package com.twitter.tweetypie.tweettext

import scala.util.matching.Regex

object TextModification {

/\*\*

\* Lift a text into a TextModification where `original` and `updated` text are the same

\* and `replacements` is empty.

\*/

def identity(text: String): TextModification =

TextModification(original = text, updated = text, replacements = Nil)

/\*\*

\* Replace each substring that matches the regex with the substitution string, returns a

\* TextModification object that contains the updated text and enough information to also

\* update entity indices.

\*

\* This method should correctly be taking into account surrogate-pairs. The returned

\* TextModification object has code-point offsets, instead of code-unit offsets.

\*/

def replaceAll(text: String, regex: Regex, substitution: String): Option[TextModification] =

replaceAll(text, regex -> substitution)

/\*\*

\* Replaces substrings that match the given `Regex` with the corresonding substitution

\* string. Returns a `TextModification` that can be used to reindex entities.

\*/

def replaceAll(

text: String,

regexAndSubstitutions: (Regex, String)\*

): Option[TextModification] = {

val matches =

(for {

(r, s) <- regexAndSubstitutions

m <- r.findAllIn(text).matchData

} yield (m, s)).sortBy { case (m, \_) => m.start }

if (matches.isEmpty) {

// no match found, return None to indicate no modifications made

None

} else {

val replacements = List.newBuilder[TextReplacement]

val indexConverter = new IndexConverter(text)

// contains the retained text, built up as we walk through the regex matches

val buf = new StringBuilder(text.length)

// the number of code-points copied into buf

var codePointsCopied = Offset.CodePoint(0)

// always holds the start code-unit offset to copy to buf when we encounter

// either a regex match or end-of-string.

var anchor = 0

import indexConverter.toCodePoints

for ((m, sub) <- matches) {

val unchangedText = text.substring(anchor, m.start)

val unchangedLen = Offset.CodePoint.length(unchangedText)

val subLen = Offset.CodePoint.length(sub)

// copies the text upto the regex match run, plus the replacement string

buf.append(unchangedText).append(sub)

codePointsCopied += unchangedLen + subLen

// the offsets indicate the indices of the matched string in the original

// text, and the indices of the replacement string in the updated string

replacements +=

TextReplacement(

originalFrom = toCodePoints(Offset.CodeUnit(m.start)),

originalTo = toCodePoints(Offset.CodeUnit(m.end)),

updatedFrom = codePointsCopied - subLen,

updatedTo = codePointsCopied

)

anchor = m.end

}

buf.append(text.substring(anchor))

Some(TextModification(text, buf.toString, replacements.result()))

}

}

/\*\*

\* Inserts a string at a specified code point offset.

\* Returns a `TextModification` that can be used to reindex entities.

\*/

def insertAt(

originalText: String,

insertAt: Offset.CodePoint,

textToInsert: String

): TextModification = {

val insertAtCodeUnit = insertAt.toCodeUnit(originalText).toInt

val (before, after) = originalText.splitAt(insertAtCodeUnit)

val updatedText = s"$before$textToInsert$after"

val textToInsertLength = TweetText.codePointLength(textToInsert)

TextModification(

original = originalText,

updated = updatedText,

replacements = List(

TextReplacement.fromCodePoints(

originalFrom = insertAt.toInt,

originalTo = insertAt.toInt,

updatedFrom = insertAt.toInt,

updatedTo = insertAt.toInt + textToInsertLength

))

)

}

}

/\*\*

\* Encodes information about insertions/deletions/replacements made to a string, providing

\* the original string, the updated string, and a list of TextReplacement objects

\* that encode the indices of the segments that were changed. Using this information,

\* it is possible to map an offset into the original string to an offset into the updated

\* string, assuming the text at the offset was not within one of the modified segments.

\*

\* All offsets are code-points, not UTF6 code-units.

\*/

case class TextModification(

original: String,

updated: String,

replacements: List[TextReplacement]) {

private val originalLen = Offset.CodePoint.length(original)

/\*\*

\* Using an offset into the original String, computes the equivalent offset into the updated

\* string. If the offset falls within a segment that was removed/replaced, None is returned.

\*/

def reindex(index: Offset.CodePoint): Option[Offset.CodePoint] =

reindex(index, Offset.CodePoint(0), replacements)

/\*\*

\* Reindexes an entity of type T. Returns the updated entity, or None if either the `fromIndex`

\* or `toIndex` value is now out of range.

\*/

def reindexEntity[T: TextEntity](e: T): Option[T] =

for {

from <- reindex(Offset.CodePoint(TextEntity.fromIndex(e)))

to <- reindex(Offset.CodePoint(TextEntity.toIndex(e) - 1))

} yield TextEntity.move(e, from.toShort, (to.toShort + 1).toShort)

/\*\*

\* Reindexes a sequence of entities of type T. Some entities could be filtered

\* out if they span a region of text that has been removed.

\*/

def reindexEntities[T: TextEntity](es: Seq[T]): Seq[T] =

for (e <- es; e2 <- reindexEntity(e)) yield e2

/\*\*

\* Swaps `original` and `updated` text and inverts all `TextReplacement` instances.

\*/

def inverse: TextModification =

TextModification(updated, original, replacements.map(\_.inverse))

// recursively walks through the list of TextReplacement objects computing

// offsets to add/substract from 'shift', which accumulates all changes and

// then gets added to index at the end.

private def reindex(

index: Offset.CodePoint,

shift: Offset.CodePoint,

reps: List[TextReplacement]

): Option[Offset.CodePoint] =

reps match {

case Nil =>

if (index.toInt >= 0 && index <= originalLen)

Some(index + shift)

else

None

case (r @ TextReplacement(fr, to, \_, \_)) :: tail =>

if (index < fr) Some(index + shift)

else if (index < to) None

else reindex(index, shift + r.lengthDelta, tail)

}

}

object TextReplacement {

def fromCodePoints(

originalFrom: Int,

originalTo: Int,

updatedFrom: Int,

updatedTo: Int

): TextReplacement =

TextReplacement(

Offset.CodePoint(originalFrom),

Offset.CodePoint(originalTo),

Offset.CodePoint(updatedFrom),

Offset.CodePoint(updatedTo)

)

}

/\*\*

\* Encodes the indices of a segment of text in one string that maps to a replacement

\* segment in an updated version of the text. The replacement segment could be empty

\* (updatedTo == updatedFrom), indicating the segment was removed.

\*

\* All offsets are code-points, not UTF16 code-units.

\*

\* `originalFrom` and `updatedFrom` are inclusive.

\* `originalTo` and `updatedTo` are exclusive.

\*/

case class TextReplacement(

originalFrom: Offset.CodePoint,

originalTo: Offset.CodePoint,

updatedFrom: Offset.CodePoint,

updatedTo: Offset.CodePoint) {

def originalLength: Offset.CodePoint = originalTo - originalFrom

def updatedLength: Offset.CodePoint = updatedTo - updatedFrom

def lengthDelta: Offset.CodePoint = updatedLength - originalLength

def shiftOriginal(offset: Offset.CodePoint): TextReplacement =

copy(originalFrom = originalFrom + offset, originalTo = originalTo + offset)

def shiftUpdated(offset: Offset.CodePoint): TextReplacement =

copy(updatedFrom = updatedFrom + offset, updatedTo = updatedTo + offset)

def shift(offset: Offset.CodePoint): TextReplacement =

TextReplacement(

originalFrom + offset,

originalTo + offset,

updatedFrom + offset,

updatedTo + offset

)

def inverse: TextReplacement =

TextReplacement(

originalFrom = updatedFrom,

originalTo = updatedTo,

updatedFrom = originalFrom,

updatedTo = originalTo

)

}