gpcsets:
Pitch Class Sets for Haskell
Library Documentation

Bruce H. McCosar

May 14, 2009

Contents

Chapter 1

Data.PcSets

1.1 Introduction

1.1.1 The Module Export List

```
{-|
  The basic module for working with Pitch Class Sets of all kinds,
  including Tone Rows.  The broadest datatypes ('GenSet' and 'GenRow')
  can model any equal temperament system; the standard datatypes
  ('StdSet' and 'StdRow') model /12 Tone Equal Temperament/ (12-TET).
-}
module Data.PcSets
  (
    -- * Classes
      PcSet (modulus, elements, pMap)
    , Selective (complement)
    , Inclusive (reconcile)
    -- * Types
    -- ** Selective (Sets)
    , GenSet
    , StdSet
    -- ** Inclusive (Rows)
    , GenRow
    , StdRow
    -- * Constructors
    -- ** Selective (Sets)
    , genset
    , stdset
    -- ** Inclusive (Rows)
    , genrow
    , stdrow
    -- * General Operations (All Sets)
```

```
    -- ** Transformations
    , transpose
    , invert
    , invertXY
    , zero
    -- ** Permutations
    , retrograde
    , rotate
    -- * Selective Set Operations
    -- ** Systematically Equivalent Forms
    , sort
    , normal
    , reduced
    , prime
    -- ** Scalar Quantities
    , cardinality
    , binaryValue
    -- ** Vector Quantities
    , avec
    , cvec
    , ivec
    -- * Inclusive Set (Tone Row) Operations
    , rowP
    , rowR
    , rowI
    , rowRI
  )
where
```

1.1.2 The Module Import List

```
import qualified Data.List (nub, sort, sortBy, elemIndices)
```

1.2 Classes

1.2.1 PcSet

```
{-|
  The broadest class of Pitch Class Set. All members of this class
  have a 'modulus' which restricts their 'elements' in some way. They
  also have 'pMap', a method for lifting integer list functions to act
  on set elements.  The 'modulus' corresponds to the underlying system
  of equivalent pitch classes, for example, 12-TET = modulus 12.
-}
class PcSet a where
```

```
  -- | Determines the range of possible 'elements' of the set,
  --   from 0 to (m-1).  If m = 0, the set can only be empty.
  modulus  :: a -> Int
  -- | Returns the elements of the set as a list.
  elements :: a -> [Int]
  -- | Maps an integer list function across the members of the set,
  --   and returns the results in a new set of the same type.
  pMap     :: ([Int] -> [Int]) -> a -> a
```

1.2.2 Selective PcSets (Pitch Class Sets)

```
{-|
  Selective Pitch Class Sets can have 'elements' in a range of values
  permitted by their 'modulus'. They can have as few as 0 (the empty
  set) or as many as all. The set 'complement' operation only makes
  sense for 'Selective' sets.
-}
class PcSet a => Selective a where
  -- | Returns a new PcSet which is the complement of the original:
  --   it contains all the 'elements' which the original does not.
  complement :: a -> a
```

1.2.3 Inclusive PcSets (Tone Rows)

```
{-|
  Inclusive Pitch Class Sets, or Tone Rows, have all the possible
  'elements' permitted by their 'modulus'. The most important
  characteristic of a Tone Row is not its 'elements', but the
  /ordering/ of its 'elements'.
-}
class PcSet a => Inclusive a where
  -- | Transposes the 'elements' of a Tone Row so that the first
  --   element is /n/.
  reconcile :: Int -> a -> a
  reconcile n ps = transpose r ps
    where
      firstElement = head . elements $ ps
      r = n - firstElement
```

1.3 Types

1.3.1 GenSet: General Pitch Class Sets

```
{-|
  General Pitch Class Set.  This represents a Pitch Class Set that
  can have a 'modulus' of any positive integer value, representing
  the number of equivalent pitch classes in a given system; for
  example, 19-TET would be a modulus 19 set.  The members of a the
  set can be as few as zero and as many as all possible values.
-}
data GenSet = GenSet Int [Int]
  deriving (Eq,Ord,Show)
```

text

```
instance PcSet GenSet where
  modulus  (GenSet m _ ) = m
  elements (GenSet _ es) = es
  pMap f   (GenSet m es) = genset m . f $ es
```

text

```
instance Selective GenSet where
  complement (GenSet 0 _ ) = GenSet 0 []
  complement (GenSet m es) = GenSet m cs
    where cs = filter (`notElem` es) [0..(m-1)]
```

1.3.2 StdSet: Standard Pitch Class Sets

```
{-|
  Standard Pitch Class Set.  This represents the traditional
  definition of a pitch class set, based on 12-TET, with the
  pitch classes numbered C = 0, C#/Db = 1, D = 2, and so on
  up to B = 11.  This set can have anywhere from zero to 12
  members (the empty set vs. the chromatic scale).
-}
data StdSet = StdSet [Int]
  deriving (Eq,Ord,Show)
```

text

```
instance PcSet StdSet where
  modulus  (StdSet _ ) = 12
  elements (StdSet es) = es
  pMap f   (StdSet es) = stdset . f $ es
```

text

```
instance Selective StdSet where
  complement (StdSet es) = StdSet cs
    where cs = filter (`notElem` es) [0..11]
```

1.3.3 GenRow: General Tone Rows

```
{-|
  General Tone Row.  A /Tone Row/ is a collection of all possible
  Pitch Class Set 'elements' within a given 'modulus'.  Since it
  contains all elements, the significant information in this type
  of set is the ordering of the 'elements'.  This set always has
  a length equal to its 'modulus'.
-}
data GenRow = GenRow [Int]
  deriving (Eq,Ord,Show)
```

text

```
instance PcSet GenRow where
  modulus  (GenRow es) = length es
  elements (GenRow es) = es
  pMap f   (GenRow es) = genrow (length es) . f $ es
```

text

```
instance Inclusive GenRow
```

1.3.4 StdRow: Standard Tone Rows

```
{-|
  Standard Tone Row.  This is the traditional Tone Row, a collection
  of all the elements @[0..11]@, based on 12-TET.  As with 'GenRow',
  the most significant information in this type of set is the ordering
  of the elements.  Since this is always a complete set, this set
  always has a length of 12.
-}
data StdRow = StdRow [Int]
  deriving (Eq,Ord,Show)
```

text

```
instance PcSet StdRow where
  modulus  (StdRow _ ) = 12
  elements (StdRow es) = es
  pMap f   (StdRow es) = stdrow . f $ es
```

text

```
instance Inclusive StdRow
```

1.4 Constructors

1.4.1 genset

```
{-|
  Constructor for General Pitch Class Sets.  This constructor accepts
  any @Int@ value for 'modulus', and any @[Int]@ values for an input
  list.  Zero 'modulus' always returns an empty set; a negative 'modulus'
  is always taken as positive (since the number represent the /absolute/
  size of the equivalence class).
-}
genset :: Int -> [Int] -> GenSet
genset 0 _ = GenSet 0 []
genset m_in es = GenSet m (f es)
  where
    m = abs m_in
    f = Data.List.nub . map (`mod` m)
```

1.4.2 stdset

```
{-|
  Constructor for Standard Pitch Class Sets.  This constructor accepts
  any @[Int]@ values for elements.  The 'modulus' is always 12 (12-TET).
-}
stdset :: [Int] -> StdSet
stdset es = StdSet ps
  where ps = elements $ genset 12 es
```

1.4.3 genrow

```
{-|
  Constructor for General Tone Rows.  This constructor accepts any @Int@
  value for 'modulus', and any @[Int]@ values for an input list.  Zero
  'modulus' always returns an empty set; a negative 'modulus' is always
  taken as positive (see 'GenSet').  If the input list of 'elements' is
  incomplete, the remaining 'elements' are filled in at the end, in order.
-}
genrow :: Int -> [Int] -> GenRow
genrow m es = GenRow (os ++ cs)
  where
    ps = genset m es
    os = elements ps
    cs = elements $ complement ps
```

1.4.4 stdrow

```
{-|
  Constructor for Standard Tone Rows.  This constructor accepts any @[Int]@
  values for an input list.  The 'modulus' is always 12 (12-TET).  If the
  input list of 'elements' is incomplete, the remaining 'elements' are filled
  in at the end, in order.
-}
stdrow :: [Int] -> StdRow
stdrow es = StdRow ts
  where ts = elements $ genrow 12 es
```

1.5 General Operations (All Sets)

1.5.1 Transformations

1.5.1.1 transpose

```
-- | Returns a new 'PcSet' which is the original transposed by /n/.
transpose :: PcSet a => Int -> a -> a
transpose = pMap . map . (+)
```

1.5.1.2 invert

```
{-|
  Returns a new 'PcSet' which is the /standard inverse/ of the original,
  that is, about an axis containing pitch class 0.
-}
invert :: PcSet a => a -> a
invert ps = pMap (map (m -)) ps
  where m = modulus ps
```

1.5.1.3 invertXY

```
{-|
  Inversion around an axis specified by pitch classes /x/ and /y/.
  This inverts the set in such a way that /x/ becomes /y/ and /y/
  becomes /x/.
-}
invertXY :: PcSet a => Int -> Int -> a -> a
invertXY x y = transpose (x + y) . invert
```

1.5.1.4 zero

```
{-|
  Returns a new 'PcSet' in which the elements have been transposed
  so that the first element is zero.
-}
zero :: PcSet a => a -> a
zero ps = transpose (-n) ps
  where n = head . elements $ ps
```

1.5.2 Permutations

1.5.2.1 retrograde

```
-- | Returns a new 'PcSet' with the elements of the original reversed.
retrograde :: PcSet a => a -> a
retrograde = pMap reverse
```

1.5.2.2 rotate

```
-- | Returns a new 'PcSet' with the elements shifted /n/ places to the left.
rotate :: PcSet a => Int -> a -> a
rotate n ps = pMap nShift ps
  where
    nShift = take sameLength . drop offset . cycle
    sameLength = (length . elements) ps
    offset = n `mod` sameLength
```

1.6 Selective Set Operations

1.6.1 Systematically Equivalent Forms

1.6.1.1 sort

```
{-|
  Returns a 'Selective' 'PcSet' in which the elements of the original
  have been sorted in ascending order. (Note this is restricted to Sets,
  as sorting a Tone Row produces only an ascending chromatic scale.)
-}
sort :: (PcSet a, Selective a) => a -> a
sort = pMap Data.List.sort
```

1.6.1.2 normal

```
{-|
  Returns a 'Selective' 'PcSet' in which the elements of the original have
  been put into /normal form/.  This can be defined as an ascending order
  in which the elements fit into the smallest overall interval. In the event
  of a tie, the arrangement with the closest leftward packing is chosen.
-}
normal :: (PcSet a, Selective a) => a -> a
normal = nform . bestPack . pcsArrangements
```

1.6.1.3 reduced

```
{-|
  Returns a 'Selective' 'PcSet' in which the elements of the original
  have been put into /reduced form/.  This can be thought of as the
  'normal' form, transposed so that the first element starts on 'zero'.
-}
reduced :: (PcSet a, Selective a) => a -> a
reduced = rform . bestPack . pcsArrangements
```

1.6.1.4 prime

```
{-|
  Returns a 'Selective' 'PcSet' in which the elements of the original
  have been put into /prime form/.  A prime form is able to generate
  all the members of its set family through the some combination of the
  operations 'transpose', 'invert', and simple permutation.
-}
prime :: (PcSet a, Selective a) => a -> a
prime ps = if i_val < o_val then inversion else original
  where
    original = reduced ps
    inversion = reduced $ invert ps
    o_val = binaryValue original
    i_val = binaryValue inversion
```

1.6.2 Scalar Quantities

1.6.2.1 cardinality

```
-- | Returns the number of elements in a 'Selective' 'PcSet'.
cardinality :: (PcSet a, Selective a) => a -> Int
cardinality = length . elements
```

1.6.2.2 binaryValue

```
{-|
  Binary Value. For a given 'Selective' 'PcSet', this returns a
  /unique/ number relating to the elements of the set -- a measure
  of the "leftward packing" of the sorted set (overall closeness
  of each element to zero).
-}
binaryValue :: (PcSet a, Selective a) => a -> Integer
binaryValue = sum . map (2 ^) . elements
```

1.6.3 Vector Quantities

1.6.3.1 avec

```
{-|
  Ascending Vector. If the elements of a 'Selective' 'PcSet' are
  taken to be in strictly ascending order, the ascending vector is
  the interval difference between each element.
-}
avec :: (PcSet a, Selective a) => a -> [Int]
avec ps = map (`mod` m) $ zipWith (-) rs os
  where
    m = modulus ps
    os = elements ps
    rs = elements . rotate 1 $ ps
```

1.6.3.2 cvec

```
{-|
  Common Tone Vector: finds the number of common tones for each possible
  value of /n/ in the operation 'transpose' /n/ . 'invert'. Returns a list
  where element 0 is the number of common tones with /n/=0, element 1 is
  with /n/=1, and so on.
-}
cvec :: (PcSet a, Selective a) => a -> [Int]
cvec ps = count . concatMap f $ es
  where
    m = modulus ps
    es = elements ps
    count cs = map (\n ->
      length (Data.List.elemIndices n cs)) [0..(m-1)]
    f x = map (\y -> (x + y) `mod` m) es
```

1.6.3.3 ivec

```
{-|
  Interval Vector.  Each element of the interval vector represents
  the number of intervals in the set for that particular interval
  class.  Element 0 measures the number of 1-interval leaps;
  element 1 measures the number of 2-interval leaps, and so on,
  up to half of the modulus /m/.
-}
ivec :: (PcSet a, Selective a) => a -> [Int]
ivec ps = if m == 0 then []
    else pivotguard . spacefold . count . intervals . elements $ ps
  where
    m = modulus ps
    -- pivotguard: compensates for even lists, where the largest possible
    -- interval is equal to its inverse (and thereby counted twice, here).
    pivotguard es = if odd m then es
      else init es ++ [last es `div` 2]
    -- spacefold: wraps interval list to interval classes
    spacefold = take (m `div` 2) . flipSum
    flipSum es = zipWith (+) es (reverse es)
    -- count: counts each occurrence of each possible diff
    count ivs = map (g ivs) [1..(m-1)]
    g ivs n = length (Data.List.elemIndices n ivs)
    -- intervals: returns recursive list of diffs
    intervals [] = []
    intervals (e:es) = diffs e es ++ intervals es
    -- diffs: interval difference between pitches
    diffs = map . f
    f a b = (b - a) `mod` m
```

1.7 Inclusive Set (Tone Row) Operations

1.7.1 Permutation-Transformations

1.7.1.1 rowP

```
{-|
  Returns a new Tone Row in which the elements are /Prograde/
  (in their original order) and transposed so that the first
  element is /n/.
-}
rowP :: (PcSet a, Inclusive a) => Int -> a -> a
rowP = reconcile
```

1.7.1.2 rowR

```
{-|
  Returns a new Tone Row in which the elements are /Retrograde/
  (reversed compared to their original order) and transposed so
  that the first element is /n/.
-}
rowR :: (PcSet a, Inclusive a) => Int -> a -> a
rowR = (. retrograde) . reconcile
```

1.7.1.3 rowI

```
{-|
  Returns a new Tone Row in which the elements have been /Inverted/
  (see 'invert') and transposed so that the first element is /n/.
-}
rowI :: (PcSet a, Inclusive a) => Int -> a -> a
rowI = (. invert) . reconcile
```

1.7.1.4 rowRI

```
{-|
  Returns a new Tone Row in which the elements are both /Retrograde/
  and /Inverted/, and transposed so that the first element is /n/.
-}
rowRI :: (PcSet a, Inclusive a) => Int -> a -> a
rowRI = (. (invert . retrograde)) . reconcile
```

1.8 Not Exported

1.8.1 Related to Normal, Reduced, and Prime

```
data (PcSet a, Selective a) => Candidate a = Candidate
  {
    idx :: Integer,
    nform :: a,
    rform :: a
  }
```

```
interview :: (PcSet a, Selective a) => a -> Candidate a
interview ps = Candidate
  {
    idx = binaryValue zs,
```

```
    nform = ps,
    rform = zs
  }
  where zs = zero ps
```

```
sortFunction :: (PcSet a, Selective a) =>
  Candidate a -> Candidate a -> Ordering
sortFunction a b = compare (idx a) (idx b)
```

```
bestPack :: (PcSet a, Selective a) => [a] -> Candidate a
bestPack arrs = head (Data.List.sortBy sortFunction candidates)
  where candidates = [interview ps | ps <- arrs]
```

```
pcsArrangements :: (PcSet a, Selective a) => a -> [a]
pcsArrangements ps = if n == 0
  then [ps] -- only one possible arrangement for nothing.
  else take n $ iterate f (sort ps)
  where
    n = cardinality ps
    f = rotate 1
```

Chapter 2

Data.PcSets.Svg

2.1 Introduction

2.1.1 The Module Export List

```
{-|
  This module produces simple representations of Pitch Class Sets
  suitable for use in Scalable Vector Graphics.  By default it
  does not generate the files -- instead, it generates a printable
  string, which can be captured to standard output or directed to
  a file at your discretion.
-}
module Data.PcSets.Svg
  (
    -- * Simple Usage
      pcSvg
    , pcSvgAx
    -- * Advanced Usage
    , pcSvg'
    , pcSvgAx'
    -- * Rendering Style
    , Rendering (Rendering, pxSize, lnColor, psColor, csColor,
                 axColor, relMain, relElem, relAxis)
    -- ** Default Rendering Values
    , stdRen
  )
where
```

2.1.2 The Module Import List

```
import qualified Data.PcSets as P
```

2.2 Simple Usage

```
-- | The basic idea: generate SVG data for an input pitch class set.
pcSvg :: (P.PcSet a) => a -> String
pcSvg = pcSvg' stdRen
```

```
-- | Same as 'pcSvg', but includes an /invertXY/ style axis.
pcSvgAx :: (P.PcSet a) => a -> (Int,Int) -> String
pcSvgAx = pcSvgAx' stdRen
```

2.3 Advanced Usage

```
-- | Same as 'pcSvg' but allows a custom 'Rendering'.
pcSvg' :: (P.PcSet a) => Rendering -> a -> String
pcSvg' ren ps = svgHeader ++ show (toSvg ren ps parts)
  where parts = [psFrame,psCircle]
```

```
-- | Same as 'pcSvgAx', but allows a custom 'Rendering'.
pcSvgAx' :: (P.PcSet a) => Rendering -> a -> (Int,Int) -> String
pcSvgAx' ren ps invAx = svgHeader ++ show (toSvg ren ps parts)
  where parts = [psFrame, psAxis invAx ps, psCircle]
```

2.3.1 Rendering Style

2.3.1.1 Rendering Data Structure

```
-- | Stores the rendering information for the SVG file.
data Rendering = Rendering
  {
    pxSize  :: Int,    -- ^ sets the (square) image dimensions
    lnColor :: String, -- ^ line color for the main structures
    psColor :: String, -- ^ pitch class set color
    csColor :: String, -- ^ complementary set color
    axColor :: String, -- ^ axis color
    relMain :: Float,  -- ^ proportion of main circle compared to image
    relElem :: Float,  -- ^ proportion of elements compared to main circle
    relAxis :: Float   -- ^ proportion of axis (if any) compared to image
  }
```

2.3.1.2 Default Rendering

```
{-|
  The Standard 'Rendering' is a 500x500 image using black lines, with
  elements of the set in red, the complement in black, and any axis in
  blue. The pitch class set circle is 80% of the frame, each element
  is 10% of the main circle's size, and any axis is 95% frame size.
-}
stdRen :: Rendering
stdRen = Rendering
  {
    pxSize = 500,
    lnColor = "black",
    psColor = "red",
    csColor = "black",
    axColor = "blue",
    relMain = 0.80,
    relElem = 0.10,
    relAxis = 0.95
  }
```

2.4 Not Exported

2.4.1 XML functions

2.4.1.1 Attributes

```
data Attr = Attr String String

instance Show Attr where
  show (Attr n v) = n ++ "=\"" ++ v ++ "\""
```

```
attrs :: [Attr] -> String
attrs as = unwords [show a | a <- as]
```

```
nattr :: String -> Int -> Attr
nattr s = Attr s . show -- 'numerical attributes'
```

2.4.1.2 Tags

```
data Tag = Tag String [Attr]

instance Show Tag where
  show (Tag n as) = "<" ++ n ++ " " ++ attrs as ++ "/>"
```

2.4.1.3 Parent Tags

```
data PTag = PTag String [Attr] [Tag]

instance Show PTag where
  show (PTag n as ts) = "<" ++ n ++ " " ++ attrs as
      ++ ">\n" ++ tags ++ "</" ++ n ++ ">"
    where tags = unlines [show t | t <- ts]
```

2.4.2 SVG functions

2.4.2.1 SVG Elements

```
circle :: (Int,Int) -> Int -> String -> Int -> String -> Tag
circle (cx,cy) r s sw f =
  Tag "circle" [nattr "cx" cx, nattr "cy" cy, nattr "r" r,
    Attr "stroke" s, nattr "stroke-width" sw, Attr "fill" f]
```

```
line :: (Int,Int) -> (Int,Int) -> String -> Int -> String -> Tag
line (x1,y1) (x2,y2) s sw sd =
  Tag "line" [nattr "x1" x1, nattr "y1" y1,
              nattr "x2" x2, nattr "y2" y2,
              Attr "stroke" s, nattr "stroke-width" sw,
              Attr "stroke-dasharray" sd]
```

```
rect :: (Int,Int) -> (Int,Int) -> String -> Int -> String -> Tag
rect (x,y) (w,h) s sw f =
  Tag "rect" [nattr "x" x, nattr "y" y,
              nattr "width" w, nattr "height" h,
              Attr "stroke" s, nattr "stroke-width" sw,
              Attr "fill" f]
```

2.4.2.2 SVG Toplevel

```
svgHeader :: String
svgHeader = "<?xml version=\"1.0\" standalone=\"no\"?>\n" ++
            "<!DOCTYPE svg PUBLIC \"-//W3C//DTD SVG 1.1//EN\" " ++
            "\"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd\">\n"
```

```
svg :: (Int,Int) -> [Tag] -> PTag
svg (w,h) ts =
  PTag "svg" stdatt ts
    where
      stdatt = [nattr "width" w,
                nattr "height" h,
```

```
                Attr "version" "1.1",
                Attr "xmlns" "http://www.w3.org/2000/svg"]
```

2.4.3 Convenient Rendering Data Shortcuts

2.4.3.1 Centering

```
ctr :: Rendering -> Int
ctr ren = pxSize ren `div` 2
```

```
ctrxy :: Rendering -> (Int,Int)
ctrxy ren = (ctr ren, ctr ren)
```

2.4.3.2 Framing

```
frameSize :: Rendering -> (Int,Int)
frameSize ren = (p,p)
  where p = pxSize ren
```

2.4.3.3 Radii

Main Circle:

```
mainRad :: Rendering -> Float
mainRad ren = relMain ren * fromIntegral (pxSize ren) / 2
```

Inversion Axes:

```
axisRad :: Rendering -> Float
axisRad ren = relAxis ren * fromIntegral (pxSize ren) / 2
```

Pitch Class Elements:

```
elemRad :: Rendering -> Float
elemRad ren = relElem ren * mainRad ren
```

2.4.4 Element Placement

```
phase :: Int -> Int -> Float
phase t m = 2 * pi * fromIntegral t / fromIntegral m
```

```
pos :: Int -> Int -> Rendering -> (Rendering -> Float) -> (Int,Int)
pos t m ren radf = (x,y)
  where
    r = radf ren
    x = ctr ren + round (r * sin (phase t m))
    y = ctr ren - round (r * cos (phase t m))
```

2.4.5 Builder Functions

```
psFrame :: Rendering -> Tag
psFrame ren = rect (1,1) (frameSize ren) "none" 1 "none"
```

```
psCircle :: Rendering -> Tag
psCircle ren = circle (ctrxy ren) (round r) (lnColor ren) 2 "none"
  where r = mainRad ren
```

```
psElements :: (P.PcSet a) => a -> Rendering -> [Tag]
psElements ps ren = if m == 0 then [] else map f [0..(m-1)]
  where
    m = P.modulus ps
    es = P.elements ps
    f t = circle (p t) (round r) (lnColor ren) 1 (onOff t ren)
    p t = pos t m ren mainRad
    r = elemRad ren
    onOff t = if t `elem` es then psColor else csColor
```

```
psAxis :: (P.PcSet a) => (Int,Int) -> a -> Rendering -> Tag
psAxis (x,y) ps ren =
    line (p t1) (p t2) (axColor ren) 1 "9, 3, 3, 3"
  where
    m = P.modulus ps
    p t = pos t (m * 2) ren axisRad
    t1 = x + y
    t2 = x + y + m
```

This is the big one.

```
toSvg :: (P.PcSet a) => Rendering -> a -> [(Rendering -> Tag)] -> PTag
toSvg ren ps parts = svg (frameSize ren) tags
  where
    tags = map ($ ren) parts ++ psElements ps ren
```

Chapter 3

Data.PcSets.Compact

3.1 Introduction

Important Note. My philosophy in designing these modules has been to avoid pointless errors and exceptions, that is, to create acceptable, sane default operations. I assume if someone is going to try to run the text of Tolstoy's "War and Peace" through the String-to-Pitch-Class-Set routines in this module, they are prepared to accept the default behavior, which is to silently ignore much of the input. This makes much more sense than attempting to handle every possible input string (and provide a suitable error message for each case).

3.1.1 The Module Export List

```
{-|
  This module translates Pitch Class Sets to and from /Compact Format/.
  In Compact Format, data such as StdSet [0,4,7,11] could be represented
  by the string 047B, which uses a single alphanumeric character for each
  pitch class element.

  Limitations: this module is only usable for pitch class sets of modulus
  36 or below. Beyond that, it's not really certain that a compact format
  would be of any practical use.
-}
module Data.PcSets.Compact
  (
    -- * Constructors: Compact Format to PcSet
      toGenSet
    , toStdSet
    , toStdSet'
    , toGenRow
    , toStdRow
```

```
    , toStdRow'
    -- * Abbreviators: PcSet to Compact Format
    , compact
    , compact'
  )
where
```

3.1.2 The Module Import List

```
import qualified Data.PcSets as P
```

3.2 Constructors: Compact Format to PcSet

3.2.1 Pitch Class Sets

3.2.1.1 toGenSet

```
{-|
  Creates a new General Pitch Class Set of modulus /n/. Alphanumeric
  character values 0–9 and A–Z represent the numbers 0 to 36. Other
  inputs, including whitespace, are ignored.
-}
toGenSet :: Int -> String -> P.GenSet
toGenSet n = P.genset n . trBase36
```

3.2.1.2 toStdSet

```
{-|
  Creates a new Standard (modulus 12) Pitch Class Set. Here, input
  characters 0–9 count as their decimal equivalents; the letter /A/
  stands for 10, and the letter /B/ stands for 11. Other inputs,
  including whitespace, are ignored.
-}
toStdSet :: String -> P.StdSet
toStdSet = P.stdset . trBase36
```

3.2.1.3 toStdSet'

```
{-|
  Creates a new Standard (modulus 12) Pitch Class Set, using an alternative
  duodecimal format. Here, input characters 0–9 count as their decimal
```

```
   equivalents; the letter /T/ stands for 10, and the letter /E/ stands for
   11. Other inputs, including whitespace, are ignored.
-}
toStdSet' :: String -> P.StdSet
toStdSet' = P.stdset . trBase12
```

3.2.2 Tone Rows

3.2.2.1 toGenRow

```
{-|
   Creates a new General Tone Row of modulus /n/. Alphanumeric character
   values 0–9 and A–Z represent the numbers 0 to 36. Other inputs, including
   whitespace, are ignored. Since Tone Rows must contain all possible
   elements, an incomplete entry list will result in a new row with the
   missing tones added at the end.
-}
toGenRow :: Int -> String -> P.GenRow
toGenRow n = P.genrow n . trBase36
```

3.2.2.2 toStdRow

```
{-|
   Creates a new Standard (modulus 12) Tone Row. Here, input characters 0–9
   count as their decimal equivalents; the letter /A/ stands for 10, and
   the letter /B/ stands for 11. Other inputs, including whitespace, are
   ignored. (Also, see notes for 'toGenRow'.)
-}
toStdRow :: String -> P.StdRow
toStdRow = P.stdrow . trBase36
```

3.2.2.3 toStdRow'

```
{-|
   Creates a new Standard (modulus 12) Tone Row, using an alternative
   duodecimal format. Here, input characters 0–9 count as their decimal
   equivalents; the letter /T/ stands for 10, and the letter /E/ stands for
   11. Other inputs, including whitespace, are ignored. (Also, see notes for
   'toGenRow'.)
-}
toStdRow' :: String -> P.StdRow
toStdRow' = P.stdrow . trBase12
```

3.3 Abbreviators: PcSet to Compact Format

3.3.1 General Case

```
{-|
  Translates a Pitch Class Set or Tone Row to Compact Format. Values from
  0-9 are translated as the characters 0-9; values from 10 to 35 are
  translated as charaters A-Z. Values which are out of the representable
  range are ignored, therefore this function is not suitable for sets of
  modulus 37 or greater.
-}
compact :: P.PcSet a => a -> String
compact = filter (/= '#') . map f . P.elements
  where f n
          | 0 <= n && n <= 9 = toEnum (n + 48)
          | 9 < n && n < 37  = toEnum (n + 55)
          | otherwise        = '#' -- ignore out of range
```

3.3.2 Specialized Standard

```
{-|
  This function is identical to 'compact', except that Standard (modulus
  12) sets and rows are rendered using 'T' for 10 and 'E' for 11.
-}
compact' :: P.PcSet a => a -> String
compact' ps = if P.modulus ps /= 12 then compact ps
    else filter (/= '#') . map f . P.elements $ ps
  where f n
          | 0 <= n && n <= 9 = toEnum (n + 48)
          | n == 10          = 'T'
          | n == 11          = 'E'
          | otherwise        = '#' -- ignore out of range
```

3.4 Not Exported

3.4.1 Alphanumeric Translators

3.4.1.1 Base 36

```
trBase36 :: String -> [Int]
trBase36 = filter (>= 0) . map f
  where f c
          | '0' <= c && c <= '9' = fromEnum c - 48
          | 'A' <= c && c <= 'Z' = fromEnum c - 55
```

```
          | otherwise              = -1 -- ignore nonsense
```

3.4.1.2 Duodecimal

```
trBase12 :: String -> [Int]
trBase12 = filter (>= 0) . map f
  where f c
          | '0' <= c && c <= '9' = fromEnum c - 48
          | c == 'T'             = 10
          | c == 'E'             = 11
          | otherwise            = -1 -- ignore nonsense
```

Chapter 4

Data.PcSets.Notes

text

```
module Data.PcSets.Notes where
```

text

```
test :: Int
test = 0
```

final

Chapter 5

Data.PcSets.Catalog

text

```
module Data.PcSets.Catalog where
```

text

```
test :: Int
test = 0
```

final