Al-Azhar UNIVERSITY

Faculty of Engineering

Computers and Systems Engineering Department

- -

EXPERIMENT Regular Expression

OBJECTIVES

(A) To understand the syntax of regular expressions.
(B) Know the limitations of regular expressions

MATERIALS/EQUIPMENT NEEDED

Java and any suitable IDE

Introduction

Regular expression is to express a characteristic in a string, and then to match another string with the characteristic. For example, pattern "ab+" means "one 'a' and at least one 'b' ", so "ab", "abb", "abbbbbbb" match the pattern.

Regular expression is used to : (1) test a string whether it matches a pattern, such as a email address. (2) To find a substring which matches certain pattern, from a whole text. (3) to do complex replacement in a text.

It is very simple to study regular expression syntax, and the few abstract concepts can be understood easily too. Many articles does not introduce its concepts from simple ones to abstract ones step by step, so some persons may feel it is difficult to study. On the other hand, each regular expression engine's document will describe its special function, but this part of special function is not what we should study first.

1. Regular Expression Basic Syntax

1.1 Common Characters

Letters, numbers, the underline, and punctuations with no special definition are "common characters". When regular expression matches a string, a common character can match the same character.

1.2 Simple escaped characters

Nonprinting characters which we know:

Expression	Matches
\r, \n	Carriage return, newline character
\t	Tabs
\\	Matches "\" itself

Some punctuation is specially defined in regular expression. To match these characters in string, add "\" in pattern. For example: ^, $ has special definition, so we need to use "\^" and "\$" to match them.

Expression	Matches
\^	Matches "^" itself
\$	Matches "$" itself
\.	Matches dot(.) itself

These escaped characters have the same effect as "common characters": to match a certain character.

1.3 Expression matches anyone of many characters

Some expressions can match anyone of many characters. For example: "\d" can match any number character. Each of these expressions can match only one character at one time, though they can match any character of a certain group of characters.

Expression	Matches
\d	Any digit character, any one of 0~9
\w	Any alpha, numeric, underline, any one of A~Z,a~z,0~9,_
\s	Any one of space, tab, newline, return, or newpage character

.	'.' matches any character except the newline character(\n)

1.4 Custom expression matches anyone of many characters

Expression uses square brackets [ ] to contain a series of characters, it can match anyone of them. Uses [^] to contain a series of characters, it can match anyone character except characters contained.

Expression	Matches
[ab5@]	Matches "a" or "b" or "5" or "@"
[^abc]	Matches any character except "a","b","c"
[f-k]	Any character among "f"~"k"
[^A-F0-3]	Any character except "A"~"F","0"~"3"

1.5 Special expression to quantify matching

All expressions introduced before can match character only one time. If a expression is followed by a quantifier, it can matches more than one times.

For example: we can use the pattern "[bcd]{2}" instead of "[bcd][bcd]".

Expression	Function
{n}	Match exactly n times, example: "\w{2}" equals "\w\w"; "a{5}" equals "aaaaa"
{m,n}	At least n but no more than m times: "ba{1,3}" matches "ba","baa","baaa"
{m,}	Match at least n times: "\w\d{2,}" matches "a12","_456","M12344"...
?	Match 1 or 0 times, equivalent to {0,1}: "a[cd]?" matches "a","ac","ad".
+	Match 1 or more times, equivalent to {1,}: "a+b" matches "ab","aab","aaab"...
*	Match 0 or more times, equivalent to {0,}: "\^*b" matches "b","^^^b"...

1.6 Some special punctuation with abstract function

Some punctuation in pattern has special function:

Expression	Function
^	Match the beginning of the string
$	Match the end of the string
\b	Match a word boundary

Some special punctuation can make effect on other sub-patterns:

Expression	Function
|	Alternation, matches either left side or right side
( )	(1). Let sub-patterns in it to be a whole part when it is quantified. (2). Match result of sub-patterns in it can be retrieved individually

---

2. Regular expression advanced syntax

2.1 Reluctant or greedy quantifiers

There are several method to quantify subpattern, such as: "{m,n}", "{m,}", "?", "*", "+". By default, a quantified subpattern is "greedy", that is, it will match as many times as possible (given a particular starting location) while still allowing the rest of the pattern to match. For example, to match "dxxxdxxxd":

Expression	Match result
(d)(\w+)	"\w+" matches all characters "xxxdxxxd" behind of "d"
(d)(\w+)(d)	"\w+" matches all characters "xxxdxxx" between the first "d" and the last "d". In order to let the whole pattern match success, "\w" has to give up the last "d", although it can match the last "d" too.

Thus it can be seen that: when "\w+" matches, it will match as many characters as possible. In the second example, it does not match the last "d", but this is in order to let the whole pattern match successfully. Pattern with "*" or "{m,n}" will also match as many times as possible, pattern with "?" will match if possible. This type of matching is called "greedy matching". 。

Reluctant Matching:

To follow the quantifier with a "?", it can let the pattern to match the minimum number of times possible. This type of matching is called reluctant matching. In order to let the whole pattern match successfully, the reluctant pattern may match a few more times if it is required. For example, to match "dxxxdxxxd":

Expression	Match result
(d)(\w+?)	"\w+?" match as few times as possible, so "\w+?" matches only one "x"
(d)(\w+?)(d)	In order to let the whole pattern match successfully, "\w+?" has to match "xxx". So, match result is: "\w+?" matches "xxx"

More examples:

Example1: When pattern "<td>(.*)</td>" matches "<td><p>aa</p></td> <td><p>bb</p></td>", match result: success; substring matched: the whole "<td><p>aa</p></td> <td><p>bb</p></td>", "</td>" in the pattern matches the last "</td>" in the string.

Example2: For comparison, when pattern "<td>(.*?)</td>" matches the string in example1, it matches "<td><p>aa</p></td>". When match next, the next "<td><p>bb</p></td>" can be matched.

3. Other usually supported rules

There are several usually supported rules which have not been mentioned.

3.1 In pattern, a character can be expressed as "\xXX" or "\uXXXX" ("X" is a hex number)

Format	Character range
\xXX	0 ~ 255, such as space can be "\x20"
\uXXXX	Any character can be expressed as "\u" plus 4 hex numbers, such as "\u4E2D"

3.2 While "\s", "\d", "\w", "\b" are specially defined, their uppercase letters have the opposite meaning

Pattern	Matches
\S	All characters except spaces
\D	All characters except numeric characters
\W	All characters except alpha, numeric, "_"
\B	Characters' gap which is not a word boundary

3.3 Specially defined characters table

Character	Description
^	Matches the beginning of the string. Use "\^" to match "^" itself
$	Matches the end of the string. Use "\$" to match "$" itself
( )	Grouping. Use "\(" and "\)" to match "(" and ")"
[ ]	Character class. Use "\[" and "\]" to match "[" and "]"
{ }	Define quantifiers. Use "\{" and "\}" to match "{" and "}"
.	Match any character except newline(\n). Use "\." to match "." itself
?	Let subpattern match 0 or 1 time. Use "\?" to match "?" itself
+	Let subpattern match at least 1 times. Use "\+" to match "+" itself

*	Let subpattern match any times. Use "*" to match "*" itself
|	Alternation. Use "\\|" to match "|" itself

3.4 If a subpattern is in "(?:xxxxx)", the match result is not recorded for later use.

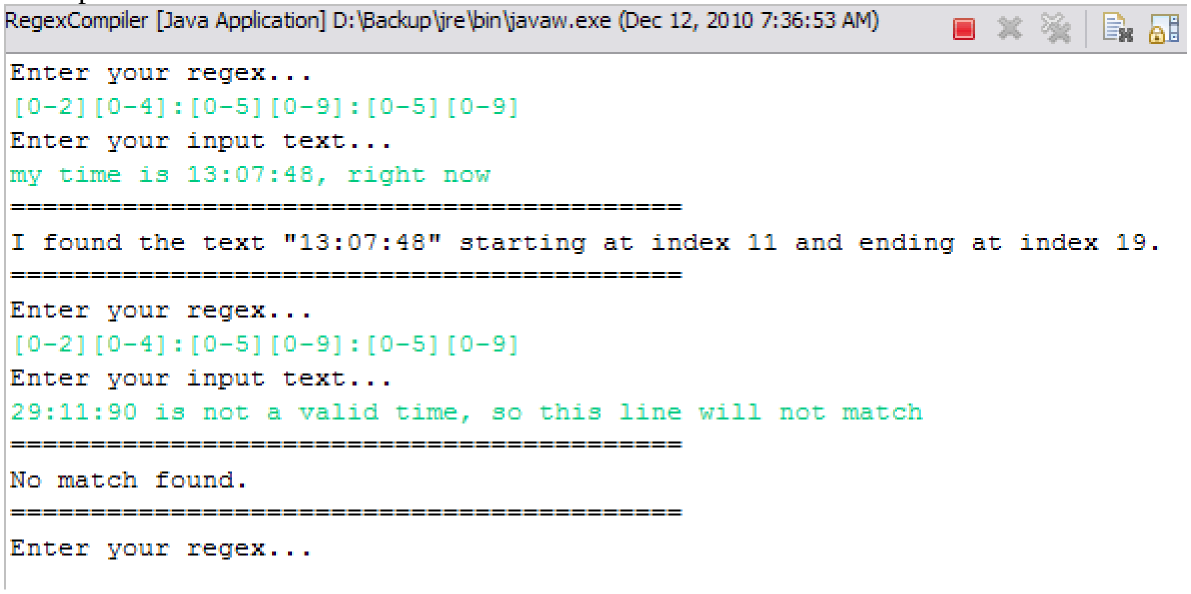
Example1: When pattern "(?:(\w)\1)+" matches "a bbccdd efg", the substring matched: "bbccdd". The match result of subpattern in "(?:)" is not recorded, so "\1" is used to refer to the match result of "(\w)".

Java Example

1. Open netbeans or any java editor
2. Create new desktop project
3. Create new class and name it “RegexCompiler”
4. Write the following code.

```java
import java.util.Scanner;
import java.util.regex.Pattern;
import java.util.regex.Matcher;

/**
 * Simple utility class that compiles a given regex pattern,
 * and try to extract the matched pattern from the given text
*/
public class RegexCompiler {

    public static void main(String[] args) {

        Scanner console = new Scanner(System.in);

        while (true) {

            System.out.println("Enter your regex...");
            String regex = console.nextLine();
            Pattern pattern = Pattern.compile(regex);

            System.out.println("Enter your input text...");
            String text = console.nextLine();
            Matcher matcher = pattern.matcher(text);

            boolean found = false;

            System.out.println("==========================================");
            while (matcher.find()) {

                System.out.format("I found the text \"%s\" starting "
                        + "at index %d and ending at "
                        + "index %d.%n", matcher.group(),
                        matcher.start(), matcher.end());
                found = true;

            }
            if (!found)
                System.out.format("No match found.%n");

            System.out.println("============================================");
        }
    }
}
```

5. Run the code.
   - This code tests for the time pattern (`[0-2][0-4]:[0-5][0-9]:[0-5][0-9]`) regular expression



```
RegexCompiler [Java Application] D:\Backup\jre\bin\javaw.exe (Dec 12, 2010 7:36:53 AM)
Enter your regex...
[0-2][0-4]:[0-5][0-9]:[0-5][0-9]
Enter your input text...
my time is 13:07:48, right now
==========================================
I found the text "13:07:48" starting at index 11 and ending at index 19.
==========================================
Enter your regex...
[0-2][0-4]:[0-5][0-9]:[0-5][0-9]
Enter your input text...
29:11:90 is not a valid time, so this line will not match
==========================================
No match found.
==========================================
Enter your regex...
```

6. Write down the results

POST-LAB

1. Write a regular expression that checks an email address (\w+?@\w+?\x2E.+)
2. Write a regular expression that checks a fax number in the form { **(990) 225623** }
3. Write a regular expression that checks an IP address `\b\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\b`
4. Write a regular expression that checks Date pattern in the form {**13-7-90 OR 13/07/1990**}
5. Write a regular expression that checks an HTML tag in the form { <title>welcome</title> }