Regular Expressions

Reg Dodds
Department of Computer Science
University of the Western Cape
rdodds@uwc.ac.za

September 18, 2017

Chapter 1

Regular Expressions

REGULAR EXPRESSIONS are an easy way to tackle a variety of problems, especially those involving searching for, or mining for, information in text files.

For example, a regular expression may be used in order to extract all the words consisting only of lower case letters from the text of a book. A regular expression can be constructed to find all the strings of four digits in a web page. A regular expression can be applied look up any tag in a web page and replace it with a different mark-up symbol, such as in the process of converting a file in Rich text format (RTF) to its Hypertext mark up language (HTML) equivalent.

1.1 Defining Regular Expressions

In order to use regular expressions it helps first to understand the rules needed to form them. Regular expressions are a way of describing a subset of strings that are built up out of an *alphabet* we will write as Σ, where Σ may be a simple set, e.g. consisting only of zero and one, i.e., the set $\{0, 1\}$.

The set of all *subsets* that can be built from the set $\Sigma = \{0, 1\}$, i.e., its superset $\mathscr{P}(\Sigma) = \{\{\}, \{0\}, \{1\}, \{0, 1\}\}$. Many *strings* can be built from the the alphabet $\Sigma = \{0, 1\}$, namely, the strings,

$$\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, 1000, \ldots,$$

to name a few of them. This particular set is called the *closure* of the set $\Sigma = \{0, 1\}$, and is written as Σ^* or $\{0, 1\}^*$. Regular expressions can be used to define subsets of Σ^*, such as all the even, or odd binary numbers, all

the numbers with odd parity, all those elements, consisting only of ones or only of zeros, all the numbers with an odd or even number of digits.

Of all the possible sets of strings that we can build from Σ the smallest one is the empty set $\emptyset$. So, the first rule to make a *regular expression* follows.

1. The empty set $\emptyset$ is a *regular expression*.

Of course, if we build all possible strings from any set, then it we uncomfortably have to make provision for the construction an empty string, that is written as ε,[1] giving the second rule for producing regular expressions.

2. The empty string ε is a *regular expression*.

Since of all the possible strings that we can produce from the alphabet Σ, any single element of Σ may also be regarded as a *regular expression*—giving us the third rule.

3. If $x \in \Sigma$ then x is a *regular expression*.

Given $\Sigma = \{0, 1\}$, and the first three rules it follows that $\emptyset$, ε, 0, and 1 can all be regular expressions. These are the building blocks for building all the other regular expressions from the alphabet $\Sigma = \{0, 1\}$.

The following rules describe how regular expressions can be built from other regular expressions. The strings from a set built by a regular expression may be collected into one set, they may concatenated, or any multiple of copies of them may be combined into regular sets. The fourth rule describes the union of two sets formed by regular expressions

4. If R_1 and R_2 are two regular expressions, the set formed by their *union* $(R_1 \cup R_2)$ is a regular expression.

The fifth rule describes how new sets can be formed by concatenating the strings formed by different regular expressions.

5. If R_1 and R_2 are two regular expressions, the set formed by their *concatenation* $(R_1 \circ R_2)$ is a regular expression.

Finally, the sixth rule describes how multiple concatenations of the same regular expression may be used to form a regular expression.

6. The set formed by (R^*), where R is a regular expression and (R^*) represents strings formed by a regular expression concatenated any multiple of times to itself, including zero multiples, is a regular expression.

[1] Programmers are usually not overly surprised by a statement such as `s = ""`, which assigns an empty string to the variable `s`.

The * in the regular expression R^* indicates what is known as the *closure* of the regular expression R. This closure is the set $\{\varepsilon, R, R^2, \ldots\}$.[2]

In summary a *regular expression* is:

1. the empty set $\emptyset$,
2. the empty string ε,
3. x if $x \in \Sigma$,
4. the *union* $(R_1 \cup R_2)$, where R_1 and R_2 are two regular expressions,
5. the *concatenation* $(R_1 \circ R_2)$, where R_1 and R_2 are two regular expressions, or
6. the *closure* (R^*) of a regular expression R.

Shorthand for $R_1 \circ R_2$ is R_1R_2, and instead of using $\cup$ for *union* a $+$ or a $|$ is used, so that $R_1 \cup R_2 = R_1 + R_2 = R_1|R_2$. Parentheses are used to alter the order of operations when the priority rules do not give the intended meaning. They may be dropped when the order of operations is according to priority.

The *priority rules* are simple. (1) An expression inside parentheses is done first, (2) then the closure operator, and (3) concatenation follows, and (4) union has the lowest priority. Two operators of the same priority are executed from left to right, e.g. two concatenations, or two unions.

Given that A, B, C, and D are regular expressions here are some examples where the parentheses clarify the order of execution of the operators.

Examples

1. $A \cup B \cup C = ((A \cup B) \cup C)$.
2. $A \circ B \circ C = ABC = ((A \circ B) \circ C)$.
3. $A \circ B^* \circ C = AB^*C = (((A \circ (B^*)) \circ C)$.

[2] Mathematically, this is written $R^* = \bigcup_{i=0}^{\infty} R^i = \{\varepsilon\} \cup R \cup R^2 \cup \ldots$, where R^i is defined as, $R^0 = \{\varepsilon\}$, and $R^{i+1} = R^iR$, for $i \in \mathbb{N}$, where $\mathbb{N}$ is the set of natural numbers $\{1, 2, 3, \ldots\}$.

4. $A \cup B^* \circ C = (A \cup ((B^*) \circ C))$.
5. $A \cup B \circ C \cup D = A \cup BC \cup D = ((A \cup (B \circ C)) \cup D)$.
6. $(A \cup B) \circ (C \cup D) = ((A \cup B) \circ (C \cup D))$.

Given that R_i is a regular expression for $i \in \mathbb{N}$ then the following are examples of regular expressions over the alphabet $\Sigma = \{a, b, c, d\}$

Examples

1. ε yields the set $\{\varepsilon\}$.
2. $\emptyset$ yields the set $\{\}$ or $\emptyset$.
3. a is the set $\{a\}$.[3]
4. $a \circ b = ab$ is the set $\{ab\}$.
5. $a \cup b$ is the set $\{a, b\}$.
6. $a \cup b \circ c$ is the set $\{a, bc\}$.
7. $a \circ b \cup c \circ d$ is the set $\{ab, cd\}$.
8. $a \cup b \circ c \cup d$ is the set $\{a, bc, d\}$.
9. $a \cup b \cup c \circ d$ is the set $\{a, b, cd\}$.
10. a^* is the set $\{\varepsilon, a, aa, aaa, \ldots\}$
11. a^*bcd is the set $\{bcd, abcd, aabcd, aaabcd, \ldots\}$
12. $a^*\emptyset = \emptyset a^* = \emptyset = \{\}$.
13. $\emptyset^* = \{\varepsilon\} = \varepsilon$.
14. $R \cup \emptyset = \emptyset \cup R = R$, and $R \circ \varepsilon = \varepsilon \circ R = R$.
    But beware, $R \cup \varepsilon \neq R$, because the union adds ε to R, when $\varepsilon \notin R$, and $R \circ \emptyset = \emptyset \circ R = \emptyset$.
15. $\emptyset\emptyset^* = \{\} = \emptyset$.
16. $R\emptyset^* = R$.

[3] The a is the regular expression, and $\{a\}$ is the set that it represents. The literature and these notes often conflate a and $\{a\}$, abc and $\{abc\}$, etc.

17. $R^* \cup \emptyset^* = R^*$, but when $\varepsilon \notin R$, then $R \cup \emptyset^* \neq R$, ε has been unioned to R.

18. $R\emptyset = \emptyset R = \emptyset = \{\}$, i.e., $R \circ \emptyset = \emptyset \circ R = \emptyset$.

19. $\varepsilon^* = \varepsilon$.

1.2 Regular Expressions in programming

A reasonably full list of commands used when programming with regular expressions is given in Section **??**.

The meta-characters are used in commands are **`{ } [ ] ( ) ^ $ . , | * + - \`**. We will describe some of their uses briefly. Most of the following applies in almost every programming language or tool.

1. The *asterisk* **`*`** is used as the closure of a regular expression, e.g. **`a*`** matches any of the empty string, **`a`**, **`aa`**, **`aaa`**, etc., greedily, i.e., as much as possible is matched.

2. A *plus* **`+`** is used as the closure of a regular expression that contains at least one item, e.g. **`b+`** matches any of **`b`**, **`bb`**, **`bbb`**, etc., greedily, i.e., as much as possible is matched.

3. The *caret* **`^`** matches the beginning of a string or line.

4. The *dollar* **`$`** matches the end of a string or line.

5. The *period* or dot **`.`** matches any character excepting the newline character, e.g. the pattern **`^.+$`** matches all the characters of a non-empty line, excepting the final newline character.

6. The *braces* **`{`** and **`}`** are used indicate a fixed number of repetitions, e.g. **`c{0,}`** has the same meaning as **`c*`**, **`c{1,}`** has the same meaning as **`c+`**, and **`c{2,4}`** will greedily match any of, **`cc`**, **`ccc`**, or **`cccc`**, and **`c{3,3}`** will match only **`ccc`**.

7. The *backslash* escapes or overrides the meaning of the following character. In order to match a string of exactly three plusses the pattern **`\\+{3,3}`** or the pattern **`\\+\\+\\+`** or can be used. The backslash used in combination with certain letters or digits gives that character a special meaning, e.g. **`\d`** will match any digit.

8. The *parentheses* **(** and **)** are used to group regular expressions, e.g. **(ab*c)** will greedily match any of **ac**, **abc**, **abbc**, **abbbc**, etc. The groups that are matched are numbered from 1. The number of the group is determined by its first left parenthesis. If any replacement is going to be done, then group 1 is denoted by **\1**, and group 2 is denoted by **\2**, etc. This can be very useful for manipulating text. e.g. in the editor **vi** to change all dates in the format **mm/dd/yyyy** to dates in the form **yyyy-mm-dd** the following search and replace command can be used, **%s:\(\d\d\)/\(\d\d\)/\(\d\d\d\d\):\3-\2-\1:g+**. This alters the dates **11/12/1982** and **20/05/1987** to **1982-12-11**, and **1987-05-20**.

9. *Downstroke* is used to separate alternatives in a pattern, e.g. the pattern **a|bc|efg** matches one of **a** or **bc** or **efg**.

10. *Brackets* **[** and **]** are used to define classes, e.g. **[aeiou]** defines a class that contains the vowels. Inside brackets, *minus* **-** indicates a range when it occurs between two characters, such as **[0-9]**, which matches any digit and has the same meaning as **\d**. The class of all the lowercase letters is given by the pattern **[a-z]**. The class **[A-Z]** matches any uppercase letter and **[a-zA-Z]** matches all the letters. The caret negates a class when it is the first character of a class, e.g. **[^0-9]** matches anything that is not a digit.

    Some regular expression engines support named classes in the form of **[[:***name***:]]**, where *name* is one of **alnum**, **alpha**, **ascii**, **blank**, **cntrl**, **digit**, **graph**, **lower**, **print**, **punct**, **space**, **upper**, and **xdigit**. These are useful when accented letters or non-roman aphabets are required. Note that **[[:blank:]]** matches only the *space*—ASCII = 32—and the *tab*—ASCII = 9—characters, whereas the class **[[:space:]]** and **\s** are both the same as **[\t\n\r\f\v]**.

1.3 Regular Expressions in the command line interface

1.4 Regular Expressions in Python

When regular

1.5 Regular Expressions in Java

1.6 Regular Expressions in the `vim` Editor

In our notation we

Username

Username with at least four and at most sixteen letters, digits underscores, or hyphens.

```
^[a-zA-Z0-9_-]{4,16}$
```

`^`	Start of the line
`[a-zA-Z0-9_-]`	Match letters, digits, underscore and hyphen
`{4,16}`	Match at least 4 and at most 16 characters
`$`	End of the line

⇒ See the explanation and example here

Password

A password that contains at least two digits, one lowercase and one uppercase letter.

```
((?=.*\d.*\d)(?=.*[a-z])(?=.*[A-Z]).{4,16})
```

`(`	Start of group
`(?=.*d.*d)`	Must contain at least two decimal digits
`(?=.*[a-z])`	Must contain one lowercase character
`(?=.*[A-Z])`	Must contain one uppercase character
`.{4,16}`	Match anything with prior condition checking with a length at least 4 and atmost 16 characters
`)`	End of group

Colour Code in Hexadecimal or Decimal

Colour code in the form of a hash followed by six hexadecimals or three comma separated one to three digit numbers, e.g. `#f0ff8e`, `#FACE33` or `1,12,123`. `^#([0-9a-fA-F]{6}\|\d{1,3},\d{1,3},\d{1,3}\$`

`^`	start of the line
`#`	must start with #
`(`	start of group 1
`[0-9a-fA-F]{6}`	hex string of length of 6
`|`	or
`\d{1,3}`	decimal string of length 1-3
`,\d{1,3}`	, decimal string of length 1-3
`,\d{1,3}`	, decimal string of length 1-3
`)`	end of group 1
`$`	end of the line

A floating point number

There are various definitions of floating point numbers. It is easy to write regular expressions that match floating point numbers. Consider the definition.

A floating point number may have an optional plus **`+`** *or* minus **`-`** *sign followed by the* whole part *of at least one and not more than* 15 *digits, possess a compulsory* point **`.`***, and this is followed by an optional* fraction *in the form of a string of up to 15 digits, and an optional* exponent *consisting of upper case* **`E`** *or lower case* **`e`** *followed by an optional* sign *of plus* **`+`** *or minus* **`-`** *which in turn is followed by one to three digits.*

```
[+-]?\d{1,15}[.]\d{,15}([Ee][+-]?\d{1,3})?
```

This matches the numbers
`+123.456789E+10`,
`123.456789E123`,
`123.E+4`,
`123.E4`,
`1230000.`,
`-123.45678`, and
`-123.456789E+01`,

but does not match
`+.456789E+10`,
`123456789E123`,
`123.E`,
`-12345678`, and
`-123.+01`,

.

Email

Image File Extension

IP Address

Time Format

Time in 24-Hour Format

Date Format (dd/mm/yyyy)

HTML Tag

HTML links

HTML A Tag

Extract HTML

1.7 Exercises

1.7.1 Exercises for ??

Let $\Sigma = \{0, 1\}$. Give regular expressions for the following subsets of Σ^*.

1. the even binary numbers,
2. the odd binary numbers,
3. all the numbers with odd parity,
4. all the numbers with odd parity,
5. elements consisting only of ones or only of zeros,
6. elements consisting only of zeros,
7. elements with an odd number of digits,
8. elements with an even number of digits,
9. elements with an number of digits that is divisible by three.

Quick Reference

	Regular Expressions Literal Characters
\n	Newline, **\x0A**
\r	Carriage return, **\x0D**
\t	Tab, **\x09**
\v	Vertical tab, **\x09**
\f	Form feed, **\x0C**
\a	Alarm, alert, bell, or beep, **\x07**
\e	Escape, **\x1B**
\ddd	ASCII character expressed in octal **ddd**, e.g. **\007**
\xdd	ASCII character expressed in hexadecimal **dd**, e.g. **\x1B**
\cA	The control character **^A**, e.g. **\cJ** is equivalent to **\n** and **\cI** is equivalent to **\t**
	Character Classes
[...]	A character between the brackets
[^...]	A character not between the brackets
.	Any character except newline, same as **[^\n]**
\d	A digit, same as **[0-9]** or **[[:digit:]]**
\D	Not a digit, same as **[^0-9]**or **[^[:digit:]]**
\w	A word character, same as **[a-zA-Z0-9_]** or **[[:alnum:]_]**
\W	A non-word character, same as **[^a-zA-Z0-9_]** or **[^[:alnum:]_]**
\s	A whitespace character, same as **[\t\n\r\f\v]** or **[[:space:]]**
\S	A non-whitespace, same as **[^\t\n\r\f\v]** or **[^[:space:]]**
[\b]	A backspace character, **b** matches a word boundary
posix class	Use as **[[:alnum:]]**: **alnum**, **alpha**, **ascii**, **blank**, **cntrl**, **digit**, **graph**, **lower**, **print**, **punct**, **space**, **upper**, **xdigit**
	Replacement
****	Turn off the special meaning of the following character
\n	Print text matched by **n**th, **1**≤**n**≤**9**, pattern, matched by **\(** and **\)**
&	Reuse matched text as part of the replacement pattern
~	Reuse matched text in the current replacement pattern
%	Reuse the previous replacement pattern in the current replacement pattern
\l	Change first character of replacement pattern to lowercase
\L	Change entire replacement pattern to lowercase
\u	Change first character of replacement to uppercase
\U	Change entire replacement pattern to uppercase

	Repetition
`{n}`	Match `n` times
`{n,m}`	Match `n`–`m` times
`{n,}`	Match at least `n` times
`?`	Ignore or match once, same as `{0,1}`
`+`	Match at least once, same as `{1,}`.
`*`	Match zero or more times, same as `{0,}`
`{}?`	Match as few times as possible, i.e., ungreedily—exclude characters from next match
`??`	Match as few times as possible, i.e., ungreedily
`+?`	Match as few times as possible, i.e., ungreedily
`*?`	Match as few times as possible, i.e., ungreedily, e.g. `^(.*?)\s*$`, grouped expression excludes trailing spaces.
	Options
`g`	Match everything, do not stop after first match.
`i`	Ignore case.
`m`	Match multiple lines, i.e., `^` and `$` match internal `\n`.
`s`	Treat string as a line, i.e., `^` and `$` ignore `\n`, but `.` matches it.
`x`	Insert comments or whitespace.
	Extended Regular Expression
`(?#...)`	Comment, ignore "...".
`(?:...)`	Matches but does not return "..."
`(?=...)`	Matches if expression matches "..." next
`(?!...)`	Matches if expression idoes notmatch "..." next
`(?imsx)`	Change matching options during matching.
	Grouping
`(...)`	Group items into a unit, e.g. to use with `*`, `+`, `?`, `|`, etc.
`|`	Alternation. Match one of the alternatives.
$\backslash n$	Match characters of group n.
	Anchors
`\^`	Match beginning of string or line.
`$`	Match end of string or line
`\b`	Match a word boundary, i.e., match all between a `\w` character and a `\W` character
`\B`	Match a position that is not a word boundary
`[\b]`	Matches backspace, `\b` matches a word boundary

The Posix Character Classes

Class	Meaning
alnum	Letter or digit.
alpha	Letter.
blank	Space or tab only.
cntrl	Control character.
digit	Decimal digit.
graph	Printing character, excluding space.
lower	Lowercase letter.
print	Printing character, including space.
punct	Printing character, excluding letters and digits.
space	Whitespace.
upper	Uppercase letter.
xdigit	Hexadecimal digit.