Haskell
Modules

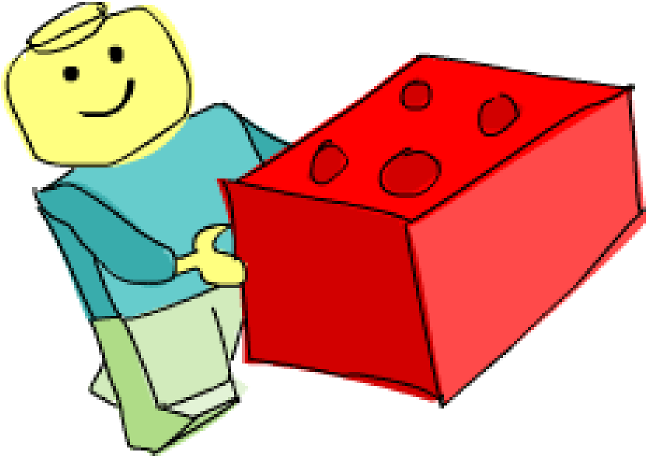
http://igm.univ-mlv.fr/~vialette/?section=teaching

Stéphane Vialette

LIGM, Université Paris-Est Marne-la-Vallée

November 28, 2022

Modules



Modules

- A Haskell module is a collection of related functions, types and typeclasses.
- A Haskell program is a collection of modules where the main module loads up the other modules and then uses the functions defined in them to do something.
- Having code split up into several modules has quite a lot of advantages. If a module is generic enough, the functions it exports can be used in a multitude of different programs. If your own code is separated into self-contained modules which don't rely on each other too much (we also say they are loosely coupled), you can reuse them later on.
- It makes the whole deal of writing code more manageable by having it split into several parts, each of which has some sort of purpose.

Modules

- The Haskell standard library is split into modules, each of them contains functions and types that are somehow related and serve some common purpose.
- There's a module for manipulating lists, a module for concurrent programming, a module for dealing with complex numbers, etc.
- All the functions, types and typeclasses that we've dealt with so far were part of the `Prelude` module, which is imported by default.
- In this chapter, we're going to examine a few useful modules and the functions that they have.

Modules

- The syntax for importing modules in a Haskell script is `import module_name`.
- This must be done before defining any functions, so imports are usually done at the top of the file.
- One script can, of course, import several modules. Just put each import statement into a separate line.

Modules

```haskell
import Data.List

countUniques :: (Eq a) => [a] -> Int
countUniques = length . nub
```

- When you do `import Data.List`, all the functions that `Data.List` exports become available in the global namespace, meaning that you can call them from wherever in the script.
- `nub` is a function defined in `Data.List` that takes a list and weeds out duplicate elements.
- Composing `length` and `nub` by doing `length . nub` produces a function that's the equivalent of `\ xs -> length (nub xs)`.

Importing modules

You can also put the functions of modules into the global namespace when using GHCI. If you're in GHCI and you want to be able to call the functions exported by Data.List, do this:

```
λ: :m + Data.List
```

If we want to load up the names from several modules inside GHCI, we don't have to do :m + several times, we can just load up several modules at once.

```
λ: :m + Data.List Data.Map Data.Set
```

However, if you've loaded a script that already imports a module, you don't need to use `:m +` to get access to it.

Importing modules

If you just need a couple of functions from a module, you can selectively import just those functions.

If we wanted to import only the `nub` and `sort` functions from Data.List, we'd do this:

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

You can also choose to import all of the functions of a module except a few select ones. That's often useful when several modules export functions with the same name and you want to get rid of the offending ones. Say we already have our own function that's called `nub` and we want to import all the functions from `Data.List` except the `nub` function:

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

Importing modules

Another way of dealing with name clashes is to do qualified imports.

The `Data.Map` module, which offers a data structure for looking up values by key, exports a bunch of functions with the same name as `Prelude` functions, like `filter` or `null`.

So when we import `Data.Map` and then call `filter`, Haskell won't know which function to use.

Here's how we solve this:

```
import qualified Data.Map
```

Importing modules

```
import qualified Data.Map
```

This makes it so that if we want to reference `Data.Map`'s `filter` function, we have to do `Data.Map.filter`, whereas just `filter` still refers to the normal filter we all know (and love ☺).

But typing out `Data.Map` in front of every function from that module is kind of tedious. That's why we can rename the qualified import to something shorter:

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

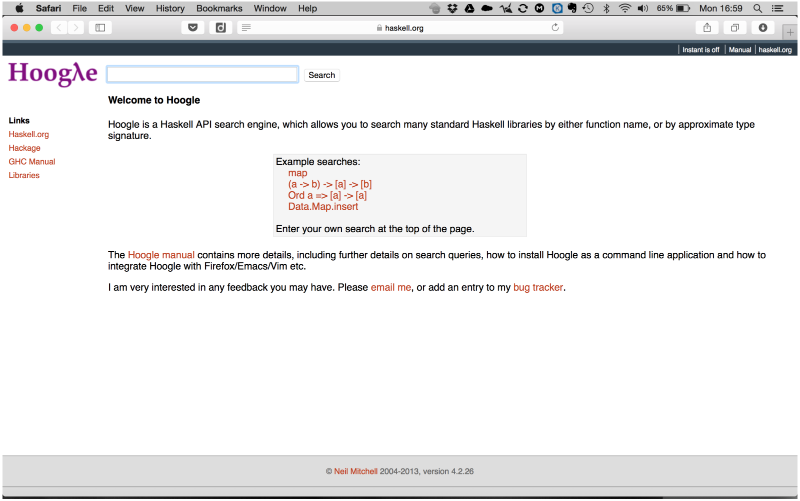
Now, to reference `Data.Map`'s `filter` function, we just use `M.filter`.

Modules

Use `https://downloads.haskell.org/~ghc/latest/docs/html/libraries/` to see which modules are in the standard library.

To search for functions or to find out where they're located, use Hoogle (`https://hoogle.haskell.org/`). It's a really awesome Haskell search engine, you can search by name, module name or even type signature.

Hoogle


Safari File Edit View History Bookmarks Window Help
65% Mon 16:59
haskell.org
Instant is off | Manual | haskell.org
Hoogλe
Search

Welcome to Hoogle

Links
Haskell.org
Hackage
GHC Manual
Libraries

Hoogle is a Haskell API search engine, which allows you to search many standard Haskell libraries by either function name, or by approximate type signature.

Example searches:
map
(a -> b) -> [a] -> [b]
Ord a => [a] -> [a]
Data.Map.insert

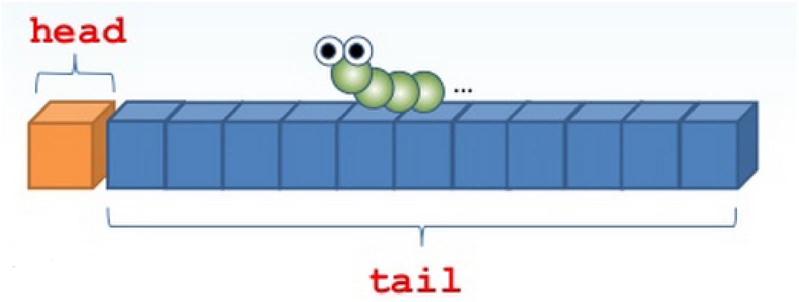
Enter your own search at the top of the page.

The Hoogle manual contains more details, including further details on search queries, how to install Hoogle as a command line application and how to integrate Hoogle with Firefox/Emacs/Vim etc.

I am very interested in any feedback you may have. Please email me, or add an entry to my bug tracker.

© Neil Mitchell 2004-2013, version 4.2.26

Data.List



Data.List

intersperse takes an element and a list and then puts that element in between each pair of elements in the list.

```
λ: :type intersperse
intersperse :: a -> [a] -> [a]
λ: intersperse '.' "MONKEY"
"M.O.N.K.E.Y"
λ: intersperse 0 [1..10]
[1,0,2,0,3,0,4,0,5,0,6,0,7,0,8,0,9,0,10]
```

Data.List

```
intersperse' :: a -> [a] -> [a]
intersperse' _ []         = []
intersperse' _ [x]        = [x]
intersperse' y (x : xs) = x : y : intersperse' y xs
```

Data.List

intercalate takes a list of lists and a list. It then inserts that list in between all those lists and then flattens the result.

```
λ: :type intercalate
intercalate :: [a] -> [[a]] -> [a]
λ: intercalate " " ["hey","there","guys"]
"hey there guys"
λ: intercalate [0,0] [[1,2],[3,4,5],[6,7]]
[1,2,0,0,3,4,5,0,0,6,7]
```

Data.List

transpose transposes a list of lists. If you look at a list of lists as a 2D matrix, the columns become the rows and vice versa.

```
λ: :type transpose
transpose :: [[a]] -> [[a]]
λ: transpose [[1,2,3],[4,5,6],[7,8,9]]
[[1,4,7],[2,5,8],[3,6,9]]
λ: transpose ["hey","there","guys"]
["htg","ehu","yey","rs","e"]
```

Data.List

concat flattens a list of lists into just a list of elements.

```
λ: :type concat
concat :: [[a]] -> [a]
λ: concat ["foo","bar","car"]
"foobarcar"
λ: concat [[3,4,5],[2,3,4],[2,1,1]]
[3,4,5,2,3,4,2,1,1]
```

It will just remove one level of nesting. So if you want to completely flatten `[[[2,3],[3,4,5],[2]],[[2,3],[3,4]]]`, which is a list of lists of lists, you have to concatenate it twice.

Data.List

```haskell
concat' :: [[a]] -> [a]
concat' = foldr (++) []
```

Data.List

Doing concatMap is the same as first mapping a function to a list and then concatenating the list with concat.

```
λ:  :type concatMap
concatMap :: (a -> [b]) -> [a] -> [b]
λ:   concatMap (replicate 4) [1..3]
[1,1,1,1,2,2,2,2,3,3,3,3]
λ:   concatMap (\ x -> [x]) [1..3]
[1,2,3]
λ:   concatMap (\ x -> [[x]]) [1..3]
[[1],[2],[3]]
```

Data.List

```
concatMap' :: (a -> [b]) -> [a] -> [b]
concatMap' f = concat . map f

concatMap'' :: Foldable t => (a -> [b]) -> t a -> [b]
concatMap'' f = concat . foldr (\ x acc -> f x : acc) []
```

Data.List

and takes a list of boolean values and returns True only if all the values in the list are True.

```
λ: :type and
and :: [Bool] -> Bool
λ: and $ map (>4) [5,6,7,8]
True
λ: and $ map (==4) [4,4,4,3,4]
False
λ: and $ map ('a' `elem`) ["to", "ti", "ta"]
False
λ: and $ map ('t' `elem`) ["to", "ti", "ta"]
True
```

Data.List

or is like and, only it returns True if any of the boolean values in a list is True.

```
λ: :type or
or :: [Bool] -> Bool
λ: or $ map (==4) [2,3,4,5,6,1]
True
λ: or $ map (>4) [1,2,3]
False
```

Data.List

```
or' :: [Bool] -> Bool
or' []        = False
or' (x : xs) = x || or' xs

or'' :: [Bool] -> Bool
or'' = foldr (||) False

and' :: [Bool] -> Bool
and' []        = True
and' (x : xs) = x && and' xs

and'' :: [Bool] -> Bool
and'' = foldr (&&) True
```

Data.List

any and all take a predicate and then check if any or all the elements in a list satisfy the predicate, respectively. Usually we use these two functions instead of mapping over a list and then doing and or or.

```
λ: :type any
any :: (a -> Bool) -> [a] -> Bool
λ: any (==4) [2,3,5,6,1,4]
True
λ: any (`elem` ['A'..'Z']) "HEYGUYSwhatsup"
True
λ: :type all
all :: (a -> Bool) -> [a] -> Bool
λ: all (>4) [6,9,10]
True
λ: all (`elem` ['A'..'Z']) "HEYGUYSwhatsup"
False
```

Data.List

```haskell
any' :: (a -> Bool) -> [a] -> Bool
any' _ []         = False
any' f (x : xs) = f x || any' f xs

any'' :: (a -> Bool) -> [a] -> Bool
any'' f = foldr (\ x acc -> f x || acc) False

all' :: (a -> Bool) -> [a] -> Bool
all' _ []         = True
all' f (x : xs) = f x && all' f xs

all'' :: (a -> Bool) -> [a] -> Bool
all'' f = foldr (\ x acc -> f x && acc) True
```

Data.List

iterate takes a function and a starting value. It applies the function to the starting value, then it applies that function to the result, then it applies the function to that result again, etc. It returns all the results in the form of an infinite list.

```
λ: :type iterate
iterate :: (a -> a) -> a -> [a]
λ: take 10 $ iterate (*2) 1
[1,2,4,8,16,32,64,128,256,512]
λ: take 3 $ iterate (++ "haha") "haha"
["haha","hahahaha","hahahahahaha"]
```

Data.List

splitAt takes a number and a list. It then splits the list at that many elements, returning the resulting two lists in a tuple.

```
λ: :type splitAt
splitAt :: Int -> [a] -> ([a], [a])
λ:  splitAt 3 "heyman"
("hey","man")
λ: splitAt 100 "heyman"
("heyman","")
λ: splitAt (-3) "heyman"
("","heyman")
λ: let (a,b) = splitAt 3 "foobar" in b ++ a
"barfoo"
```

Data.List

takeWhile is a really useful little function. It takes elements from a list while the predicate holds and then when an element is encountered that doesn't satisfy the predicate, it's cut off. It turns out this is very useful.

```
λ: :type takeWhile
takeWhile :: (a -> Bool) -> [a] -> [a]
λ: takeWhile (>3) [6,5,4,3,2,1,2,3,4,5,4,3,2,1]
[6,5,4]
λ: takeWhile (/=' ') "This is a sentence"
"This"
λ: sum $ takeWhile (<10000) $ map (^3) [1..]
53361
```

Data.List

```
any' :: (a -> Bool) -> [a] -> Bool
any' _ []          = False
any' f (x : xs) = f x || any' f xs

any'' :: (a -> Bool) -> [a] -> Bool
any'' f = foldr (\ x acc -> f x || acc) False

all' :: (a -> Bool) -> [a] -> Bool
all' _ []          = True
all' f (x : xs) = f x && all' f xs

all'' :: (a -> Bool) -> [a] -> Bool
all'' f = foldr (\ x acc -> f x && acc) True
```

Data.List

dropWhile is similar to takeWhile, only it drops all the elements while the predicate is true. Once predicate equates to False, it returns the rest of the list. An extremely useful (and lovely ☺) function!

```
λ: :type dropWhile
dropWhile :: (a -> Bool) -> [a] -> [a]
λ: dropWhile (/=' ') "This is a sentence"
" is a sentence"
λ: dropWhile (<3) [1,2,2,2,3,4,5,4,3,2,1]
[3,4,5,4,3,2,1]
λ: :{
| let stock = [(9.2,2008,01),(11.4,2008,02),
|                           (7.2,2007,03)]
| :}
λ: head $ dropWhile (\(v,y,n) -> v < 10) stock
(11.4,2008,2)
```

Data.List

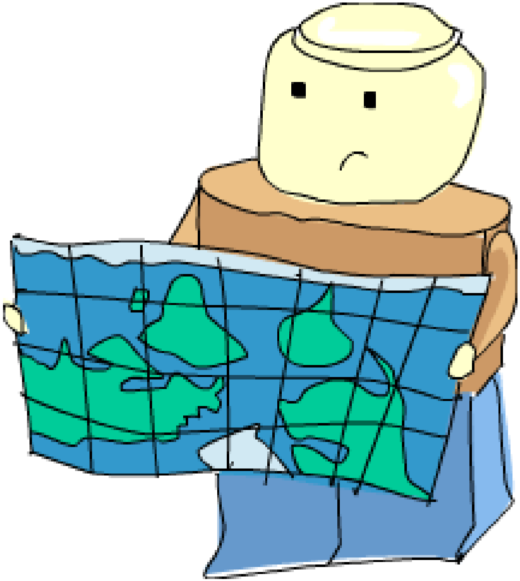
```
dropWhile' :: (a -> Bool) -> [a] -> [a]
dropWhile' _ [] = []
dropWhile' f (x : xs)
  | f x       = dropWhile' f xs
  | otherwise = x : xs
```

Data.List

sort simply sorts a list. The type of the elements in the list has to be part of the Ord typeclass, because if the elements of a list can't be put in some kind of order, then the list can't be sorted.

```
λ: :type sort
sort :: Ord a => [a] -> [a]
λ: sort [8,5,3,2,1,6,4,2]
[1,2,2,3,4,5,6,8]
λ: sort ['a','g','d','b','d','c','f','e']
"abcddefg"
```

`Data.Map`



Data.Map

Association lists (also called dictionaries) are lists that are used to store key-value pairs where ordering doesn't matter.

For instance, we might use an association list to store phone numbers, where phone numbers would be the values and people's names would be the keys. We don't care in which order they're stored, we just want to get the right phone number for the right person.

```haskell
phoneBook =
    [("betty","555-2938")
    ,("bonnie","452-2928")
    ,("patsy","493-2928")
    ,("lucille","205-2928")
    ,("wendy","939-8282")
    ,("penny","853-2492")
    ]
```

Data.Map

Let's make a function that looks up some value given a key.

```haskell
findKey :: (Eq k) => k -> [(k,v)] -> v
findKey k = snd . head . filter (\ (k',v) -> k'==k)
```

Here, if a key isn't in the association list, we'll end up trying to get the head of an empty list, which throws a runtime error. However, we should avoid making our programs so easy to crash, so let's use the Maybe data type. If we don't find the key, we'll return a `Nothing`. If we find it, we'll return `Just` something, where something is the value corresponding to that key.

```haskell
findKey :: (Eq k) => k -> [(k,v)] -> Maybe v
findKey key [] = Nothing
findKey k ((k',v):xs) = if k == k'
                        then Just v
                        else findKey k xs
```

Data.Map

```haskell
findKey :: (Eq k) => k -> [(k,v)] -> Maybe v
findKey k = foldr (\ (k',v) acc -> if k==k' then Just v else acc) Nothing
```

It's usually better to use folds for this standard list recursion pattern instead of explicitly writing the recursion because they're easier to read and identify. Everyone knows it's a fold when they see the foldr call, but it takes some more thinking to read explicit recursion.

Data.Map

```
λ: findKey "penny" phoneBook
Just "853-2492"
λ: findKey "betty" phoneBook
Just "555-2938"
λ: findKey "wilma" phoneBook
Nothing
```

We just implemented the `lookup` function from `Data.List`.

If we want to find the corresponding value to a key, we have to traverse all the elements of the list until we find it.

The `Data.Map` module offers association lists that are much faster (because they're internally implemented with trees) and also it provides a lot of utility functions.

Data.Map

Because Data.Map exports functions that clash with the Prelude and Data.List ones, we'll do a qualified import.

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

Data.Map

The fromList function takes an association list (in the form of a list) and returns a map with the same associations.

```
λ: :type M.fromList
M.fromList :: Ord k => [(k, a)] -> M.Map k a
λ: M.fromList [('a', 1),('b', 2),('c',3),('d',4)]
fromList [('a',1),('b',2),('c',3),('d',4)]
λ: M.fromList [(i,i^2) | i <- [1..5]]
fromList [(1,1),(2,4),(3,9),(4,16),(5,25)]
λ: M.fromList [(1,1),(1,1),(2,2)]
fromList [(1,1),(2,2)]
```

Data.Map

empty represents an empty map. It takes no arguments, it just returns an empty map.

```
λ: :type M.empty
M.empty :: M.Map k a
λ: M.empty
fromList []
```

Data.Map

insert

insert takes a key, a value and a map and returns a new map that's just like the old one, only with the key and value inserted.

```
λ: :type M.insert
M.insert :: Ord k => k -> a -> M.Map k a -> M.Map k a
λ: M.insert 'a' 1 M.empty
fromList [('a',1)]
λ: M.insert 'a' 1 $ M.insert 'b' 2 M.empty
fromList [('a',1),('b',2)]
λ: let m = M.empty
λ: M.insert 'a' 1 $ M.insert 'b' 2 m
fromList [('a',1),('b',2)]
λ: M.insert 'a' 2 $ M.insert 'a' 1 M.empty
fromList [('a',2)]
```

Data.Map

insert

We can implement our own fromList by using the empty map, insert and a fold.

```
fromList' :: (Ord k) => [(k,v)] -> M.Map k v
fromList' = foldr (\ (k,v) acc -> M.insert k v acc) M.empty
```

It's a pretty straightforward fold. We start of with an empty map and we fold it up from the right, inserting the key value pairs into the accumulator as we go along.

Data.Map

null

null checks if a map is empty.

```
λ: :type M.null
M.null :: M.Map k a -> Bool
λ: M.null M.empty
True
λ: M.null $ M.insert "k" "v" M.empty
False
```

Data.Map

size

size reports the size of a map.

```
λ: :type M.size
M.size :: M.Map k a -> Int
λ: M.size M.empty
0
λ: M.size $ M.insert "k" "v" M.empty
1
```

Data.Map

singleton

singleton takes a key and a value and creates a map that has exactly one mapping.

```
λ: :type M.singleton
M.singleton :: k -> a -> M.Map k a
λ: M.singleton 3 9
fromList [(3,9)]
λ: M.insert 5 9 $ M.singleton 3 9
fromList [(3,9),(5,9)]
```

Data.Map

lookup

lookup works like the Data.List lookup, only it operates on maps. It returns Just something if it finds something for the key and Nothing if it doesn't.

```
λ: :type M.lookup
M.lookup :: Ord k => k -> M.Map k a -> Maybe a
λ: let m = M.insert 5 9 $ M.singleton 3 9
λ: M.lookup 5 m
Just 9
λ: M.lookup 6 m
Nothing
```

Data.Map

member

member is a predicate takes a key and a map and reports whether the key is in the map or not.

```
λ: :type M.member
M.member :: Ord k => k -> M.Map k a -> Bool
λ: let m = M.insert 5 9 $ M.singleton 3 9
λ: M.member 5 m
True
λ: M.member 6 m
False
```

Data.Map

map and filter

map and filter work much like their list equivalents.

```
λ: let m = M.fromList [(1,1),(2,4),(3,9)]
λ: :type M.map
M.map :: (a -> b) -> M.Map k a -> M.Map k b
λ: M.map (*100) m
fromList [(1,100),(2,400),(3,900)]
λ: :type M.filter
M.filter :: (a -> Bool) -> M.Map k a -> M.Map k a
λ: M.filter (>2) m
fromList [(2,4),(3,9)]
```

Data.Map

toList

toList is the inverse of fromList.

```
λ: :type M.toList
M.toList :: M.Map k a -> [(k, a)]
λ: M.toList . M.insert 9 2 $ M.singleton 4 3
[(4,3),(9,2)]
λ: M.toList M.empty
[]
λ: :type M.assocs
M.assocs :: M.Map k a -> [(k, a)]
λ: M.assocs . M.insert 9 2 $ M.singleton 4 3
[(4,3),(9,2)]
```

Data.Map

keys and elems

keys and elems return lists of keys and values respectively. keys is the equivalent of map fst . M.toList and elems is the equivalent of map snd . M.toList.

```
λ: :type M.keys
M.keys :: M.Map k a -> [k]
λ: :type M.elems
M.elems :: M.Map k a -> [a]
λ: M.keys $ M.insert 9 2 $ M.singleton 4 3
[4,9]
λ: M.elems $ M.insert 9 2 $ M.singleton 4 3
[3,2]
```

Data.Set

The Data.Set module offers us, well, sets.

Sets are kind of like a cross between lists and maps. All the elements in a set are unique. And because they're internally implemented with trees (much like maps in Data.Map), they're ordered.

Checking for membership, inserting, deleting, etc. is much faster than doing the same thing with lists.

The most common operation when dealing with sets are inserting into a set, checking for membership and converting a set to a list.

Because the names in Data.Set clash with a lot of Prelude and Data.List names, we do a qualified import (*e.g.*, `import qualified Data.Set as S`).

Data.Set

fromList

The fromList function works much like you would expect. It takes a list and converts it into a set.

```
λ: :type S.fromList
S.fromList :: Ord a => [a] -> S.Set a
λ: S.fromList []
fromList []
λ: S.fromList "i'm a poor lonesome cowboy"
fromList " 'abceilmnoprswy"
```

Data.Set

intersection

Use intersection function to see which elements sets both share.

```
λ: let s1 = S.fromList "To be or not to Be"
λ: let s2 = S.fromList "That is the question"
λ: :type S.intersection
S.intersection :: Ord a => S.Set a -> S.Set a -> S.Set a
λ: S.intersection s1 s2
fromList " Tenot"
```

Data.Set

difference

We can use the difference function to see which letters are in the first set but aren't in the second one and vice versa.

```
λ: let s1 = S.fromList "To be or not to Be"
λ: let s2 = S.fromList "That is the question"
λ: :type S.difference
S.difference :: Ord a => S.Set a -> S.Set a -> S.Set a
λ: S.difference s1 s2
fromList "Bbr"
λ: S.difference s2 s1
fromList "ahiqsu"
```

Data.Set

union

we can see all the unique letters used in both sentences by using `union`.

```
λ: let s1 = S.fromList "To be or not to Be"
λ: let s2 = S.fromList "That is the question"
λ: :type S.union
S.union :: Ord a => S.Set a -> S.Set a -> S.Set a
λ: S.union s1 s2
fromList " BTabehinoqrstu"
```

Data.Set

null, size, ...

The null, size, member, empty, singleton, insert and delete functions all work like you'd expect them to.

```
λ: :type S.null
S.null :: S.Set a -> Bool
λ: :type S.empty
S.empty :: S.Set a
λ: S.null S.empty
True
```

Data.Set

null, size, . . .

The null, size, member, empty, singleton, insert and delete functions all work like you'd expect them to.

```
λ: S.null $ S.fromList [3,4,5,5,4,3]
False
λ: :type S.size
S.size :: S.Set a -> Int
λ: S.size $ S.fromList [3,4,5,3,4,5]
3
```

Data.Set

null, size, ...

The null, size, member, empty, singleton, insert and delete functions all work like you'd expect them to.

```
λ: :type S.singleton
S.singleton :: a -> S.Set a
λ: S.singleton 9
fromList [9]
λ: S.null $ S.singleton 9
False
λ: S.size $ S.singleton 9
1
```

Data.Set

null, size, ...

The null, size, member, empty, singleton, insert and delete functions all work like you'd expect them to.

```
λ: :type S.insert
S.insert :: Ord a => a -> S.Set a -> S.Set a
λ: S.insert 4 $ S.fromList [9,3,8,1]
fromList [1,3,4,8,9]
λ: S.insert 8 $ S.fromList [5..10]
fromList [5,6,7,8,9,10]
λ: S.insert 8 $ S.fromList [5..10]
```

Data.Set

null, size, . . .

The null, size, member, empty, singleton, insert and delete functions all work like you'd expect them to.

```
λ: :type S.delete
S.delete :: Ord a => a -> S.Set a -> S.Set a
λ: S.delete 4 $ S.fromList [3,4,5,4,3,4,5]
fromList [3,5]
λ: S.delete 1 $ S.fromList [3,4,5,4,3,4,5]
fromList [3,4,5]
```

Data.Set

isSubsetOf and isProperSubsetOf

We can also check for subsets or proper subset. Set *A* is a **subset** of set *B* if *B* contains all the elements that *A* does. Set *A* is a **proper subset** of set *B* if *B* contains all the elements that *A* does but has more elements.

```
λ: :type S.isSubsetOf
S.isSubsetOf :: Ord a => S.Set a -> S.Set a -> Bool
λ: :type S.isProperSubsetOf
S.isProperSubsetOf
  :: Ord a => S.Set a -> S.Set a -> Bool
λ: S.fromList [2,3,4] `S.isSubsetOf` S.fromList [1,2,3,4,5]
True
λ: S.fromList [1,2,3,4,5] `S.isSubsetOf` S.fromList [1,2,3,4,5]
True
λ: S.fromList [1,2,3,4,5] `S.isProperSubsetOf` S.fromList [1,2,
False
λ: S.fromList [2,3,4,8] `S.isSubsetOf` S.fromList [1,2,3,4,5]
False
```

Data.Set

map and filter

We can also map over sets and filter them.

```
λ: :type S.map
S.map :: Ord b => (a -> b) -> S.Set a -> S.Set b
λ: S.map (+1) $ S.fromList [3,4,5,6,7,2,3,4]
fromList [3,4,5,6,7,8]
λ: :type S.filter
S.filter :: (a -> Bool) -> S.Set a -> S.Set a
λ: S.filter odd $ S.fromList [3,4,5,6,7,2,3,4]
fromList [3,5,7]
```

Data.Set

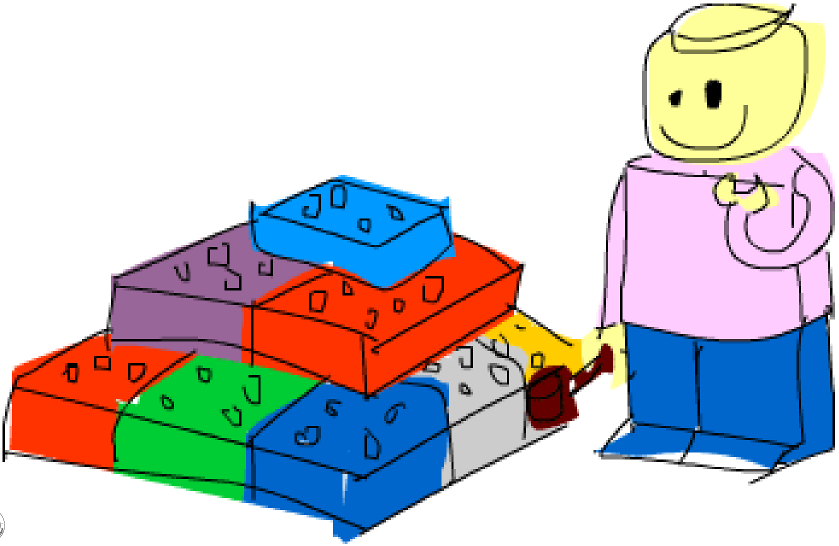
Sets are often used to weed a list of duplicates from a list by first making it into a set with fromList and then converting it back to a list with toList.

The Data.List function nub already does that, but weeding out duplicates for large lists is much faster if you cram them into a set and then convert them back to a list than using nub.

But using nub only requires the type of the list's elements to be part of the Eq typeclass, whereas if you want to cram elements into a set, the type of the list has to be in Ord.

```
λ: let setNub xs = S.toList $ S.fromList xs
λ: setNub "HEY WHATS CRACKALACKIN"
" ACEHIKLNRSTWY"
λ: import Data.List as List
λ: List.nub "HEY WHATS CRACKALACKIN"
"HEY WATSCRKLIN"
```

Making you own module



Making you own module

At the beginning of a module, we specify the module name.

If we have a file called `Geometry.hs`, then we should name our module `Geometry`. Then, we specify the functions that it exports and after t hat, we can start writing the functions.

```
module Geometry
( sphereVolume
, sphereArea
, cubeVolume
, cubeArea
, cuboidArea
, cuboidVolume
) where
...
```

Making you own module

```haskell
sphereVolume :: Float -> Float
sphereVolume radius = (4.0 / 3.0) * pi * (radius ^ 3)

sphereArea :: Float -> Float
sphereArea radius = 4 * pi * (radius ^ 2)

cubeVolume :: Float -> Float
cubeVolume side = cuboidVolume side side side

cubeArea :: Float -> Float
cubeArea side = cuboidArea side side side

cuboidVolume :: Float -> Float -> Float -> Float
cuboidVolume a b c = rectangleArea a b * c

cuboidArea :: Float -> Float -> Float -> Float
cuboidArea a b c = rectangleArea a b * 2 + rectangleArea a c * 2 +
                   rectangleArea c b * 2

-- not exported
rectangleArea :: Float -> Float -> Float
rectangleArea a b = a * b
```

Making you own module

When making a module, we usually export only those functions that act as a sort of interface to our module so that the implementation is hidden.

If someone is using our `Geometry` module, they don't have to concern themselves with functions that we don't export.

We can decide to change those functions completely or delete them in a newer version (we could delete `rectangleArea` and just use `*` instead) and no one will mind because we weren't exporting them in the first place.

Making you own module

To use our module:

```
import Geometry
```

or

```
import qualified Geometry
```

or

```
import qualified Geometry as G
```

`Geometry.hs` has to be in the same folder that the program that's importing it is in, though.

Making you own module

Modules can also be given a hierarchical structures.

Each module can have a number of sub-modules and they can have sub-modules of their own.

Let's section these functions off so that `Geometry` is a module that has three sub-modules, one for each type of object.

First, we'll make a folder called `Geometry`. Mind the capital `G`. In it, we'll place three files: `Sphere.hs`, `Cuboid.hs`, and `Cube.hs`.

Making you own module

Sphere.hs

```haskell
module Geometry.Sphere
( volume
, area
) where

volume :: Float -> Float
volume radius = (4.0 / 3.0) * pi * (radius ^ 3)

area :: Float -> Float
area radius = 4 * pi * (radius ^ 2)
```

Making you own module

Cuboid.hs

```
module Geometry.Cuboid
( volume
, area
) where

volume :: Float -> Float -> Float -> Float
volume a b c = rectangleArea a b * c

area :: Float -> Float -> Float -> Float
area a b c = rectangleArea a b * 2 + rectangleArea a c * 2
             rectangleArea c b * 2

rectangleArea :: Float -> Float -> Float
rectangleArea a b = a * b
```

Making you own module

Cube.hs

```haskell
module Geometry.Cube
( volume
, area
) where

import qualified Geometry.Cuboid as Cuboid

volume :: Float -> Float
volume side = Cuboid.volume side side side

area :: Float -> Float
area side = Cuboid.area side side side
```

Making you own module

So now if we're in a file that's on the same level as the `Geometry` folder, we can do, say:

```
import Geometry.Sphere
```

If we want to juggle two or more of these modules, we have to do qualified imports because they export functions with the same names. So we just do something like:

```
import qualified Geometry.Sphere as Sphere
import qualified Geometry.Cuboid as Cuboid
import qualified Geometry.Cube as Cube
```