11 Modules

A Haskell program consists of a collection of *modules*. A module in Haskell serves the dual purpose of controlling name-spaces and creating abstract data types.

The top level of a module contains any of the various declarations we have discussed: fixity declarations, data and type declarations, class and instance declarations, type signatures, function definitions, and pattern bindings. Except for the fact that import declarations (to be described shortly) must appear first, the declarations may appear in any order (the top-level scope is mutually recursive).

Haskell's module design is relatively conservative: the name-space of modules is completely flat, and modules are in no way "first-class." Module names are alphanumeric and must begin with an uppercase letter. There is no formal connection between a Haskell module and the file system that would (typically) support it. In particular, there is no connection between module names and file names, and more than one module could conceivably reside in a single file (one module may even span several files). Of course, a particular implementation will most likely adopt conventions that make the connection between modules and files more stringent.

Technically speaking, a module is really just one big declaration which begins with the keyword `module`; here's an example for a module whose name is `Tree`:

```
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
```

The type `Tree` and the function `fringe` should be familiar; they were given as examples in Section 2.2.1. [Because of the `where` keyword, layout is active at the top level of a module, and thus the declarations must all line up in the same column (typically the first). Also note that the module name is the same as that of the type; this is allowed.]

This module explicitly *exports* `Tree`, `Leaf`, `Branch`, and `fringe`. If the export list following the `module` keyword is omitted, *all* of the names bound at the top level of the module would be exported. (In the above example everything is explicitly exported, so the effect would be the same.) Note that the name of a type and its constructors have be grouped together, as in `Tree(Leaf,Branch)`. As short-hand, we could also write `Tree(..)`. Exporting a subset of the constructors is also possible. The names in an export list need not be local to the exporting module; any name in scope may be listed in an export list.

The `Tree` module may now be *imported* into some other module:

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

main = print (fringe (Branch (Leaf 1) (Leaf 2)))
```

The various items being imported into and exported out of a module are called *entities*. Note the explicit import list in the import declaration; omitting it would cause all entities exported from `Tree` to be imported.

11.1 Qualified Names

There is an obvious problem with importing names directly into the namespace of module. What if two imported modules contain different entities with the same name? Haskell solves this problem using *qualified names*. An import declaration may use the `qualified` keyword to cause the imported names to be prefixed by the name of the module imported. These prefixes are followed by the `.' character without intervening whitespace. [Qualifiers are part of the lexical syntax. Thus, `A.x` and `A . x` are quite different: the first is a qualified name and the second a use of the infix `.' function.] For example, using the `Tree` module introduced above:

```
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 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)))
```

Some Haskell programmers prefer to use qualifiers for all imported entities, making the source of each name explicit with every use. Others prefer short names and only use qualifiers when absolutely necessary.

Qualifiers are used to resolve conflicts between different entities which have the same name. But what if the same entity is imported from more than one module? Fortunately, such name clashes are allowed: an entity can be imported by various routes without conflict. The compiler knows whether entities from different modules are actually the same.

11.2 Abstract Data Types

Aside from controlling namespaces, modules provide the only way to build abstract data types (ADTs) in Haskell. For example, the characteristic feature of an ADT is that the *representation type* is *hidden*; all operations on the ADT are done at an abstract level which does not depend on the representation. For example, although the `Tree` type is simple enough that we might not normally make it abstract, a suitable ADT for it might include the following operations:

```
data Tree a            -- just the type name
leaf                   :: a -> Tree a
branch                 :: Tree a -> Tree a -> Tree a
cell                   :: Tree a -> a
left, right            :: Tree a -> Tree a
isLeaf                 :: Tree a -> Bool
```

A module supporting this is:

```
module TreeADT (Tree, leaf, branch, cell,
                left, right, isLeaf) where

data Tree a            = Leaf a | Branch (Tree a) (Tree a)

leaf                   = Leaf
branch                 = Branch
cell  (Leaf a)         = a
left  (Branch l r)     = l
right (Branch l r)     = r
isLeaf   (Leaf _)      = True
isLeaf   _             = False
```

Note in the export list that the type name `Tree` appears alone (i.e. without its constructors). Thus `Leaf` and `Branch` are not exported, and the only way to build or take apart trees outside of the module is by using the various (abstract) operations. Of course, the advantage of this information hiding is that at a later time we could *change* the representation type without affecting users of the type.

11.3 More Features

Here is a brief overview of some other aspects of the module system. See the report for more details.

- An `import` declaration may selectively hide entities using a `hiding` clause in the import declaration. This is useful for explicitly excluding names that are used for other purposes without having to use qualifiers for other imported names from the module.
- An `import` may contain an `as` clause to specify a different qualifier than the name of the importing module. This can be used to shorten qualifiers from modules with long names or to easily adapt to a change in module name without changing all qualifiers.

- Programs implicitly import the `Prelude` module. An explicit import of the Prelude overrides the implicit import of all Prelude names. Thus,

  ```
  import Prelude hiding length
  ```

  will not import `length` from the Standard Prelude, allowing the name `length` to be defined differently.

- Instance declarations are not explicitly named in import or export lists. Every module exports all of its instance declarations and every import brings all instance declarations into scope.
- Class methods may be named either in the manner of data constructors, in parentheses following the class name, or as ordinary variables.

Although Haskell's module system is relatively conservative, there are many rules concerning the import and export of values. Most of these are obvious---for instance, it is illegal to import two different entities having the same name into the same scope. Other rules are not so obvious---for example, for a given type and class, there cannot be more than one `instance` declaration for that combination of type and class anywhere in the program. The reader should read the Report for details (§5).
