Regular Expressions
And
Its Applications

BILICI, M. Şafak
safakk.bilici.2112@gmail.com
Yildiz Technical University

Abstract

In this paper, we will examine Regular Expressions, also known as RegEx or RegExp, and show some practical results detailed.

Introduction

A regular expression (shortened as regex or regexp; also referred to as rational expression) is a sequence of characters that define a search pattern. Usually such patterns are used by string-searching algorithms for 'find' or 'find and replace' operations on strings. It is a technique developed in theoretical computer science and formal language theory [1]. Regular expressions are used in search engines, search and replace dialogs of word processors and text editors, in text processing utilities such as sed and AWK and in lexical analysis. Many programming languages provide regex capabilities either built-in or via libraries.
Regex is one of unheard success in standardization in computer science, it is a language for specifying text search strings. This practical language is used in every computer language, word processor, and text processing tools like the `grep`, `tr` [2] or Emacs. Formally, a regular expresison is a algebraic notation for characterizing a set of strings.

History

Regular expressions originated in 1951, when mathematician Stephen Cole Kleene described regular languages using his mathematical notation called regular events. These arose in theoretical computer science, in the subfields of automata theory (models of computation) and the description and classification of formal languages. Other early implementations of pattern matching include the SNOBOL language, which did not use regular expressions, but instead its own pattern matching constructs.
Regular expressions entered popular use from 1968 in two uses: pattern matching in a text editor and lexical analysis in a compiler. Among the first appearances of regular expressions in program form was when Ken Thompson built Kleene's notation into the editor QED as a means to match patterns in text files.
He later added this capability to the Unix editor ed, which eventually led to the popular search tool grep's use of regular expressions ("grep" is a word derived from the command for regular expression searching in the ed editor: g/re/p meaning "Global search for Regular Expression

and Print matching lines")
In the 1980s the more complicated regexes arose in Perl, which originally derived from a regex library written by Henry Spencer.
Today, regexes are widely supported in programming languages, text processing programs, advanced text editors, and some other programs. Regex support is part of the standard library of many programming languages, including Java and Python, and is built into the syntax of others.

Diving Into Regular Expression Patterns

The simplest kind of regular expression is a sequence of simple character/characters . To search *safak*, we type `/safak/`. The expression `/safak/` matches any string containing the substring *safak*.

RE	**Matched**
`/safak/`	"safak's teaching characteristics are boring"
`/a/`	"safak and sadi are best friends"
`/!/`	"Sky Lab AI is the best !"

Regular expressions are case sensitive; lower case `/s/` is distinct from upper case `/S/`. This means that the pattern `/safak/` will not match the string *Safak*. We can solve this problem with the use of square braces [and]. The string of characters inside the braces specifies **disjunction** of characters to match.

RE	**Matched**
`/[Ss]afak/`	"safak's teaching characteristics are boring. Safak sucks"
`/[abc]/`	"safak and sadi are best friends c."
`/[0123456789]/`	"I have 2 kinds and husband."

You can cleary see that it is too long to define the whole letters in a braces or even whole numbers. We can define the **range** with (-) character. The pattern `/[2-5]/` specifies any one of the characters 2, 3, 4, or 5. The pattern `/[A-Za-z]/` matches the whole letters in english including upper cased letters.
The square braces can also be used to specify what a single character **cannot** be, by use of the caret (^). If the caret ^ is the first symbol after the open square brace [, the resulting pattern is **negated**.

RE	**Matched**
`/[^A-Z]/`	not an upper case
`/[^Ss]/`	neither "S" nor "s"
`/[^.]/`	not a period
`/[e^]/`	either "e" or "^"
`/a^b/`	the pattern 'a^b'

How can we talk about optional elements, like an optional *s* in *researcher* and *researchers*? We can't use the square brackets, because they allow us to say "s or S1, not "s or nothing". For this we use the question mark (?), which **the predecing character or nothing**. We can think of the question mark as meaning **zero or one instances of the previous character**.

RE	**Matched**
`/researchers?/`	researcher or researchers
`/colou?r/`	color or colour

But there is still a problem about "how many of something we want" which is very important in regular expressions. For example we want to match with patterns like

→ yy!

→ yeey!

→ yeeeeeeey!

→ yeeeeeeeeeeeeeeeey!

→ ...

A type of regex operator called **Kleene Star (*)** can do this sequential string matching. So, the expression `/e*/` means **any string of zero or more *e*'s**. And the regular expressions for matching one or more *e* is `/ee*/`, meaning one *e* followed by zero or more *e*'s. But sometimes it is annoying to have to write the regular expression for digits twice, so there are another operator called **Kleene Plus (+)** to represent this type of matching. It means **one or more ocurrences of the immediately preceding character or regular expression**. So `/ye+y!/` matches all strings above except the *yy!*.

One very important special characteris the period (/./), a **wildcard** expression that matches any single character

RE	**Matched**
`/beg.n/`	"begin, began, begun, beg'n ..."

The wildcard is often used with Kleene star to mean "any string of characters". For example to find any line which a particular word, for example, *aardvark*, appears twice. We can specify this with `/aardvark.*aardvark/`.

Anchors are special characters that anchor regular expressions to particular places in a string. The most common anchors are the caret ^ and the dollar sign $. The caret ^ matches the start of a line. The pattern `/^The/` matches the word *The* only at the start of a line. The dollar sign $ matches the end of a line. So the pattern ␣$ is a useful pattern for mathing a space at the end of a line and the `/^The dog$/` matches a line that contains only the phrase "*The dog.*" (ṁeans only the character dot).
There are also two other anchors; \b matches a word-boundary, \B matches non-word-boundary.

RE	Matched
`/^[^A-Za-z]/`	"Hello"
`/\.$/`	"The end."
`/.$/`	"The end. The end?"

Suppose we need to search for texts about pets; perhaps we are particularly interested in cats and dogs. In such a case, we might want to search for either the string cat or the string dog. Since we can't use the square brackets to search for "cat or dog" (why disjunction can't we say `/[catdog]/`, we need a new operator, the **disjunction** operator, also called the **pipe** symbol (|). The pattern `/cat|dog/` matches either string cat or dog.

But there is a problem. How can I specify both guppy and guppies? We cannot simply say `/guppy|ies/`, because that would match only the strings *guppy* and *ies*. To make the disjunction operator apply only to a specific pattern, we need to use the parenthesis operators (and). Enclosing a pattern in parentheses makes it act like a single character for the purposes of neighboring operators like the pipe | and the Kleene*. So the pattern `/gupp(y|ies)/` would specify that we meant the disjunction only to apply to the suffixes *y* and *ies*.
There is also quantifiers like `/{3,5}/` that means 3, 4, or 5.

An Application

Suppose that you want to build an application to help a user to buy a computer on the web. The user might want "any machine with at least 6 GHz and 500 GB of disk space for less than $1000". Let's build a regex. First let's start with regular expression for prices

```
/\$[1-9]+[0-9]*/
```

This can specifies prices. But prices can be type of float. We need to define floating point.

```
/\$[1-9]+[0-9]*([̇0-9]*)?\b/
```

can do that. One last catch, this pattern also matches prices like $19999999.99999 which is too expensive, let's add quantifier.

```
/\$[1-9][0-9]{0,2}([̇0-9]*)?\b/
```

Now let's define disk spaces with quantifier.

```
/\$[1-9][0-9]{0,2}\b(\.[0-9]*)?␣[1-4][0-9]{0,2}(GB|␣[Gg]igabytes)/
```

References

- https://lse-my472.github.io/week08/regular-expressions-cheat-sheet-v2.pdf
- Speech and Language Processing - Daniel Jurafsky, James H. Martin