binary: Serializing data

binary is straightforward to use library for decoding (parsing) and encoding binary data. There are a number of ways to work with binary, ways that range from simple to complex.

Decoding and encoding standard data types.

Byte strings can be trivially decoded to Haskell values, provided that the types of those values are instances of `Binary`. Analogously, types that are instances of `Binary` can easily be encoded back to byte strings. *binary* comes with convenient `Binary` instances for a good number of the standard data types including numbers, booleans, lists, tuples, and so on.

Let us try these instances out in GHCi. You'll need to import `Data.Binary` to run the examples below.

```
> encode True
"\SOH"

> decode (encode True) :: Bool
True

> encode 'A'
"A"

> decode (encode 'A') :: Char
'A'

> encode "AB"
"\NUL\NUL\NUL\NUL\NUL\NUL\NUL\STXAB"

> decode (encode "AB") :: String
"AB"

> encode ("A", "B")
"\NUL\NUL\NUL\NUL\NUL\NUL\NUL\SOHA\NUL\NUL\NUL\NUL\NUL\NUL\NUL\SOHB"

> decode (encode ("A", "B") ):: (String, String)
("A","B")
```

Automatic decoding and encoding

So how do you define `Binary` instances for types that do not already have them?

One simple way to do this is to use the `DeriveGeneric` extension and have the compiler generate the implementation for you.

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE DeriveGeneric #-}
{-# LANGUAGE OverloadedStrings #-}

import Data.Binary
import Data.Text
import GHC.Generics (Generic)

data Transaction =
  Txn { account :: Text, amount :: Float }
  deriving (Generic, Show)

instance Binary Transaction

main :: IO ()
main = do
  let bytes = encode (Txn { account = "Cash", amount = 10 })
  putStrLn $ "Encode: " ++ show bytes
  putStrLn $ "Decode: " ++
    show (decode bytes :: Transaction)
```

Custom decoding and encoding

`Binary` instances may also be written by hand.

For decoding, you need to define `get`, of type `Get t` where `Get` is an instance of `Monad`.

For encoding, you need to provide a definition for `put`, which is a function that takes a value of the type you wish to encode and a value c type `Put`. By definition, `Put = PutM ()` where `PutM` is also an instance of `Monad`.

```haskell
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE OverloadedStrings #-}
import Data.Binary
import Data.Text (Text)


data Transaction =  -- Same type as before, but without the Generic instance.
  Txn { account :: Text, amount :: Float } deriving Show

instance Binary Transaction where
  put (Txn acct amt) = do
    put acct
    put amt

  get = do
    acct <- get
    amt <- get
    return $ Txn acct amt


main :: IO () -- The main action is unchanged from before.
main = do
  let bytes = encode (Txn { account = "Cash", amount = 1000 })
  putStrLn $ "Encode: " ++ show bytes
  putStrLn $ "Decode: " ++
    show (decode bytes :: Transaction)
```

Conclusion

While quite straightforward to use, the *binary* library is vulnerable to two criticisms. Firstly, as its creator Duncan Coutts pointed out, `binary` entangles the issue of serializing Haskell values with that of read/writing externally defined formats. This can be confusing. Secondly, the error messages one may encounter when using `binary`are perhaps not as helpful as one would like. The `cereal` library offers to address the latter (though not the former) problem.