RegexKit

Bernard Sufrin

Oxford, May 2022

Introduction

`RegexKit` provides an implementation of (more or less) standard regular expression matching and group capture, **forwards or backwards**, over (subsequences of) sequences of arbitrary type. We designed it because the standared Java/Scala API was too restrictive for our application – a text editor.

There is no backtracking for anchored matches which – in the case of a failure – take time bounded in proportion to the product of the 'length' of the regular expression (or, to be more precise, the number of non-housekeeping instructions in the code that it is translated to by the implementation) and the length of the subject subsequence.

Its `CharSequence` matching facilities are well-developed and tested (genericity came as an unplanned bonus).

The principal methods of the `CharSequence`-applicable API are the matching and searching methods of `Regex` that act on specified subsequences of the given subject:

```
// anchored methods
def suffixes  (subject: CharSequence, from: Int, to: Int,   bound: Int):Option[StringMatch]
def matches   (subject: CharSequence, from: Int, to: Int,   bound: Int):Option[StringMatch]

// searching methods
def findPrefix(subject: CharSequence, from: Int, to: Int,   bound: Int):Option[StringMatch]
def findSuffix(subject: CharSequence, from: Int, to: Int,   bound: Int):Option[StringMatch]

// repeated searches
def allPrefixes(subject: CharSequence, _from: Int, _to: Int, bound: Int): Iterator[StringMatch]
def allSuffixes(subject: CharSequence, _from: Int, _to: Int, bound: Int): Iterator[StringMatch]
```

and its substitution method, that substitutes an expanded instance of the template for each matching instance of this regular expression in the input. Return the substituted result with a count of the number of substitutions that were made. When `literal` is true, the template is not expanded.

```
def substituteAll(subject: CharSequence, template: String, literal: Boolean, bound: Int): (Int, String)
```

An individual `StringMatch` has a substitute method that replaces all occurences of `$$` in the template with `$`, and all occurences of `$`i (when i is a digit) with `group(`i`).getOrElse("")`.

```
def substitute(template: String): String
```

The API also provides methods for scanning forwards and backwards for (properly-nested) bracketed text, whose opening and closing brackets are specified by regular expressions.

Examples

The following examples are abstracted, somewhat edited, from the source code of the `Red` text editor.

Patterns matching the left and right boundaries of various granularities of text lump.

```
object Boundaries {
 import sufrin.regex.Regex
 val leftWord  : Regex   = Regex("""\W\w""")
 val rightWord : Regex   = Regex("""\w\W""")
 val leftLine  : Regex   = Regex("""(\n|^)[^\n]*""")
 val rightLine : Regex   = Regex.literal("\n")
 val leftPara  : Regex   = Regex("(\n|^)\\s*\n")
 val rightPara : Regex   = Regex("\n\\s*(\n|$)")
}
```

The method `selectChunk` performs a search, around the `start`, for a "chunk" (for example a word, line, or paragraph) whose left and right boundaries are specified by the regular expressions `l` and `r`. It returns the left and right boundaries of the chunk.

```
def selectChunk(start: Int, l: Regex, r: Regex, adjl: Int, adjr: Int): Option[(Int, Int)] =
    // find the last suffix matching l that ends no later than start
    l.findSuffix(document.characters, 0, start) match {
      // no such suffix
      case None => None
      // suffix extends from lp.start to lp.end
      // find the next prefix that starts no earlier than lp.end
      case Some(lp) =>
        r.findPrefix(document.characters, lp.end, document.characters.length) match {
          // no such suffix
          case None => None
          // suffix extends from rp.start to rp.end
          // chunk extends from lp.start to rp.end
          case Some(rp) =>
            val (start, end) =
                (if (lp.start == 0) lp.start else lp.start + adjl, rp.end - adjr)
            Some((start, end))
        }
    }
```

The method `find` performs an unanchored search from the current cursor position in the indicated direction.

```
def find(pattern: String, backwards: Boolean, literal: Boolean): Boolean = {
    val chars = document.characters
    try {
      val regex = if (lit) Regex.literal(pattern) else Regex(pattern)
      if (backwards) {
        val lastMatch = regex.findSuffix(chars, 0, cursor, Utils.stepLimit)
        lastMatch match {
          case None           => feedback.notify("Not found upwards", s"$pattern")
                                 false
          case Some(instance) => cursor = instance.start
                                 setMark(instance.end)
                                 true
        }
      } else {
        val nextMatch = regex.findPrefix(chars, cursor, chars.length, Utils.stepLimit)
        nextMatch match {
          case None           => feedback.notify("Not found", s"$pattern")
                                 false
          case Some(instance) => cursor = instance.end min (chars.length-1)
                                 setMark(instance.start)
                                 true
        }
      }
    } catch { case exn: sufrin.regex.syntax.lexer.SyntaxError =>
              notify("Find", s"Pattern: "$pattern" is ill-formed\n${exn.getMessage}")
              false }
  }
```

The search is for `pattern` interpreted either as a literal or as a regular expression. The cursor is moved to the start of the match, and the *mark* (the other end of the selection) is moved to the opposite end.

Implementation method

Regular expressions are translated into programs for an abstract "recognition machine". During matching, the machine simulates a nondeterministic finite state machine (NFA) by running a pool of *deterministic machines* (we call them *threads*) in pseudo-parallel: one for each unrefuted potential match. The state of each thread is embodied in its *program counter*, together with the starting and ending locations of each (parenthesised) group (it has so far) recognised.

Subject sequences are offered element by element to each thread in the pool, which either accepts, refutes, or jumps to its "next" instruction. Where there would be more than one outgoing edge from the state of the NFA, this "move" results in the construction of one (or more) additional *threads*. For example

```
r1 | r2 | r3
```

translates to (the equivalent) of

```
      goto L1, L2, L3
 L1:  code to recognise r1
      goto END
 L2:  code to recognise r2
      goto END
 L3:  code to recognise r3
END:
```

A deterministic match is refuted by an input that fails to match any outgoing edge from its current state; the corresponding machine drops out of the pool of active threads.

Potential Nontermination

The translation from regular expression to recognition machine code is straightforward, and no attempt is made to tarnsform the expression into a more efficient form, *or to a form that cannot (under some circumstances) cause nontermination*. With this in mind, the matching and searching methods may be given an upper bound to the number of recognition machine cycles they execute that causes them to abort if it is exceeded. *This is a pragmatic and expedient solution to the performance issues that would arise from an attempt to detect non termination: we find a more efficient method of non-termination detection in due course; but it's not a priority for us.*

Potential Speedups

Whilst the cost of a failing anchored RE match (`matches` or `suffixes`) is bounded linearly by the product of its "length" and the length of the subject subsequence, this is not so for the unanchored matches (`findPrefix`, `findSuffix`), for these are implemented as anchored matches from adjacent positions in the subject: the aggregate cost being, therefore, quadratic in the length of the subject subsequence. The resulting performance is (better than) acceptable in a text editor, but could nevertheless be improved by using a two-stage search.

In the context of using the kit in a text editor we made some compromises with efficiency, and noted in the source code, *and we quote*:

1. I gratefully acknowledge my colleague Mike Spivey's observation (based on the 1965 Thompson paper) that an appropriate technique for *searching* rather than *matching* is to spawn another thread at the origin of the program whenever the machine is about to move to consideration of the next element of a sequence.

2. Although implemented here (*i.e.* in the machine), the technique cannot be guaranteed to capture the content of nested groups appropriately during a search; indeed it can only ever be relied upon to find the end position of a match. One case in point is an expression of the form `(R1)+R2` when `R2` is empty, or has a prefix that may match a prefix of `R1`.

3. My judgment is that under many circumstances it is not worth trying to compose the above technique to find the endpoint of a match with a quadratic "match backwards" (from the endpoint) to capture groups accurately.

end quote

We now believe that we should provide methods in the API that are implemented

using the method sketched in (3). We'd welcome collaboration in implementing this. Much of the necessary machinery is already present in the current codebase.