CSCE 314
Programming Languages

Haskell: The Module System

Dr. Hyunyoung Lee

Reference: https://www.haskell.org/tutorial/modules.html

Modules

- A Haskell program consists of a collection of modules. The purposes of using a module are:
  1. To control namespaces.
  2. To create abstract data types.

- A module contains various declarations: First, import declarations, and then, data and type declarations, class and instance declarations, type signatures, function definitions, and so on (in any order)

- Module names must begin with an uppercase letter

- One module per file

Example of a Module

export list

```
module Tree ( Tree(Leaf,Branch), fringe ) where
data Tree a = Leaf a | Branch (Tree a) (Tree a)
fringe :: Tree a -> [a]
fringe (Leaf x) = [x]
fringe (Branch left right) = fringe left ++ fringe right
```

- A module declaration begins with the keyword `module`
- The module name may be the same as that of the type
- Same indentation rules as with other declarations apply
- The type name and its constructors need be grouped together, as in `Tree(Leaf,Branch)`; short-hand possible, `Tree(..)`
- Now, the `Tree` module may be imported:

```
module Main (main) where
import Tree ( Tree(Leaf,Branch), fringe )
main = print (fringe (Branch (Leaf 1) (Leaf 2)))
```

import list: omitting it will cause all *entities* exported from Tree to be imported

Qualified Names

```
module Fringe(fringe) where
import Tree( Tree(..) )
fringe :: Tree a -> [a] -- A different definition of fringe
fringe (Leaf x) = [x]
fringe (Branch x y) = fringe x

module Main (main) where
import Tree ( Tree(Leaf,Branch), fringe )
import qualified Fringe ( fringe )

main = do print (fringe (Branch (Leaf 1) (Leaf 2)))
          print (Fringe.fringe (Branch (Leaf 1) (Leaf 2)))
```

shortcut for Tree(Leaf,Branch)

imported names are prefixed by the name of the module imported

- Qualifiers are used to resolve conflicts between different entities with the same name

More Features

- Entities can be hidden in the import declaration. For example, the following explicit import of the Prelude:

```
import Prelude hiding (length, sum)
```

  will not import `length` and `sum` from the Standard Prelude.

- Entities can be renamed with `as`. Used to shorten long names:

```
import AnExtremelyLongModuleName as A
myFun n = A.foo n
```

- or to easily adapt to a change in module name without changing all qualifiers (the following is possible if there are no name conflicts):

```
import Module1 as M
import Module2 as M
```

Abstract Data Types – Tree (1)

Modules are Haskell's mechanism to build abstract data types (ADTs). For example, an ADT for the Tree type might include the following operations (interfaces):

```
data Tree a  -- just the type name
leaf       :: a -> Tree a  -- construct a leaf
branch     :: a -> Tree a -> Tree a -> Tree a  -- construct a branch
cell       :: Tree a -> a  -- return a value of the tree
left, right  :: Tree a -> Tree a  -- return left or right subtree
isLeaf       :: Tree a -> Bool    -- check is a leaf
```

A module supporting this is:

```
module TreeADT (Tree, leaf, branch, cell, left, right, isLeaf) where
data Tree a = Leaf a | Branch a (Tree a) (Tree a)
leaf        = Leaf
branch      = Branch
cell  (Leaf a)        = a
cell  (Branch a _ _) = a
left  (Branch _ l _) = l
right (Branch _ _ r) = r
isLeaf  (Leaf _)      = True
isLeaf  _             = False
```

Leaf and Branch are not exported – information hiding (at a later time the representation type could be changed without affecting users of the type)

Abstract Data Types – Tree (2)

An ADT for the Tree type:

```
data Tree a  -- just the type name
leaf       :: a -> Tree a  -- construct a leaf
branch     :: a -> Tree a -> Tree a -> Tree a  -- construct a branch
cell       :: Tree a -> a  -- return a value of the tree
left, right  :: Tree a -> Tree a  -- return left or right subtree
isLeaf       :: Tree a -> Bool    -- check is a leaf
```

Another module supporting this is:

```
module TreeADT (Tree, leaf, branch, cell, left, right, isLeaf) where
data Tree a = Tnil | Node a (Tree a) (Tree a)
leaf = \x -> (Node x Tnil Tnil)
branch = Node
cell  (Node a Tnil Tnil)   = a
cell  (Node _ l Tnil)      = cell l
cell  (Node _ Tnil r)      = cell r
left  (Node _ l _) = l
right (Node _ _ r) = r
isLeaf  (Node _ Tnil Tnil) = True
isLeaf  _                  = False
```

Another Example ADT - Stack

```
module Stack ( StkType, push, pop, top, empty ) where

data StkType a  = EmptyStk | Stk a (StkType a)
push x s        = Stk x s
pop (Stk _ s)   = s
top (Stk x _)   = x
empty           = EmptyStk
```

```
module Stack ( StkType, push, pop, top, empty ) where

newtype StkType a  = Stk [a]
push x (Stk xs)    = Stk (x:xs)
pop (Stk (_:xs))   = Stk xs
top (Stk (x:_))    = x
empty              = Stk []
```

```
module Main where
import Stack
myStk = push 3 . push 4 . push 2 $ empty
```