Haskell/do notation

< Haskell

Using `do` blocks as an alternative monad syntax was first introduced way back in the Simple input and output chapter. There, we used `do` to sequence input/output operations, but we hadn't introduced monads yet. Now, we can see that `IO` is yet another monad.

Since the following examples all involve `IO`, we will refer to the computations/monadic values as *actions* (as we did in the earlier parts of the book). Of course, `do` works with any monad; there is nothing specific about `IO` in how it works.

Translating the *then* operator

The `(>>)` (*then*) operator works almost identically in `do` notation and in unsugared code. For example, suppose we have a chain of actions like the following one:

```
putStr "Hello" >>
putStr " " >>
putStr "world!" >>
putStr "\n"
```

We can rewrite that in `do` notation as follows:

```
do { putStr "Hello"
   ; putStr " "
   ; putStr "world!"
   ; putStr "\n" }
```

(using the optional braces and semicolons explicitly, for clarity). This sequence of instructions nearly matches that in any imperative language. In Haskell, we can chain any actions as long as all of them are in the same monad. In the context of the `IO` monad, the actions include writing to a file, opening a network connection, or asking the user for an input.

Here's the step-by-step translation of `do` notation to unsugared Haskell code:

```
do { action1           -- by monad laws equivalent to:  do {
action1
```

```
   ; action2         --                               ; do {
action2
   ; action3 }       --                                   ;
action3 } }
```

becomes

```
action1 >>
do { action2
   ; action3 }
```

and so on, until the `do` block is empty.

Translating the *bind* operator

The *bind* operator `(>>=)` is a bit more difficult to translate to and from the `do` notation. `(>>=)` passes a value, namely the result of an action or function, downstream in the binding sequence. `do` notation assigns a variable name to the passed value using the `<-`.

```
do { x1 <- action1
   ; x2 <- action2
   ; mk_action3 x1 x2 }
```

The curly braces and the semicolons are optional if every line of code is indented to line up equally (NB: beware the mixing of tabs and spaces in that case; with the explicit curly braces and semicolons indentation plays no part and there's no danger).

`x1` and `x2` are the results of `action1` and `action2`. If, for instance, `action1` is an `IO Integer` then `x1` will be bound to an `Integer` value. The two bound values in this example are passed as arguments to `mk_action3`, which creates a third action. The `do` block is broadly equivalent to the following vanilla Haskell snippet:

```
action1 >>= (\ x1 -> action2 >>= (\ x2 -> mk_action3 x1 x2 ))
```

The second argument of the first (leftmost) bind operator (`>>=`) is a function (lambda expression) specifying what to do with the result of the action passed as the bind's first argument. Thus, chains of lambdas pass the results downstream. The parentheses could be omitted, because a lambda expression extends as far as possible. `x1` is still in scope at the point we call the final action maker `mk_action3`. We can rewrite the chain of lambdas more legibly by using separate lines and indentation:

```
action1
  >>=
    (\ x1 -> action2
       >>=
         (\ x2 -> mk_action3 x1 x2 ))
```

That shows the scope of each lambda function clearly. To group things more like the `do` notation, we could show it like this:

```
action1 >>= (\ x1 ->
  action2 >>= (\ x2 ->
    mk_action3 x1 x2 ))
```

These presentation differences are only a matter of assisting readability.[1]

The *fail* method

Above, we said the snippet with lambdas was "broadly equivalent" to the `do` block. The translation is not exact because the `do` notation adds special handling of pattern match failures. When placed at the left of either `<-` or `->`, `x1` and `x2` are patterns being matched. Therefore, if `action1` returned a `Maybe Integer` we could write a `do` block like this...

```
do { Just x1 <- action1
   ; x2      <- action2
   ; mk_action3 x1 x2 }
```

...and `x1` be an `Integer`. In such a case, what happens if `action1` returns `Nothing`? Ordinarily, the program would crash with an non-exhaustive patterns error, just like the one we get when calling `head` on an empty list. With `do` notation, however, failures are handled with the `fail` method for the relevant monad. The `do` block above translates to:

```
action1 >>= f
```

```
where f (Just x1) = do { x2 <- action2
                         ; mk_action3 x1 x2 }
      f _         = fail "..." -- A compiler-generated message.
```

What `fail` actually does depends on the monad instance. Though it will often rethrow the pattern matching error, monads that incorporate some sort of error handling may deal with the failure in their own specific ways. For instance, `Maybe` has `fail _ = Nothing`; analogously, for the list monad `fail _ = []`.[2]

The `fail` method is an artifact of `do` notation. Rather than calling `fail` directly, you should rely on automatic handling of pattern match failures whenever you are sure that `fail` will do something sensible for the monad you are using.

Example: user-interactive program

Note

We are going to interact with the user, so we will use `putStr` and `getLine` alternately. To avoid unexpected results in the output, we must disable output buffering when importing `System.IO`. To do this, put `hSetBuffering stdout NoBuffering` at the top of your `do` block. To handle this otherwise, you would explicitly flush the output buffer before each interaction with the user (namely a `getLine`) using `hFlush stdout`. If you are testing this code with ghci, you don't have such problems.

Consider this simple program that asks the user for their first and last names:

```
nameDo :: IO ()
nameDo = do putStr "What is your first name? "
            first <- getLine
            putStr "And your last name? "
            last <- getLine
            let full = first ++ " " ++ last
            putStrLn ("Pleased to meet you, " ++ full ++ "!")
```

A possible translation into vanilla monadic code:

```
nameLambda :: IO ()
nameLambda = putStr "What is your first name? " >>
             getLine >>= \ first ->
             putStr "And your last name? " >>
             getLine >>= \ last ->
             let full = first ++ " " ++ last
             in putStrLn ("Pleased to meet you, " ++ full ++ "!")
```

In cases like this, where we just want to chain several actions, the imperative style of `do` notation feels natural and convenient. In comparison, monadic code with explicit binds and lambdas is something of an acquired taste.

Notice that the first example above includes a `let` statement in the `do` block. The de-sugared version is simply a regular `let` expression where the `in` part is whatever follows from the `do` syntax.

Returning values

The last statement in `do` notation is the overall result of the `do` block. In the previous example, the result was of the type `IO ()`, i.e. an empty value in the `IO` monad.

Suppose that we want to rewrite the example but return an `IO String` with the acquired name. All we need to do is add a `return`:

```
nameReturn :: IO String
nameReturn = do putStr "What is your first name? "
                first <- getLine
                putStr "And your last name? "
                last <- getLine
                let full = first ++ " " ++ last
                putStrLn ("Pleased to meet you, " ++ full ++ "!")
                return full
```

This example will "return" the full name as a string inside the `IO` monad, which can then be utilized downstream elsewhere:

```
greetAndSeeYou :: IO ()
greetAndSeeYou = do name <- nameReturn
                    putStrLn ("See you, " ++ name ++ "!")
```

Here, `nameReturn` will be run and the returned result (called "full" in the `nameReturn` function) will be assigned to the variable "name" in our new function. The greeting part of `nameReturn` will be printed to the screen because that is part of the calculation process. Then, the additional "see you" message will print as well, and the final returned value is back to being `IO ()`.

If you know imperative languages like C, you might think `return` in Haskell matches `return` elsewhere. A small variation on the example will dispel that impression:

```
nameReturnAndCarryOn = do putStr "What is your first name? "
                          first <- getLine
                          putStr "And your last name? "
                          last <- getLine
                          let full = first++" "++last
                          putStrLn ("Pleased to meet you, 
"++full++"!")
                          return full
                          putStrLn "I am not finished yet!"
```

The string in the extra line *will* be printed out because `return` is *not* a final statement interrupting the flow (as it would be in C and other languages). Indeed, the type of `nameReturnAndCarryOn` is `IO ()`, — the type of the final `putStrLn` action. After the function is called, the `IO String` created by the `return full` will disappear without a trace.

Just sugar

As a syntactical convenience, `do` notation does not add anything essential, but it is often preferable for clarity and style. However, `do` is not needed for a single action, at all. The Haskell "Hello world" is simply:

```
main = putStrLn "Hello world!"
```

Snippets like this one are totally redundant:

```haskell
fooRedundant = do { x <- bar
                  ; return x }
```

Thanks to the monad laws, we can write it simply as

```haskell
foo = do { bar }   -- which is, further,
foo = bar
```

A subtle but crucial point relates to function composition: As we already know, the `greetAndSeeYou` action in the section just above could be rewritten as:

```haskell
greetAndSeeYou :: IO ()
greetAndSeeYou = nameReturn >>= (\ name -> putStrLn ("See you, "
++ name ++ "!"))
```

While you might find the lambda a little unsightly, suppose we had a `printSeeYou` function defined elsewhere:

```haskell
printSeeYou :: String -> IO ()
printSeeYou name = putStrLn ("See you, " ++ name ++ "!")
```

Now, we can have a clean function definition with neither lambdas or `do`:

```haskell
greetAndSeeYou :: IO ()
greetAndSeeYou = nameReturn >>= printSeeYou
```

Or, if we have a *non-monadic* `seeYou` function:

```haskell
seeYou :: String -> String
seeYou name = "See you, " ++ name ++ "!"
```

Then we can write:

```haskell
-- Reminder: fmap f m  ==  m >>= (return . f)  ==  liftM f m
greetAndSeeYou :: IO ()
greetAndSeeYou = fmap seeYou nameReturn >>= putStrLn
```

Keep this last example with `fmap` in mind; we will soon return to using non-monadic functions in monadic code, and `fmap` will be useful there.

Notes

1. Actually, the indentation isn't needed in this case. This is equally valid:

   ```
   action1 >>= \ x1 ->
   action2 >>= \ x2 ->
   action3 x1 x2
   ```

   Of course, we could use even more indentation if we wanted. Here's an extreme example:

   ```
   action1
     >>=
       \
         x1
           ->
             action2
               >>=
                 \
                   x2
                     ->
                       action3
                         x1
                         x2
   ```

   While that indention is certainly overkill, it could be worse:

   ```
   action1
     >>= \
       x1
         -> action2 >>=
           \
             x2 ->
               action3 x1
                 x2
   ```

That is valid Haskell but is baffling to read; so please don't ever write like that. Write your code with consistent and meaningful groupings.

2. This explains why, as we pointed out in the "Pattern matching" chapter, pattern matching failures in list comprehensions are silently ignored.