# Comma Police: The Design and Implementation of a CSV Library

George Wilson

Data61/CSIRO

george.wilson@data61.csiro.au

23rd May 2018

JSON
YAML
XML
CSV
PSV

# SV

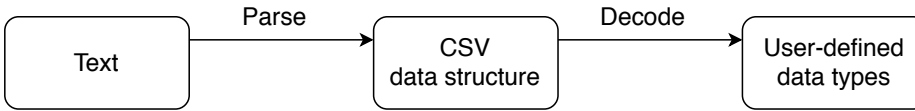
{CSV, PSV, . . . } library for Haskell

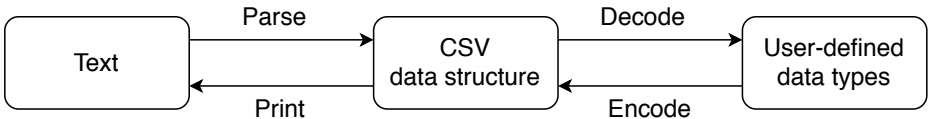
CSV

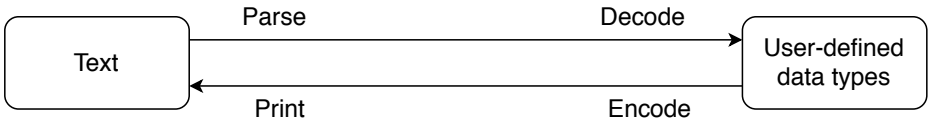
- Very popular format for data science
- Described *not standardised* by RFC 4180

example.csv

```
"id","species","count"
1,"kangaroo",30
2,"kookaburra",460
3,"platypus",5
```


Text
Parse
CSV
data structure
Decode
User-defined
data types


Text
Parse
Print
CSV
data structure
Decode
Encode
User-defined
data types


Text
Parse
Decode
User-defined
data types
Print
Encode

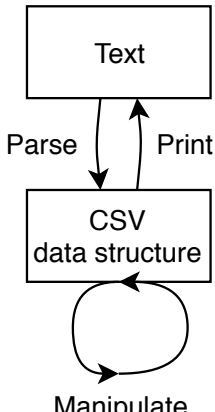
```
parse :: ByteString -> Either ByteString (Sv ByteString)

decode :: Decode s a -> Sv s -> DecodeValidation a

encodeSv :: Encode a -> [a] -> Sv ByteString

printSv :: Sv ByteString -> ByteString
```

**Direct**

- less memory allocated
- faster
- streaming made easier

**Intermediate structure**

- potential for better errors (often)
- make decisions based on the structure
- manipulate the tree to alter documents


Text
Parse
Print
CSV
data structure
Manipulate

needs-fixing.csv

```
'name',"age"
"Frank",30
George, '25'
"Harry","32"
```

```haskell
fixQuotes :: Sv s -> Sv s
fixQuotes =
  over headerFields fixQuote . over recordFields fixQuote
    where
      headerFields = traverseHeader . fields
      recordFields = traverseRecords . fields

      fixQuote :: Field a -> Field a
      fixQuote f = case f of
        Unquoted a -> Quoted DoubleQuote (noEscape a)
        Quoted _ v -> Quoted DoubleQuote v
```

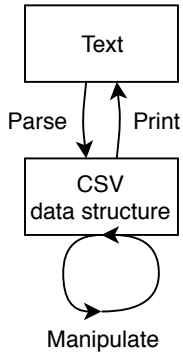
needs-fixing.csv

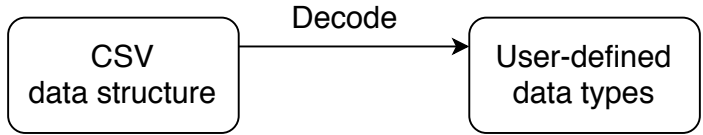
```
'name',"age"
"Frank",30
George, '25'
"Harry","32"
```

fixed.csv

```
"name","age"
"Frank","30"
"George", "25"
"Harry","32"
```

Use `sv` to define custom linters and sanitisers




Decode
CSV
data structure
User-defined
data types

```haskell
data Decode s a = ...
```

```haskell
data Decode s a = ...


raw    :: Decode a a
ignore :: Decode a ()
int    :: Decode ByteString Int
ascii  :: Decode ByteString String
text   :: Decode ByteString Text
```

```haskell
data Decode s a = ...


raw    :: Decode a a
ignore :: Decode a ()
int    :: Decode ByteString Int
ascii  :: Decode ByteString String
text   :: Decode ByteString Text


instance Functor (Decode s)
instance Applicative (Decode s)
instance Alt (Decode s) where
```

person.csv

```
"name","age"
"Frank","30"
"George", "25"
"Harry","32"
```

person.csv

```
"name","age"
"Frank","30"
"George", "25"
"Harry","32"
```

```haskell
data Person = Person Text Int
```

person.csv

```
"name","age"
"Frank","30"
"George", "25"
"Harry","32"
```

```haskell
data Person = Person Text Int


personD :: Decode ByteString Person
personD = Person <$> text <*> int
```

ragged.csv

```
"George","Wilson",25
"Frank",33
"Tim",18
"John","Smith",45
```

ragged.csv

```
"George","Wilson",25
"Frank",33
"Tim",18
"John","Smith",45
```

```haskell
data Person
  = OneName Text Int
  | TwoNames Text Text Int
```

ragged.csv

```
"George","Wilson",25
"Frank",33
"Tim",18
"John","Smith",45
```

```
data Person
  = OneName Text Int
  | TwoNames Text Text Int


personDecoder :: Decode Person
personDecoder =
      OneName <$> text <*> int
  <!> TwoNames <$> text <*> text <*> int
```

```haskell
class Profunctor p where
  dimap :: (a -> b) -> (c -> d) -> p b c -> p a d

instance Profunctor Decode
```

```haskell
class Profunctor p where
  dimap :: (a -> b) -> (c -> d) -> p b c -> p a d

instance Profunctor Decode


-- make a Decode work on a different string type
decoder :: Decode ByteString A
input :: Text
```

```haskell
class Profunctor p where
  dimap :: (a -> b) -> (c -> d) -> p b c -> p a d

instance Profunctor Decode


-- make a Decode work on a different string type
decoder :: Decode ByteString A
input :: Text

encodeUtf8 :: Text -> ByteString
```

```haskell
class Profunctor p where
  dimap :: (a -> b) -> (c -> d) -> p b c -> p a d

instance Profunctor Decode


-- make a Decode work on a different string type
decoder :: Decode ByteString A
input :: Text

encodeUtf8 :: Text -> ByteString

dimap encodeUtf8 id decoder :: Decode Text A
```

## Why not a type class?

- A decoder is something I want to *manipulate*
- There are often many different ways to decode the same type

```haskell
ignoreFailure :: Decode s a -> Decode s (Maybe a)
ignoreFailure a =
      Just <$> a
  <!> Nothing <* ignore
```

```haskell
ignoreFailure :: Decode s a -> Decode s (Maybe a)
ignoreFailure a =
      Just <$> a
  <!> Nothing <* ignore
```

ints.csv

```
3
4
8.8
1
null
```

```haskell
ignoreFailure :: Decode s a -> Decode s (Maybe a)
ignoreFailure a =
      Just <$> a
  <!> Nothing <* ignore
```

ints.csv

```
3
4
8.8
1
null
```

```haskell
parseDecodefromFile (ignoreFailure int) "ints.csv"

-- [Just 3, Just 4, Nothing, Just 1, Nothing]
```

```haskell
-- succeeds with Nothing when
-- the underlying decoder fails
ignoreFailure :: Decode s a -> Decode s (Maybe a)

-- succeeds with Nothing only when
-- the field is completely empty
orEmpty :: Decode s a -> Decode s (Maybe a)

-- succeeds with Nothing only when
-- there is no field at all
optionalField :: Decode s a -> Decode s (Maybe a)
```

conferences.csv

```
"name","date"
"Compose Conf",20170828
"Compose Conf",20180827
"Lambda Jam",20170508
"Lambda Jam",20180521
```

```haskell
import Data.Thyme

data Conference = Conf Text YearMonthDay
```

```haskell
import Data.Thyme

data Conference = Conf Text YearMonthDay


ymdParser :: A.Parser YearMonthDay
ymdParser = buildTime <$> timeParser defaultTimeLocale "%Y%m%d"
```

```haskell
import Data.Thyme

data Conference = Conf Text YearMonthDay


ymdParser :: A.Parser YearMonthDay
ymdParser = buildTime <$> timeParser defaultTimeLocale "%Y%m%d"


trifecta    :: T.Parser a -> Decode ByteString a
attoparsec :: A.Parser a -> Decode ByteString a
```

```haskell
import Data.Thyme

data Conference = Conf Text YearMonthDay


ymdParser :: A.Parser YearMonthDay
ymdParser = buildTime <$> timeParser defaultTimeLocale "%Y%m%d"


trifecta   :: T.Parser a -> Decode ByteString a
attoparsec :: A.Parser a -> Decode ByteString a


ymd :: Decode YearMonthDay
ymd = attoparsec ymdParser
```

```haskell
import Data.Thyme

data Conference = Conf Text YearMonthDay


ymdParser :: A.Parser YearMonthDay
ymdParser = buildTime <$> timeParser defaultTimeLocale "%Y%m%d"


trifecta   :: T.Parser a -> Decode ByteString a
attoparsec :: A.Parser a -> Decode ByteString a


ymd :: Decode YearMonthDay
ymd = attoparsec ymdParser

confD :: Decode ByteString Conference
confD = Conf <$> text <*> ymd
```

`sv` uses error values

```
data DecodeError s
  = UnexpectedEndOfRow
  | ExpectedEndOfRow [Field s]
  | BadParse s
  | BadDecode s
  ...
```

```
onError :: Decode s a
        -> (DecodeErrors s -> Decode s a)
        -> Decode s a
```

Rather than `Either` for errors, sv uses the `Validation` data type

```
data Validation e a = Failure e | Success a
```

Rather than `Either` for errors, sv uses the `Validation` data type

```
data Validation e a = Failure e | Success a

instance Semigroup e => Applicative (Validation e)
```

Rather than `Either` for errors, sv uses the `Validation` data type

```haskell
data Validation e a = Failure e | Success a
```

```haskell
instance Semigroup e => Applicative (Validation e)
```

```haskell
newtype DecodeErrors s =
  DecodeErrors (NonEmpty (DecodeError s))
  deriving Semigroup
```

example.csv

```
"a","b","c"
```

example.csv

```
"a","b","c"
```

```haskell
data Two = Two Int Int
```

example.csv

```
"a","b","c"
```

```haskell
data Two = Two Int Int

twoD :: Decode ByteString Two
twoD = Two <$> int <*> int
```

example.csv

```
"a","b","c"
```

```haskell
data Two = Two Int Int

twoD :: Decode ByteString Two
twoD = Two <$> int <*> int

parseDecodeFromFile twoD "example.csv"
```

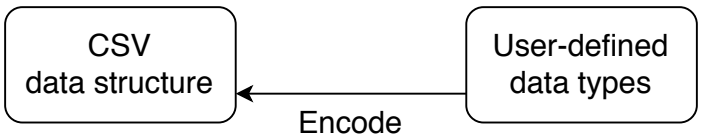
example.csv

```
"a","b","c"
```

```
data Two = Two Int Int

twoD :: Decode ByteString Two
twoD = Two <$> int <*> int

parseDecodeFromFile twoD "example.csv"
```

```
Failure (DecodeErrors (
    BadDecode "Couldn't parse \"a\" as an int" :|
  [ BadDecode "Couldn't parse \"b\" as an int"
  , ExpectedEndOfRow ["c"]
  ]
))
```

What about encoding?



```
data Encode a = ...
```

```haskell
data Encode a = ...


int    :: Encode Int
double :: Encode Double
string :: Encode String
const  :: ByteString -> Encode a
encodeOf :: Prism' s a -> Encode a -> Encode s
```

```haskell
data Encode a = ...


int    :: Encode Int
double :: Encode Double
string :: Encode String
const  :: ByteString -> Encode a
encodeOf :: Prism' s a -> Encode a -> Encode s


instance Semigroup     (Encode a)

instance Contravariant  Encode
instance Divisible      Encode
instance Decidable      Encode
```

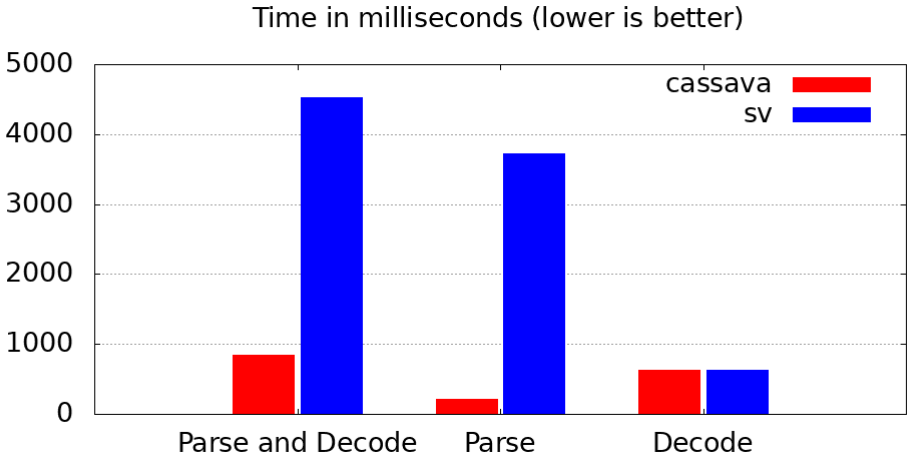
# Is it fast?

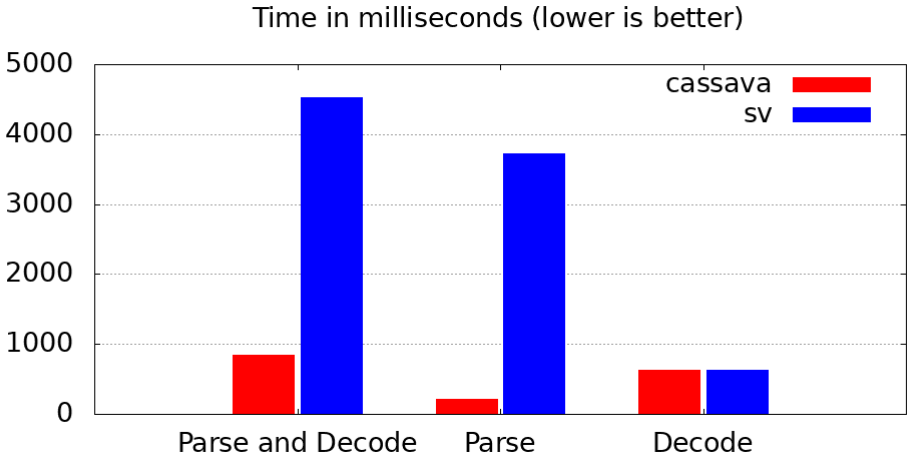
Is it fast?

No

Benchmarks

- Benchmarked with a 100,000 line
- Text, ints, doubles, products, sums
- cassava vs. sv (instantiated to attoparsec)


Time in milliseconds (lower is better)
5000
4000
3000
2000
1000
0
cassava
sv
Parse and Decode
Parse
Decode




Use `sv-cassava` for now

**Noteworthy limitations as at 2018-05-23**

- No column-name-based decoding
- Errors don't report source-file positions
- No streaming
- Performance needs work (particularly in parsing)

Contributions to sv are welcome.

Do you have a crazy CSV file to challenge sv?

Contact me at george.wilson@data61.csiro.au

# References

- `sv` library
  `https://github.com/qfpl/sv`
  `https://github.com/qfpl/sv-cassava`
- validation data type
  `https://hackage.haskell.org/package/validation`
  `https://hackage.haskell.org/package/either`
- CSV RFC
  `https://tools.ietf.org/html/rfc4180`
- Hedgehog
  `https://hackage.haskell.org/package/hedgehog`

Thanks for listening!