package com.twitter.tweetypie.tweettext

import scala.collection.immutable

/\*\*

\* An Offset is a typed index into a String.

\*/

trait Offset[T] extends Ordering[T] {

def toInt(t: T): Int

def count(text: String, start: Offset.CodeUnit, end: Offset.CodeUnit): T

def compare(t1: T, t2: T): Int = toInt(t1).compare(toInt(t2))

def length(input: String): T = count(input, Offset.CodeUnit(0), Offset.CodeUnit.length(input))

}

object Offset {

/\*\*

\* UTF-16 code unit offsets are the native offsets for Java/Scala

\* Strings.

\*/

case class CodeUnit(toInt: Int) extends AnyVal with Ordered[CodeUnit] {

def compare(other: CodeUnit): Int = toInt.compare(other.toInt)

def +(other: CodeUnit) = CodeUnit(toInt + other.toInt)

def -(other: CodeUnit) = CodeUnit(toInt - other.toInt)

def min(other: CodeUnit): CodeUnit = if (toInt < other.toInt) this else other

def max(other: CodeUnit): CodeUnit = if (toInt > other.toInt) this else other

def incr: CodeUnit = CodeUnit(toInt + 1)

def decr: CodeUnit = CodeUnit(toInt - 1)

def until(end: CodeUnit): immutable.IndexedSeq[CodeUnit] =

toInt.until(end.toInt).map(CodeUnit(\_))

/\*\*

\* Converts this `CodeUnit` to the equivalent `CodePoint` within the

\* given text.

\*/

def toCodePoint(text: String): CodePoint =

CodePoint(text.codePointCount(0, toInt))

def offsetByCodePoints(text: String, codePoints: CodePoint): CodeUnit =

CodeUnit(text.offsetByCodePoints(toInt, codePoints.toInt))

}

implicit object CodeUnit extends Offset[CodeUnit] {

def toInt(u: CodeUnit): Int = u.toInt

override def length(text: String): CodeUnit = CodeUnit(text.length)

def count(text: String, start: CodeUnit, end: CodeUnit): CodeUnit = end - start

}

/\*\*

\* Offsets in whole Unicode code points. Any CodePoint is a valid

\* offset into the String as long as it is >= 0 and less than the

\* number of code points in the string.

\*/

case class CodePoint(toInt: Int) extends AnyVal with Ordered[CodePoint] {

def toShort: Short = toInt.toShort

def compare(other: CodePoint): Int = toInt.compare(other.toInt)

def +(other: CodePoint) = CodePoint(toInt + other.toInt)

def -(other: CodePoint) = CodePoint(toInt - other.toInt)

def min(other: CodePoint): CodePoint = if (toInt < other.toInt) this else other

def max(other: CodePoint): CodePoint = if (toInt > other.toInt) this else other

def until(end: CodePoint): immutable.IndexedSeq[CodePoint] =

toInt.until(end.toInt).map(CodePoint(\_))

def toCodeUnit(text: String): CodeUnit =

CodeUnit(text.offsetByCodePoints(0, toInt))

}

implicit object CodePoint extends Offset[CodePoint] {

def toInt(p: CodePoint): Int = p.toInt

def count(text: String, start: CodeUnit, end: CodeUnit): CodePoint =

CodePoint(text.codePointCount(start.toInt, end.toInt))

}

/\*\*

\* Offsets into the String as if the String were encoded as UTF-8. You

\* cannot use a [[Utf8]] offset to index a String, because not all

\* Utf8 indices are valid indices into the String.

\*/

case class Utf8(toInt: Int) extends AnyVal with Ordered[Utf8] {

def compare(other: Utf8): Int = toInt.compare(other.toInt)

def +(other: Utf8) = Utf8(toInt + other.toInt)

def -(other: Utf8) = Utf8(toInt - other.toInt)

def min(other: Utf8): Utf8 = if (toInt < other.toInt) this else other

def max(other: Utf8): Utf8 = if (toInt > other.toInt) this else other

}

implicit object Utf8 extends Offset[Utf8] {

def toInt(u: Utf8): Int = u.toInt

/\*\*

\* Count how many bytes this section of text would be when encoded as

\* UTF-8.

\*/

def count(s: String, start: CodeUnit, end: CodeUnit): Utf8 = {

def go(i: CodeUnit, byteLength: Utf8): Utf8 =

if (i < end) {

val cp = s.codePointAt(i.toInt)

go(i + CodeUnit(Character.charCount(cp)), byteLength + forCodePoint(cp))

} else {

byteLength

}

go(start, Utf8(0))

}

/\*\*

\* Unfortunately, there is no convenient API for finding out how many

\* bytes a unicode code point would take in UTF-8, so we have to

\* explicitly calculate it.

\*

\* @see http://en.wikipedia.org/wiki/UTF-8#Description

\*/

def forCodePoint(cp: Int): Utf8 =

Utf8 {

// if the code point is an unpaired surrogate, it will be converted

// into a 1 byte replacement character

if (Character.getType(cp) == Character.SURROGATE) 1

else {

cp match {

case \_ if cp < 0x80 => 1

case \_ if cp < 0x800 => 2

case \_ if cp < 0x10000 => 3

case \_ => 4

}

}

}

}

/\*\*

\* Display units count what we consider a "character" in a

\* Tweet. [[DisplayUnit]] offsets are only valid for text that is

\* NFC-normalized (See: http://www.unicode.org/reports/tr15) and

\* HTML-encoded, though this interface cannot enforce that.

\*

\* Currently, a [[DisplayUnit]] is equivalent to a single Unicode code

\* point combined with treating "&lt;", "&gt;", and "&amp;" each as a

\* single character (since they are displayed as '<', '>', and '&'

\* respectively). This implementation is not directly exposed.

\*

\* It should be possible to change this definition without breaking

\* code that uses the [[DisplayUnit]] interface e.g. to count

\* user-perceived characters (graphemes) rather than code points,

\* though any change has to be made in concert with changing the

\* mobile client and Web implementations so that the user experience

\* of character counting remains consistent.

\*/

case class DisplayUnit(toInt: Int) extends AnyVal with Ordered[DisplayUnit] {

def compare(other: DisplayUnit): Int = toInt.compare(other.toInt)

def +(other: DisplayUnit) = DisplayUnit(toInt + other.toInt)

def -(other: DisplayUnit) = DisplayUnit(toInt - other.toInt)

def min(other: DisplayUnit): DisplayUnit = if (toInt < other.toInt) this else other

def max(other: DisplayUnit): DisplayUnit = if (toInt > other.toInt) this else other

}

implicit object DisplayUnit extends Offset[DisplayUnit] {

def toInt(d: DisplayUnit): Int = d.toInt

/\*\*

\* Returns the number of display units in the specified range of the

\* given text. See [[DisplayUnit]] for a descrption of what we

\* consider a display unit.

\*

\* The input string should already be NFC normalized to get

\* consistent results. If partially html encoded, it will correctly

\* count html entities as a single display unit.

\*

\* @param text the string containing the characters to count.

\* @param the index to the first char of the text range

\* @param the index after the last char of the text range.

\*/

def count(text: String, start: CodeUnit, end: CodeUnit): DisplayUnit = {

val stop = end.min(CodeUnit.length(text))

@annotation.tailrec

def go(offset: CodeUnit, total: DisplayUnit): DisplayUnit =

if (offset >= stop) total

else go(offset + at(text, offset), total + DisplayUnit(1))

go(start, DisplayUnit(0))

}

/\*\*

\* Return the length of the display unit at the specified offset in

\* the (NFC-normalized, HTML-encoded) text.

\*/

def at(text: String, offset: CodeUnit): CodeUnit =

CodeUnit {

text.codePointAt(offset.toInt) match {

case '&' =>

if (text.regionMatches(offset.toInt, "&amp;", 0, 5)) 5

else if (text.regionMatches(offset.toInt, "&lt;", 0, 4)) 4

else if (text.regionMatches(offset.toInt, "&gt;", 0, 4)) 4

else 1

case cp => Character.charCount(cp)

}

}

}

/\*\*

\* Ranges of offsets, useful for avoiding slicing entities.

\*/

sealed trait Ranges[T] {

def contains(t: T): Boolean

}

object Ranges {

private[this] case class Impl[T](toSeq: Seq[(T, T)])(implicit off: Offset[T])

extends Ranges[T] {

def contains(t: T): Boolean = toSeq.exists { case (lo, hi) => off.gt(t, lo) && off.lt(t, hi) }

}

/\*\*

\* Non-inclusive range of offsets (matches values that are strictly

\* between `hi` and `lo`)

\*/

def between[T](lo: T, hi: T)(implicit off: Offset[T]): Ranges[T] =

if (off.toInt(hi) > off.toInt(lo) + 1 && off.toInt(lo) < Int.MaxValue) Impl(Seq((lo, hi)))

else Impl(Nil)

/\*\*

\* The union of all of the specified ranges.

\*/

def all[T](ranges: Seq[Ranges[T]])(implicit off: Offset[T]): Ranges[T] =

Impl(

// Preprocess the ranges so that each contains check is as cheap

// as possible.

ranges

.flatMap { case r: Impl[T] => r.toSeq }

.sortBy(\_.\_1)

.foldLeft(Nil: List[(T, T)]) {

case ((a, b) :: out, (c, d)) if off.lt(c, b) => (a, d) :: out

case (out, r) => r :: out

}

)

def Empty[T: Offset]: Ranges[T] = Impl[T](Nil)

private[this] val HtmlEscapes = """&(?:amp|lt|gt);""".r

/\*\*

\* Match [[CodeUnit]]s that would split a HTML entity.

\*/

def htmlEntities(s: String): Ranges[CodeUnit] = {

val it = HtmlEscapes.findAllIn(s)

all(it.map(\_ => between(CodeUnit(it.start), CodeUnit(it.end))).toSeq)

}

def fromCodePointPairs(pairs: Seq[(Int, Int)]): Ranges[CodePoint] =

all(pairs.map { case (lo, hi) => between(CodePoint(lo), CodePoint(hi)) })

}

}