pnm

Claude Heiland-Allen

2012–2015

Contents

1 Codec/PNM.hs

```
{- |
Module      :  Codec.PNM
Copyright   :  (c) Claude Heiland-Allen 2012
License     :  BSD3

Maintainer  :  claude@mathr.co.uk
Stability   :  provisional
Portability :  portable

PNM image format header parsing and pretty printing.

PNM is PBM + PGM + PPM, a family of lowest common denominator image file formats.

References: <http://netpbm.sourceforge.net/doc/pnm.html>.

-}
module Codec.PNM
  ( PNM(..), pnmRasterBytes, pnmPretty, Parse(..), pnmParse
  , pnms, unpnms, onPNMs, onPNMs'
  ) where

import Prelude hiding ((++), (**), concat, splitAt)
import qualified Prelude as P

import Data.ByteString.Lazy (ByteString, uncons, empty, concat, splitAt)
import Data.Int (Int64)
import Codec.PNM.Parse

-- | PNM image headers.
data PNM
  = PBM{ pnmPlain :: Bool, pnmWidth, pnmHeight              :: Integer }
```

```
  | PGM{ pnmPlain :: Bool, pnmWidth, pnmHeight, pnmMaxVal :: Integer }
  | PPM{ pnmPlain :: Bool, pnmWidth, pnmHeight, pnmMaxVal :: Integer }
  deriving (Eq, Ord, Show)

-- | Compute the raster size in bytes for binary PNM images.
pnmRasterBytes :: PNM -> Maybe Integer
pnmRasterBytes pnm
  | pnmPlain pnm = Nothing
  | otherwise = Just $ case pnm of
      PBM _ w h -> ((w + 7) `div` 8) * h
      PGM _ w h m
        | m < 256 -> w * h
        | otherwise -> 2 * w * h
      PPM _ w h m
        | m < 256 -> 3 * w * h
        | otherwise -> 6 * w * h

-- | Pretty-print a PNM image header without any comments.
pnmPretty :: PNM -> ByteString
pnmPretty (PBM True  w h  ) = str $ "P1\n" ** show w ** " " ** show h ** "\n"
pnmPretty (PBM False w h  ) = str $ "P4\n" ** show w ** " " ** show h ** "\n"
pnmPretty (PGM True  w h m) = str $ "P2\n" ** show w ** " " ** show h ** "\n" **↙
    ↳ show m ** "\n"
pnmPretty (PGM False w h m) = str $ "P5\n" ** show w ** " " ** show h ** "\n" **↙
    ↳ show m ** "\n"
pnmPretty (PPM True  w h m) = str $ "P3\n" ** show w ** " " ** show h ** "\n" **↙
    ↳ show m ** "\n"
pnmPretty (PPM False w h m) = str $ "P6\n" ** show w ** " " ** show h ** "\n" **↙
    ↳ show m ** "\n"

(**) :: String -> String -> String
(**) = (P.++)

-- | Parse a PNM image header.
pnmParse :: ByteString -> Parse PNM
pnmParse s = case uncons s of
  Nothing -> Empty
  Just (p, ps)
    | p == 80{-P-} -> case uncons ps of
        Just (d, _)
          | d == d1 -> p1 s
          | d == d2 -> p2 s
          | d == d3 -> p3 s
          | d == d4 -> p4 s
          | d == d5 -> p5 s
          | d == d6 -> p6 s
        _ -> Wrong s
  _ -> Wrong s

p1, p2, p3, p4, p5, p6 :: ByteString -> Parse PNM

(p1, p4) = (pbm "P1" (PBM True), pbm "P4" (PBM False))
  where
    pbm magic ctor s0 = case string (str magic) s0 of
      Empty -> Empty
      Parse () ps s1 -> case number s1 of
        Parse w ws s2 -> case number s2 of
```

```
            Parse h hs s3 -> case oneSpace s3 of
              Parse () ss s4 -> Parse (ctor w h) (ps ++ ws ++ hs ++ ss) s4
              _ -> Wrong s0
            _ -> Wrong s0
          _ -> Wrong s0
        _ -> Wrong s0

  (p2, p3, p5, p6) = (ppm "P2" (PGM True), ppm "P3" (PPM True), ppm "P5" (PGM ↙
      ↳ False), ppm "P6" (PPM False))
    where
      ppm magic ctor s0 = case string (str magic) s0 of
        Empty -> Empty
        Parse () ps s1 -> case number s1 of
          Parse w ws s2 -> case number s2 of
            Parse h hs s3 -> case number s3 of
              Parse m ms s4 | m < 65536 -> case oneSpace s4 of
                Parse () ss s5 -> Parse (ctor w h m) (ps ++ ws ++ hs ++ ms ++ ss) ↙
                    ↳ s5
                _ -> Wrong s0
              _ -> Wrong s0
            _ -> Wrong s0
          _ -> Wrong s0
        _ -> Wrong s0

  -- |  Parse a sequence of binary PNM images.
  --
  --    Malformed input (including huge raster sizes or plain images)
  --    is treated as end-of-image-stream.
  --
  pnms :: ByteString -> [(PNM, ByteString)]
  pnms s = case pnmParse s of
    Parse pnm _ rest
      | Nothing < bytes && bytes <= Just (toInteger (maxBound :: Int64)) ->
          (pnm, raster) : pnms next
      where
        ~(raster, next) = splitAt (fromInteger bytes') rest
        bytes@(~(Just bytes')) = pnmRasterBytes pnm
    _ -> []  -- FIXME what to do about huge rasters or malformed input?

  -- |  Pretty-print a sequence of binary PNM images.
  --
  --    The precondition that the raster is of the correct length is not
  --    checked, so malformed output is possible.
  --
  unpnms :: [(PNM, ByteString)] -> ByteString
  unpnms = concat . map (\ ~(pnm, raster) -> pnmPretty pnm ++ raster)

  -- |  Process a sequence of binary PNM images.
  --
  --    Malformed input (including huge raster sizes or plain images)
  --    is treated as end-of-image-stream.
  --
  --    The precondition that the raster is of the correct length is not
  --    checked, so malformed output is possible.
  --
  --    Header comments are not preserved.
  --
```

```
onPNMs :: (PNM -> ByteString -> (PNM, ByteString)) -> ByteString -> ByteString
onPNMs f = unpnms . map (\ ~(pnm, raster) -> f pnm raster) . pnms

-- |  Process a sequence of binary PNM images.
--
--    Malformed input (including huge raster sizes or plain images)
--    is treated as end-of-image-stream.
--
--    The precondition that the raster is of the correct length is not
--    checked, so malformed output is possible.
--
--    Header comments are preserved.  Assuming well-formed input:
--
--    > onPNMs' (\_ r -> r) = id
--
onPNMs' :: (PNM -> ByteString -> ByteString) -> ByteString -> ByteString
onPNMs' f s = case pnmParse s of
  Parse pnm header rest
    | Nothing < bytes && bytes <= Just (toInteger (maxBound :: Int64)) ->
        header ++ raster' ++ onPNMs' f next
    where
      raster' = f pnm raster
      ~(raster, next) = splitAt (fromInteger bytes') rest
      bytes@(~(Just bytes')) = pnmRasterBytes pnm
  _ -> empty  -- FIXME what to do about huge rasters or malformed input?
```

2 Codec/PNM/Parse.hs

```
{- |
Module      :  Codec.PNM.Parse
Copyright   :  (c) Claude Heiland-Allen 2012
License     :  BSD3

Maintainer  :  claude@mathr.co.uk
Stability   :  provisional
Portability :  portable

Lower-level functions for parsing PNM image format headers.  Most users
shouldn't need to import this module directly.

-}
module Codec.PNM.Parse where

import Prelude hiding ((++), dropWhile)
import Data.ByteString.Lazy (ByteString, append, empty, singleton, uncons, pack,↙
    ↳ unpack)
import Data.Word (Word8)
import Data.Char (chr, ord)

-- |  The result of parsing.
data Parse p
  = Wrong
      { parseMalformed :: ByteString
      }
  | Empty
  | Parse
      { parseResult :: p
```

```
      , parseRawResult :: ByteString
      , parseRemainder :: ByteString
      }
  deriving (Eq, Ord, Show)

-- | The next character.
rawChar :: ByteString -> Parse Word8
rawChar s = case uncons s of
  Nothing -> Empty
  Just (c, cs) -> Parse c (singleton c) cs

-- | The next non-comment character. Comments can occur in the middle
--   of what might be considered tokens.
char :: ByteString -> Parse Word8
char s = case uncons s of
  Nothing -> Empty
  Just (c, cs)
    -- handle comments: "# ... EOL" can occur anywhere, including mid-token.
    | isStartComment c -> case dropWhile (not . isEndComment) rawChar s of
        Wrong _ -> Wrong s
        Empty -> Wrong s
        Parse () ds es -> case uncons es of
            Nothing -> Wrong s
            Just (f, fs)
              | isEndComment f -> case char fs of
                  Wrong _ -> Wrong s
                  Empty -> Wrong s
                  Parse g gs hs -> Parse g (ds ++ singleton f ++ gs) hs
              | otherwise -> Wrong s  -- should never happen
    | otherwise -> Parse c (singleton c) cs

-- | Drop input while a predicate holds.
dropWhile :: (Word8 -> Bool) -> (ByteString -> Parse Word8) -> ByteString -> ↙
    ↳ Parse ()
dropWhile p get s = case get s of
  Wrong _ -> Wrong s
  Empty -> Empty
  Parse c cs ds
    | p c -> case dropWhile p get ds of
        Wrong _ -> Wrong s
        Empty -> Parse () cs ds
        Parse () es fs -> Parse () (cs ++ es) fs
    | otherwise -> Parse () empty s

-- | Take input until a predicate holds.
takeUntil :: (Word8 -> Bool) -> (ByteString -> Parse Word8) -> ByteString -> ↙
    ↳ Parse ByteString
takeUntil p get s = case get s of
  Wrong _ -> Wrong s
  Empty -> Empty
  Parse c cs ds
    | p c -> Parse empty empty s
    | otherwise -> case takeUntil p get ds of
        Wrong _ -> Wrong s
        Empty -> Parse (singleton c) cs ds
        Parse e es fs -> Parse (singleton c ++ e) (cs ++ es) fs
```

```
-- |  Parse a token.
token :: ByteString -> Parse ByteString
token s = case dropWhile isSpace char s of
  Wrong _ -> Wrong s
  Empty -> Empty
  Parse () cs ds -> case takeUntil isSpace char ds of
    Wrong _ -> Wrong s
    Empty -> Wrong s
    Parse t ts rs -> Parse t (cs ++ ts) rs

-- |  Parse a fixed string.
string :: ByteString -> ByteString -> Parse ()
string match s = case token s of
  Empty -> Empty
  Parse t ts rs | t == match -> Parse () ts rs
  _ -> Wrong s

-- |  Parse a positive decimal number.
number :: ByteString -> Parse Integer
number s = case token s of
  Parse t ts rs -> case decimal t of
    Just n -> Parse n ts rs
    _ -> Wrong s
  _ -> Wrong s

-- |  Number conversion.
decimal :: ByteString -> Maybe Integer
decimal s = case unpack s of
  ts@(t:_)
    | all isDigit ts && t /= d0 -> case reads (map (chr . fromEnum) ts) of
        [(n, "")] -> Just n
        _ -> Nothing  -- should never happen?
  _ -> Nothing

-- |  Parse a single space.
oneSpace :: ByteString -> Parse ()
oneSpace s = case char s of
  Parse c cs rs | isSpace c -> Parse () cs rs
  _ -> Wrong s

-- |  Convert from a string.  Crashes hard on non-ASCII input.
str :: String -> ByteString
str = pack . map (toEnum . ord) -- toEnum crashes on out of bounds

-- | Character classes.
isSpace, isDigit, isStartComment, isEndComment :: Word8 -> Bool
isSpace c = c `elem` [ht, lf, vt, ff, cr, space]
isDigit c = c `elem` [d0, d1, d2, d3, d4, d5, d6, d7, d8, d9]
isStartComment c = c == hash
isEndComment c = c == lf || c == cr

-- |  White space characters.
ht, lf, vt, ff, cr, space :: Word8
ht    = 9
lf    = 10
vt    = 11
ff    = 12
```

```
cr    = 13
space = 32

-- |  Comment start character.
hash :: Word8
hash  = 35

-- |  Decimal digit characters.
d0, d1, d2, d3, d4, d5, d6, d7, d8, d9 :: Word8
d0 = 48
d1 = 49
d2 = 50
d3 = 51
d4 = 52
d5 = 53
d6 = 54
d7 = 55
d8 = 56
d9 = 57

-- |  Alias for 'append'.
(++) :: ByteString -> ByteString -> ByteString
(++) = append
```

3 .gitignore

```
dist
```

4 LICENSE

```
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

5 pnm.cabal

```
name:                pnm
version:             0.1.0.0
synopsis:            PNM image format header parsing and pretty printing
description:
  PNM image format header parsing and pretty printing. PNM is PBM + PGM + PPM,
  a family of lowest common denominator image file formats. This library makes
  no attempt to interpret raster data, instead providing a lazy ByteString.

license:             BSD3
license-file:        LICENSE
author:              Claude Heiland-Allen
maintainer:          claude@mathr.co.uk
category:            Graphics
build-type:          Simple
cabal-version:       >=1.8

library
  exposed-modules:     Codec.PNM, Codec.PNM.Parse
  build-depends:       base < 5, bytestring

source-repository head
  type:                git
  location:            http://code.mathr.co.uk/pnm.git

source-repository this
  type:                git
  location:            http://code.mathr.co.uk/pnm.git
  tag:                 v0.1.0.0
```

6 pnminfo.hs

```
import Codec.PNM (pnms)
import Codec.PNM.Parse (str)
import qualified Data.ByteString.Lazy as BS
main = BS.interact (BS.concat . map (str . (++"\n") . show . fst) . pnms)
```

7 pnmstrip.hs

```
import Codec.PNM (onPNMs)
import qualified Data.ByteString.Lazy as BS
main = BS.interact (onPNMs (curry id))
```

8 Setup.hs

```
import Distribution.Simple
main = defaultMain
```