

PROGRAMMING IN HASKELL



Parsing CSV files using cassava

Parsers

There are many library collections/packages in Haskell.

There are many libraries to help us parse information:
Parsing involves taking 'flat' information and returning structured data, e.g.

- a sentence can be parsed into its component parts (verbs, nouns, etc.)
- a program can be parsed into the various statement blocks (e.g. if .. else)
- A csv file (which can be seen as one long String) can be parsed into a collection of a matching data type/model
- a json file can similarly be parsed into a collection of matching data type/model

csv parser libraries

There are various libraries that help us parse csv files.
e.g.

- Data.Csv (cassava is the name of the package)
- parsec
- csv-conduit

cassava

The two important functions are

- | **encode** – takes structured text and encodes it as csv
- | **decode** – takes csv format and returns structured data when possible (based on your defined structure)

cassava options

Does the csv file have a header?

no

Use decode

yes

Use
decodeByName

Does the csv file have
exactly the same
structure as your data
structure? ?

yes

Use decode with generic
FromRecord/FromNamedRecord

no

Rewrite
FrommRecord/FromNamedRecord
tp reshape data for your data
structure

Using Data.Csv and cassava

A good complete example is here :

<https://www.stackbuilders.com/tutorials/haskell/csv-encoding-decoding/>

The data is of the form (note heading)

Item,Link,Type

Japan,<http://www.data.go.jp/>,International Country

United Kingdom,<http://data.gov.uk/>,International Country

United Nations,<http://data.un.org/>,International Regional

Uruguay,<http://datos.gub.uy/>,International Country

Utah,<http://www.utah.gov/data/>,US State

Vancouver,<http://data.vancouver.ca/>,International Regional

Using Data.Csv and cassava

We want to parse this data into a collection of Items:

```
data Item =  
  Item  
  { itemName :: Text  
  , itemLink :: Text  
  , itemType :: ItemType  
  }  
deriving (Show,Eq)
```

```
data ItemType  
  = Country  
  | Other Text  
deriving (Show,Eq)
```

decodeByName

```
decodeByName :: FromNamedRecord a =>  
    ByteString -> Either String (Header, Vector a)
```

This can be
default or
defined by
you

The
header

Either to
allow for
errors

Error
message

Collection
of the
parsed
data

FromNamedRecord

In order to decode one such record into a value of type Item, we need Item to be an instance of either

- FromRecord (no header) or
- FromNamedRecord (header),

Cassava's type classes for decoding CSV records.

In this case, the CSV file has a header, so we need to make Item an instance of FromNamedRecord.

Also, in this case, the default parsing will not work (need the header names to be the same as field names of Item – this is not possible as the header names are Uppercase.)

Instance FromNamedRecord

We need to implement
parseNamedRecord to
declare an instance of
FromNamedRecord

instance FromNamedRecord Item *where*

```
parseNamedRecord m =  
  Item  
  <$> m .: "Item"  
  <*> m .: "Link"  
  <*> m .: "Type"
```

.: is a lookup operator
so this looks for "Item"
in the header and if it
exists, we use that field
to correspond to the
first field of Item,
itemName

Parsing fields

Usually the fields will be parsed by default if the parsing is standard, e.g. for `itemName` and `ItemLink` (they are `Text` and `cassava` manages that). If you need to manage this, use `FromField`:

```
instance FromField ItemType where
  parseField "International Country" =
    pure Country
  parseField otherType =
    Other <$> parseField otherType
```

Note the
FromField
Day in the
StockQuotes
example

Parsing fields.. Dealing with empty fields

For instance, in our StockQuotes example, we use a type Fixed2 (which holds a double number to 2 decimal places (see code in QuoteData.hs)). If there is an error in this field (e. g. empty (we would like to default the value to 0. We do this as follows:

```
instance FromField Fixed2 where
  parseField = pure . readDef 0 . unpack
  -- readDef 0 is a safe read -if parsing of the string fails, it
  -- returns 0 – we need to include import Safe(readDef) and
  -- include safe in package.yaml
```

