phonebook
8
ghci> Map.size newphonebook
9
```

Association List with `Data.Map`

- To map a function to every **value** in a map, use the function `map` from `Data.Map`

```
ghci> :t Map.map
Map.map :: (a -> b) -> Map.Map k a -> Map.Map k b
ghci> Map.map (filter isDigit) phonebook
fromList [("Charley","7828084631"),("Hana","3085993830"),
("Jane","5548838260"),("Josie","6795372084"),("Lana","9316745053"),
("Louisa","6588510163"),("Maisy","6078957258"),
("Patricia","9168099262")]
```

- Note that the − symbols in phone numbers are gone

Association List with Data.Map

- Since a key must be unique, if we want to keep duplicates, we need to tell what to do with values of the same key
- The `fromListWith` function is the same as `fromList` but it will apply a given function to values with duplicate key

```
phonebook = Map.fromListWith concat
   [("Charley", "782-808-4631"),
    ("Maisy", "607-895-7258"),
    ("Josie", "679-537-2084"),
    ("Lana", "931-674-5053"),
    ("Maisy", "607-345-6789"),
    ("Lana", "931-441-1278")]
   where concat xs ys = xs ++ ", " ++ y
```

- Let's check the phonebook

```
ghci> phonebook
fromList [("Charley","782-808-4631"),("Josie","679-537-2084"),
("Lana","931-441-1278, 931-674-5053"),
("Maisy","607-345-6789, 607-895-7258")]
ghci> Map.lookup "Lana" phonebook
Just "931-441-1278, 931-674-5053"
```

User Defined Modules

- Suppose we want to create the `Vector` module with some functions
- Here is a simple example (`Vector.hs`):

```
module Vector
( Vector (Vector3D, Vector4D),
  vvAdd,
  vvDot) where

data Vector = Vector3D Float Float Float
            | Vector4D Float Float Float Float

vvAdd :: Vector -> Vector -> Vector
vvAdd (Vector3D x1 y1 z1) (Vector3D x2 y2 z2) =
    Vector3D (x1 + x2) (y1 + y2) (z1 + z2)
vvAdd (Vector4D x1 y1 z1 w1) (Vector4D x2 y2 z2 w2) =
    Vector4D (x1 + x2) (y1 + y2) (z1 + z2) (w1 + w2)

vvDot :: Vector -> Vector -> Float
vvDot (Vector3D x1 y1 z1) (Vector3D x2 y2 z2) =
    (x1 * x2) + (y1 * y2) + (z1 * z2)
vvDot (Vector4D x1 y1 z1 w1) (Vector4D x2 y2 z2 w2) =
    (x1 * x2) + (y1 * y2) + (z1 * z2) + (w1 * w2)
```

- This file should be in the same directory of modules that want to import it

User Defined Modules

- Note that the module name and the file name must be the same
- The file must start with the `module` keyword
- Items inside parentheses (after the module name) are data types and functions that will be visible to users
- Items inside parentheses (after a type name) are value constrctors of that type that will be visible to users
- Expression of types and functions are define after the `where` keyword
- To export all functions, simply use `..`

```
module Vector (..) where
```

- To use the module, simply import as usual

```
import Vector
```

- A couple tests:

```
ghci> v1 = Vector3D 1 2 3
ghci> v2 = Vector3D 4 5 6
ghci> vvAdd v1 v2
Vector3D 5.0 7.0 9.0
```

Module Hierarchy

- Hierarchical structure of modules is possible
- In our work directory, first create a directory such as `MyModules`
- In side the `MyModules` directory, we have a couple files
- `Module01.hs`:

```
module MyModule.Module01
(..)
:
```

- `Module02.hs`:

```
module MyModule.Module02
(..)
:
```

- To import `Module01` in our work directory:

```
import MyModule.Module01
```