



SimpleJSON (or creating useful modules)

Jacek Malec

Dept. of Computer Science, Lund University, Sweden

April 2, 2019



Source

The source material for this lecture is Chapter 5 in the book
“Real World Haskell”

by Bryan O’Sullivan, Don Stewart, and John Goerzen

available in its entirety at <http://book.realworldhaskell.org>

O’Reilly, 2008



What are modules and why should I care?

- modules provide libraries
- standardized way to package useful code
- interoperability
- wide distribution
- three levels:
 - ① GHC standard libraries (Haskell 2010)
 - ② Haskell Platform Libraries
 - ③ Hackage database
- Library installer for Haskell: `cabal`
- The Common Architecture for Building Applications and Libraries
- Browse hackage.haskell.org for more...



What is JSON and why should I care?

- Lightweight data exchange format
- JSON = Java Script Object Notation
- Communication between a server and applets on client side
- www.json.org, RFC 4627 (www.ietf.org/rfc/rfc4627.txt)
- simple types (string, number, boolean, null) and compound ones (arrays and objects)
- an object is a collection of (name : JSON value) pairs
- and that’s it!



Data constructors

```
data JValue = JString String
            | JNumber Double
            | JBool Bool
            | JNull
            | JObject [(String, JValue)]
            | JArray [JValue]
            deriving (Eq, Ord, Show)
```



Data renderers

```
-- file: ch05/PutJSON.hs
module PutJSON where

import Data.List (intercalate)
import SimpleJSON

renderJValue :: JValue -> String

renderJValue (JString s)   = show s
renderJValue (JNumber n)   = show n
renderJValue (JBool True)  = "true"
renderJValue (JBool False) = "false"
renderJValue JNull         = "null"
```



Data renderers

```
renderJValue (JObject o) = "{" ++ pairs o ++}"
  where pairs [] = ""
        pairs ps = intercalate ", " (map renderPair ps)
        renderPair (k,v) = show k ++ ": " ++ renderJValue v

renderJValue (JArray a) = "[" ++ values a ++]"
  where values [] = ""
        values vs = intercalate ", " (map renderJValue vs)
```



A bit more to do

```
-- file: ch05/PutJSON.hs
putJValue :: JValue -> IO ()
putJValue v = putStrLn (renderJValue v)

Now, the following:
module Main () where

import SimpleJSON
import PutJSON

main = putJValue (JObject [("foo", JNumber 1),
                           ("bar", JBool False)])

yields

{"foo": 1.0, "bar": false}
```



We are ready! Not yet.

- We have a representation for all JSON objects
- We can render them as strings (before possibly sending out)
- Why anything else?

It can be done more generally.

Let's introduce pretty-printing

- for machine readability (JSON)
- for human readability

Module will be called Prettify.



New version of renderer

```
-- file: ch05/PrettyJSON.hs
renderJValue :: JValue -> Doc
renderJValue (JBool True) = text "true"
renderJValue (JBool False) = text "false"
renderJValue JNull = text "null"
renderJValue (JNumber num) = double num
renderJValue (JString str) = string str
```



Interactiveness

```
-- file: ch05/PrettyStub.hs
import SimpleJSON

data Doc = ToBeDefined
  deriving (Show)

string :: String -> Doc
string str = undefined

text :: String -> Doc
text str = undefined

double :: Double -> Doc
double num = undefined
```



Pretty printing a string

```
-- file: ch05/PrettyJSON.hs
string :: String -> Doc
string = enclose '"' ' ' . hcat . map oneChar

enclose :: Char -> Char -> Doc -> Doc
enclose left right x = char left <> x <> char right

(<>) :: Doc -> Doc -> Doc
a <> b = undefined

char :: Char -> Doc
char c = undefined

hcat :: [Doc] -> Doc
hcat xs = undefined
```



Pretty printing a string

```
-- file: ch05/PrettyJSON.hs
oneChar :: Char -> Doc
oneChar c = case lookup c simpleEscapes of
    Just r -> text r
    Nothing | mustEscape c -> hexEscape c
            | otherwise -> char c
  where mustEscape c = c < ' ' || c == '\x7f' || c > '\xff'

simpleEscapes :: [(Char, String)]
simpleEscapes = zipWith ch "\b\n\f\r\t\\\"/" "bnfrt\\\"/"
  where ch a b = (a, ['\\',b])

hexEscape :: Char -> Doc
hexEscape c | d < 0x10000 = smallHex d
            | otherwise = astral (d - 0x10000)
  where d = ord c
```



Pretty printing complex objects

```
-- file: ch05/PrettyJSON.hs
series :: Char -> Char -> (a -> Doc) -> [a] -> Doc
series open close item = enclose open close
    . fsep . punctuate (char ',') . map item

-- file: ch05/PrettyStub.hs
fsep :: [Doc] -> Doc
fsep xs = undefined

punctuate :: Doc -> [Doc] -> [Doc]
punctuate p [] = []
punctuate p [d] = [d]
punctuate p (d:ds) = (d <> p) : punctuate p ds
```



Pretty printing complex objects

```
-- file: ch05/PrettyJSON.hs
renderJValue (JArray ary) = series '[' ']' renderJValue ary

renderJValue (JObject obj) = series '{' '}' field obj
  where field (name,val) = string name
    <> text ":"
    <> renderJValue val
```



Module header

```
-- file: ch05/PrettyJSON.hs
module PrettyJSON
(
    renderJValue
) where

import Numeric (showHex)
import Data.Char (ord)
import Data.Bits (shiftR, (.&))

import SimpleJSON (JValue(..))
import Prettify (Doc, (<>), char, double, fsep, hcat, punctuate,
    text, compact, pretty)
```



Concretizing Prettify.hs

```
-- file: ch05/Prettify.hs
data Doc = Empty
  | Char Char
  | Text String
  | Line
  | Concat Doc Doc
  | Union Doc Doc
  deriving (Show,Eq)

empty :: Doc
empty = Empty

char :: Char -> Doc
char c = Char c
```



Concretizing Prettify.hs

```
-- file: ch05/Prettify.hs
text :: String -> Doc
text "" = Empty
text s = Text s

double :: Double -> Doc
double d = text (show d)

line :: Doc
line = Line

(<>) :: Doc -> Doc -> Doc
Empty <> y = y
x <> Empty = x
x <> y = x 'Concat' y
```



Concretizing Prettify.hs

```
-- file: ch05/Prettify.hs
hcat :: [Doc] -> Doc
hcat = fold (<>)

fold :: (Doc -> Doc -> Doc) -> [Doc] -> Doc
fold f = foldr f empty

fsep :: [Doc] -> Doc
fsep = fold (</>)

(</>) :: Doc -> Doc -> Doc
x </> y = x <> softline <> y

softline :: Doc
softline = group line
```



Keeping alternatives

```
-- file: ch05/Prettify.hs
group :: Doc -> Doc
group x = flatten x 'Union' x

flatten :: Doc -> Doc
flatten (x 'Concat' y) = flatten x 'Concat' flatten y
flatten Line           = Char ' '
flatten (x 'Union' _)  = flatten x
flatten other          = other
```



The real work: machine

```
-- file: ch05/Prettify.hs
compact :: Doc -> String
compact x = transform [x]
  where transform [] = ""
        transform (d:ds) =
          case d of
            Empty      -> transform ds
            Char c      -> c : transform ds
            Text s      -> s ++ transform ds
            Line        -> '\n' : transform ds
            a 'Concat' b -> transform (a:b:ds)
            _ 'Union' b  -> transform (b:ds)
```



The real work: man

```
-- file: ch05/Prettify.hs
pretty :: Int -> Doc -> String
pretty width x = best 0 [x]
  where best col (d:ds) =
          case d of
            Empty      -> best col ds
            Char c      -> c : best (col + 1) ds
            Text s      -> s ++ best (col + length s) ds
            Line        -> '\n' : best 0 ds
            a 'Concat' b -> best col (a:b:ds)
            a 'Union' b  -> nicest col (best col (a:ds))
                               (best col (b:ds))

          best _ _ = ""
          nicest col a b | (width - least) 'fits' a = a
                        | otherwise                = b
          where least = min width col
```



The real work: man

```
-- file: ch05/Prettify.hs
fits :: Int -> String -> Bool
w 'fits' _ | w < 0 = False
w 'fits' ""       = True
w 'fits' ('\n':_) = True
w 'fits' (c:cs)  = (w - 1) 'fits' cs
```



Results

```
*PrettyJSON> let value = renderJValue (JObject [{"f",
                                                JNumber 1},
                                                {"q",
                                                 JBool True}])

*PrettyJSON> :t value
value :: Doc
*PrettyJSON> putStrLn (pretty 10 value)
{"f": 1.0,
 "q": true
}
*PrettyJSON> putStrLn (pretty 20 value)
{"f": 1.0, "q": true
}
*PrettyJSON> putStrLn (pretty 30 value)
{"f": 1.0, "q": true }
*PrettyJSON>
```



Cabal

The Common Architecture for Building Applications and Libraries

- Cabal organizes software in *packages*
- A package is a *library* and possibly several executable programs
- package description: *.cabal
- package *setup* file: Setup.hs
- then the usual triple:


```
runghc Setup {configure, build, install}
```
- or sometimes just cabal install foobar



Hackage 2010

hackageDB :: [Package]

Introduction	Packages	Hayo!	What's new	Upload	User accounts
--------------	----------	-------	------------	--------	---------------

Packages by category

Categories: NET (2), Accessibility (1), ACME (3), AI (11), Algebra (3), Algorithms (44), Algorithms Network (1), Animation (3), Application (5), Audio (1), Backup (1), Benchmarking (2), Bindings (1), Bioinformatics (23), Browser (1), BSD (1), CGI (1), Classification (2), Clustering (4), Code Generation (6), Codec (47), Codescs (2), Combinator (1), Combinators (6), Comonads (1), Compiler (11), Compilers/Interpreters (37), Composition (5), Compression (1), Concurrency (54), Concurrent (6), Configuration (1), Console (22), Control (120), Crosswords (1), Crypto (1), Cryptography (22), Data (345), Data Mining (5), Data Structures (54), Database (65), Datamining (2), Debug (10), Dependent Types (6), Desktop (7), Development (120), Digest (1), Disassembler (1), Distributed Computing (14), Distribution (36), Documentation (3), Editor (6), Education (6), Email (3), Embedded (4), Enumerator (10), Error Handling (6), Factual (1), Failure (16), FFT (3), FFI Tools (2), Finance (3), Foreign Binding (1), Formal Methods (7), FPP (21), Game (69), Game Engine (1), Generic (1), Generics (27), Gentoo (1), GHC (3), GIS Programs (1), Graph (1), Graphics (185), Graphs (9), GUI (30), Hardware (10), Help (2), Heuristics (1), IDE (4), Interfaces (5), IRC (2), JSON (1), Language (128), List (5), Local Search (1), Logic (1), Math (114), Media (5), Middleware (3), Monad (4), Monadic Regions (11), Monads (44), Music (31), Natural Language Processing (21), Network (154), Network APIs (1), Networking (1), Number Theory (1), Numeric (3), Numerical (26), OAuth (1), Optimisation (1), Other (6), Parsing (55), Parsing Text (1), Pattern Classification (1), Performance (1), Phantom Types (1), Physics (6), PL/SQL Tools (1), Profiling (6), Program Transformation (1), Protocol (2), Pugs (9), Quantum (1), Reactivity (15), Records (1), Refactoring (1), Reflection (4), RFC (1), Robotics (1), Scientific Simulation (1), Screensaver (1), Scripting (1), Search (4), Security (3), Semantic Web (1), Sound (83), Source Tools (1), Source-tools (5), Statistics (8), Stochastic Control (1), System (189), System Console (1), Template Haskell (4), Test (1), Testing (36), Text (192), Theorem Provers (11), Thread Profiling Utility (1), Tools (2), Trace (4), Type System (8), UI (1), Uniform (4), User Interfaces (39), User-interface (1), Utility (2), Utils (25), Video (1), Visual Programming (2), Web (183), Web Server (1), Workflow (1), XML (40), Yampa (1), Unclassified (29).



mypretty.cabal

Name: mypretty
Version: 0.1
Synopsis: My pretty printing library, with JSON support
Description:
 A simple pretty printing library that illustrates how to develop a Haskell library.
Author: Real World Haskell
Maintainer: nobody@realworldhaskell.org
Cabal-Version: >= 1.2
library
Exposed-Modules: Prettify
 PrettyJSON
 SimpleJSON
Build-Depends: base >= 2.0



Hackage 2019

Packages by category

Categories: AI (11), Accessibility (1), ACME (3), AI (11), Algebra (3), Algorithms (44), Algorithms Network (1), Animation (3), Application (5), Audio (1), Backup (1), Benchmarking (2), Bindings (1), Bioinformatics (23), Browser (1), BSD (1), CGI (1), Classification (2), Clustering (4), Code Generation (6), Codec (47), Codescs (2), Combinator (1), Combinators (6), Comonads (1), Compiler (11), Compilers/Interpreters (37), Composition (5), Compression (1), Concurrency (54), Concurrent (6), Configuration (1), Console (22), Control (120), Crosswords (1), Crypto (1), Cryptography (22), Data (345), Data Mining (5), Data Structures (54), Database (65), Datamining (2), Debug (10), Dependent Types (6), Desktop (7), Development (120), Digest (1), Disassembler (1), Distributed Computing (14), Distribution (36), Documentation (3), Editor (6), Education (6), Email (3), Embedded (4), Enumerator (10), Error Handling (6), Factual (1), Failure (16), FFT (3), FFI Tools (2), Finance (3), Foreign Binding (1), Formal Methods (7), FPP (21), Game (69), Game Engine (1), Generic (1), Generics (27), Gentoo (1), GHC (3), GIS Programs (1), Graph (1), Graphics (185), Graphs (9), GUI (30), Hardware (10), Help (2), Heuristics (1), IDE (4), Interfaces (5), IRC (2), JSON (1), Language (128), List (5), Local Search (1), Logic (1), Math (114), Media (5), Middleware (3), Monad (4), Monadic Regions (11), Monads (44), Music (31), Natural Language Processing (21), Network (154), Network APIs (1), Networking (1), Number Theory (1), Numeric (3), Numerical (26), OAuth (1), Optimisation (1), Other (6), Parsing (55), Parsing Text (1), Pattern Classification (1), Performance (1), Phantom Types (1), Physics (6), PL/SQL Tools (1), Profiling (6), Program Transformation (1), Protocol (2), Pugs (9), Quantum (1), Reactivity (15), Records (1), Refactoring (1), Reflection (4), RFC (1), Robotics (1), Scientific Simulation (1), Screensaver (1), Scripting (1), Search (4), Security (3), Semantic Web (1), Sound (83), Source Tools (1), Source-tools (5), Statistics (8), Stochastic Control (1), System (189), System Console (1), Template Haskell (4), Test (1), Testing (36), Text (192), Theorem Provers (11), Thread Profiling Utility (1), Tools (2), Trace (4), Type System (8), UI (1), Uniform (4), User Interfaces (39), User-interface (1), Utility (2), Utils (25), Video (1), Visual Programming (2), Web (183), Web Server (1), Workflow (1), XML (40), Yampa (1), Unclassified (29).

* mypretty is a Haskell library. It is not a Haskell package.
 * mypretty is a Haskell library. It is not a Haskell package.