```
{-# LANGUAGE TypeSynonymInstances, FlexibleInstances, OverlappingInstances,
OverloadedStrings, Arrows #-}

module CsvDatabase
    (
        Row,
        popRow,
        popCell,
        tailTdb,
        Db,
        Selector(..),
        LValue (LN, LS),
        exec,
        Col,
        dselect,
        isdouble,
        mfoldl,
        fromDb,
        fromTdb,
        apply,
        fromCsv,
        Tdb,
        sortdb,
        CParser(..)
    ) where

import Control.Applicative
import Data.Monoid
import Data.List
-- Hide a few names that are provided by Applicative.
import Text.ParserCombinators.Parsec hiding ( many, optional, (<|>))

import Control.Arrow ((>>>))
import Prelude hiding (id)
import Control.Category (id)
import Control.Monad (forM_, MonadPlus(..), ap)
import Control.Arrow (arr, (>>^), (&&&), (>>>), (***), second)
import Data.Monoid (mempty, mconcat)
import qualified Data.Map as M
import Numeric (readSigned, readFloat, readHex)
import System.FilePath (dropExtension, takeFileName)

import CsvParser

-- We have 2 types, string or numbers = Maybe Double
-- So, strings are NOT NULL, NULL = Nothing in the Maybe Double
data LValue = LS [String] | LN [Maybe Double] deriving (Eq, Ord)

tailLv::LValue->LValue
tailLv lv = case lv of
                LS ss -> LS $ tail ss
                LN ns -> LN $ tail ns

-- Overide the standard show
instance Show (Maybe Double) where
    show (Just v) = show v
    show Nothing = ""
```

```
instance Monoid LValue where
    mappend l1 l2 = case (l1, l2) of
                    (LN ln1, LN ln2) -> LN (ln1++ln2)
                    (LN ln1, LS ls2) -> LS ((map show ln1)++ls2)
                    (LS ls1, LN ln2) -> LS (ls1++ (map show ln2))
                    (LS ls1, LS ls2) -> LS (ls1++ls2)
    mempty = LN []

-- Shows the LValue in a column - for the xml file for charts package
-- Shows the strings for when it's eg the names of the books
-- Shows the numbers for when it's eg the scores
instance Show LValue where
    show (LN ln) = concatMap (\md-> "<number>" ++ show md ++ "</number>" ) ln
    show (LS ls) = concatMap (\ms-> "<string>" ++ ms ++ "</string>" ) ls


type Csv = [[Value]]
type Db = [Row]
type Row = [Cell]
type Cell = (String, Value) -- key, value
type Col = (String, LValue)     -- ie. all the same field - we need to make them all doubles
or all strings...
type Tdb = [Col]

-- Writes a Db as a html table
instance Show Db where
    show [] = "<table></table>"
    show db = "<table>" ++ (write_header (head db)) ++ (concatMap show db) ++ "</table>"

-- Writes a Record as a html table-record
-- where each field is a <td>
instance Show Row where
    show r = "<tr>" ++ concatMap (\f -> "<td>" ++ show (snd f) ++ "</td>") r ++ "</tr>"

-------------------------------------------------------------------------
-- Writes a header from the strings of each field
-------------------------------------------------------------------------
write_header::Row->String
write_header r = "<tr>" ++ concatMap (\f ->"<th>" ++ fst f ++ "</th>") (r) ++ "</tr>"

tailCol::Col->Col
tailCol c = (fst c, tailLv $ snd c)

-- Add a field to a column
-- If one of the fields is a string then we get a sring
pushCell::Cell->Col->Col
pushCell fld col = case (fst fld == fst col) of
                        True -> ( fst fld, lv `mappend` snd col )
                                where lv = case (snd fld) of
                                                N n -> LN [n]
                                                S s -> LS [s]
                        False -> col -- adds nothing

popCell::Col->Cell
popCell c = case snd c of -- This is a fiddle because if [] there's something wrong
                LN [] -> (fst c, N Nothing)
```

```
                LS [] -> (fst c, N Nothing)
                LN (h:ts) -> (fst c, N h)
                LS (h:ts) -> (fst c, S h)

-- Add a Record to a Tdb (transposed Db)
pushRow::Row->Tdb->Tdb
pushRow = zipWith pushCell

popRow::Tdb->Row
popRow tdb = map popCell tdb

-- Turn a field into a column
cell2col::Cell->Col
cell2col f = (fst f, lv)
                where lv = case snd f of
                        N n -> LN [n]
                        S s -> LS [s]

-- Turn a record into a Tdb
row2tdb::Row->Tdb
row2tdb = map cell2col

-- Turn a Db into a Tdb
fromDb::Db->Tdb
fromDb db = foldl (flip pushRow) (row2tdb $ head db) (tail db)

tailTdb::Tdb->Tdb
tailTdb tdb = map tailCol tdb

--Turn a Tdb into a Db
fromTdb::Tdb->Db
fromTdb tdb = fromTdb' tdb []
    where fromTdb'::Tdb->Db->Db
          fromTdb' tdb db = case snd (head tdb) of
                                LN [] -> db
                                LS [] -> db
                                _     -> fromTdb' (tailTdb tdb) ((popRow tdb):db)

isdouble::Col->Bool
isdouble c = case (snd c) of
                LN n -> True
                LS s -> False


-- The headers are the first row, turned into strings, if not already
fromCsv::Csv->Db
fromCsv [] =  []
fromCsv (h:rs) =  map (\r-> zipWith (\hf rf ->
                                        case hf of
                                            S shf->(shf, rf)
                                            N dhf->(show dhf, rf)
                                    ) h r) rs

type Query = [Filter]
type Filter = (Selector, ValueFilter)
data Selector = CellName String | CellIndex Int deriving (Show)
```

```
type ValueFilter = [CParser]
data CParser = Char Char | Wildcard deriving (Show)

cparse :: CParser -> String -> [String]
cparse (Char c) (c' : cs') | c == c' = [cs']
cparse Wildcard []                   = [[]]
cparse Wildcard cs@(_ : cs')         = cs : cparse Wildcard cs'
cparse _ _                           = []

-- Only implemented for S = string Values
filterValue :: ValueFilter -> Value -> Bool
filterValue ps (S cs) = any null (go ps cs)
  where
    go [] cs       = [cs]
    go (p : ps) cs = concatMap (go ps) (cparse p cs)

select :: Selector -> Row -> Maybe Value
select (CellName s) r                          = lookup s r
select (CellIndex n) r | n > 0 && n <= length r = Just (snd (r !! (n - 1)))
                       | otherwise              = Nothing

dselect :: Selector -> Tdb -> Maybe LValue
dselect (CellName s) tdb                            = lookup s tdb
dselect (CellIndex n) tdb | n > 0 && n <= length tdb = Just (snd (tdb !! (n - 1)))
                          | otherwise               = Nothing

apply :: Filter -> Row -> Bool
apply (s, vf) r = case select s r of
  Nothing -> False
  Just v  -> filterValue vf v

exec :: Query -> Db -> [Row]
exec = (flip . foldl . flip) (filter . apply)

sortdb :: Selector -> Db -> Db
sortdb s db = sortBy (\a b -> compare (select s b) (select s a)) db

-- folds a function over a LValue only if it is a Maybe Double list
mfoldl::(Double->Double->Double)->Double->LValue->Double
mfoldl f acc xs =   foldl (\a x -> case x of
                        Just n -> f a n
                        Nothing -> a
                    ) acc ld
                where ld = case xs of
                    LN ln -> ln
                    LS ls -> []
```