Informatics 2

Lab 7: **Regular Expression**

Khalid Kahloot

Regular expression: a sequence of characters that define a search pattern. Usually this pattern then is used by string searching algorithms for "find" or "find and replace" operations on strings. Regular expressions are used in search engines, search and replace dialogs of word processors and text editors, in text processing utilities such as AWK and in lexical analysis. Many programming languages provide regex capabilities, built-in or via libraries.

Most formalisms provide the following operations to construct regular expressions.

Boolean "or"

A vertical bar separates alternatives. For example, `gray|gray` can match "gray" or "gray".

Grouping

Parentheses are used to define the scope and precedence of the operators (among other uses). For example, `gray|gray` and `gr(a|e)y` are equivalent patterns which both describe the set of "gray" or "grey".

Quantification

A quantifier after a token (such as a character) or group specifies how often that preceding element is allowed to occur. The most common quantifiers are the question mark `?`, the asterisk `*` and the plus sign `+` .

?	The question mark indicates *zero or one* occurrences of the preceding element. For example, `colou?r` matches both "color" and "colour".
*****	The asterisk indicates *zero or more* occurrences of the preceding element. For example, `ab*c` matches "ac", "abc", "abbc", "abbbc", and so on.
+	The plus sign indicates *one or more* occurrences of the preceding element. For example, `ab+c` matches "abc", "abbc", "abbbc", and so on, but not "ac".
{n}	The preceding item is matched exactly *n* times.
{min,}	The preceding item is matched *min* or more times.
{min,max}	The preceding item is matched at least *min* times, but not more than *max* times.

Examples:

- `a|b*` denotes {ε, "a", "b", "bb", "bbb", …}

- `(a|b)*` denotes the set of all strings with no symbols other than "a" and "b", including the empty string: {ε, "a", "b", "aa", "ab", "ba", "bb", "aaa", …}
- `ab*(c|ε)` denotes the set of strings starting with "a", then zero or more "b"s and finally optionally a "c": {"a", "ac", "ab", "abc", "abb", "abbc", …}
- `(0|(1(01*0)*1))*` denotes the set of binary numbers that are multiples of 3: { ε, "0", "00", "11", "000", "011", "110", "0000", "0011", "0110", "1001", "1100", "1111", "00000", … }

Regex Language Elements

- **Metacharacters**
  - The constructs within regular expressions that have special meaning are referred to as metacharacters
  - Characters other than . $ ^ { [(|)] } * + ? \ match themselves.

- **Character Classes**
  - Character classes are a mini-language within regular expressions, defined by the enclosing hard braces **[ ]**.

  Examples: [a-z A-Z 0-9],

Metacharacter	Description
`^`	Matches the starting position within the string. In line-based tools, it matches the starting position of any line.
`.`	Matches any single character (many applications exclude newlines, and exactly which characters are considered newlines is flavor-, character-encoding-, and platform-specific, but it is safe to assume that the line feed character is included). Within POSIX bracket expressions, the dot character matches a literal dot. For example, `a.c` matches "abc", etc., but `[a.c]` matches only "a", ".", or "c".
`[ ]`	A bracket expression. Matches a single character that is contained within the brackets. For example, `[abc]` matches "a", "b", or "c". `[a-z]` specifies a range which matches any lowercase letter from "a" to "z". These forms can be mixed: `[abcx-z]` matches "a", "b", "c", "x", "y", or "z", as does `[a-cx-z]`. The `-` character is treated as a literal character if it is the last or the first (after the `^`, if present) character within the brackets: `[abc-]`, `[-abc]`. Note that backslash escapes are not allowed. The `]` character can be included in a bracket expression if it is the first (after the `^`) character: `[]abc]`.
`[^ ]`	Matches a single character that is not contained within the brackets. For example, `[^abc]` matches any character other than "a", "b", or "c". `[^a-`

	`z]` matches any single character that is not a lowercase letter from "a" to "z". Likewise, literal characters and ranges can be mixed.
`$`	Matches the ending position of the string or the position just before a string-ending newline. In line-based tools, it matches the ending position of any line.
`( )`	Defines a marked subexpression. The string matched within the parentheses can be recalled later (see the next entry, `\n`). A marked subexpression is also called a block or capturing group. **BRE mode requires `\( \)`**.
`\n`	Matches what the *n*th marked subexpression matched, where *n* is a digit from 1 to 9. This construct is vaguely defined in the POSIX.2 standard. Some tools allow referencing more than nine capturing groups.
`*`	Matches the preceding element zero or more times. For example, `ab*c` matches "ac", "abc", "abbbc", etc. `[xyz]*` matches "", "x", "y", "z", "zx", "zyx", "xyzzy", and so on. `(ab)*` matches "", "ab", "abab", "ababab", and so on.
`{m,n}`	Matches the preceding element at least *m* and not more than *n* times. For example, `a{3,5}` matches only "aaa", "aaaa", and "aaaaa". This is not found in a few older instances of regexes. **BRE mode requires `\{m,n\}`**.
`?`	Matches the preceding element zero or one time. For example, ab?c matches only "ac" or "abc".
`+`	Matches the preceding element one or more times. For example, ab+c matches "abc", "abbc", "abbbc", and so on, but not "ac".
`|`	The choice (also known as alternation or set union) operator matches either the expression before or the expression after the operator. For example, abc|def matches "abc" or "def".

Examples:

- `.at` matches any three-character string ending with "at", including "hat", "cat", and "bat".
- `[hc]at` matches "hat" and "cat".
- `[^b]at` matches all strings matched by `.at` except "bat".
- `[^hc]at` matches all strings matched by `.at` other than "hat" and "cat".
- `^[hc]at` matches "hat" and "cat", but only at the beginning of the string or line.
- `[hc]at$` matches "hat" and "cat", but only at the end of the string or line.
- `\[.\]` matches any single character surrounded by "[" and "]" since the brackets are escaped, for example: "[a]" and "[b]".
- `s.*` matches s followed by zero or more characters, for example: "s" and "saw" and "seed".

- `[hc]?at` matches "at", "hat", and "cat".
- `[hc]*at` matches "at", "hat", "cat", "hhat", "chat", "hcat", "cchchat", and so on.
- `[hc]+at` matches "hat", "cat", "hhat", "chat", "hcat", "cchchat", and so on, but not "at".
- `cat|dog` matches "cat" or "dog".

.NET Regular Expressions
Example 1: Replacing Substrings
Assume that a mailing list contains names that sometimes include a title (Mr., Mrs., Miss, or Ms.) along with a first and last name. If you do not want to include the titles when you generate envelope labels from the list, you can use a regular expression to remove the titles, as the following example illustrates.

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = "(Mr\\.? |Mrs\\.? |Miss |Ms\\.? )";
      string[] names = { "Mr. Henry Hunt", "Ms. Sara Samuels",
                         "Abraham Adams", "Ms. Nicole Norris" };
      foreach (string name in names)
         Console.WriteLine(Regex.Replace(name, pattern, String.Empty));
   }
}
// The example displays the following output:
//    Henry Hunt
//    Sara Samuels
//    Abraham Adams
//    Nicole Norris
```

The regular expression pattern `(Mr\.? |Mrs\.? |Miss |Ms\.? )` matches any occurrence of "Mr ", "Mr. ", "Mrs ", "Mrs. ", "Miss ", "Ms or "Ms. ". The call to the Regex.Replace method replaces the matched string with String.Empty; in other words, it removes it from the original string.

Example 2: Identifying Duplicated Words

Accidentally duplicating words is a common error that writers make. A regular expression can be used to identify duplicated words, as the following example shows.

```
using System;
using System.Text.RegularExpressions;

public class Class1
```

```
{
   public static void Main()
   {
      string pattern = @"\b(\w+?)\s\1\b";
      string input = "This this is a nice day. What about this? This tastes good. I 
saw a a dog.";
      foreach (Match match in Regex.Matches(input, pattern, RegexOptions.IgnoreCase))
         Console.WriteLine("{0} (duplicates '{1}') at position {2}", 
                           match.Value, match.Groups[1].Value, match.Index);
   }
}
// The example displays the following output:
//       This this (duplicates 'This') at position 0
//       a a (duplicates 'a') at position 66
```

The regular expression pattern `\b(\w+?)\s\1\b` can be interpreted as follows:

`\b`	Start at a word boundary.
(\w+?)	Match one or more word characters, but as few characters as possible. Together, they form a group that can be referred to as `\1`.
`\s`	Match a white-space character.
`\1`	Match the substring that is equal to the group named `\1`.
`\b`	Match a word boundary.

The Regex.Matches method is called with regular expression options set to RegexOptions.IgnoreCase. Therefore, the match operation is case-insensitive, and the example identifies the substring "This this" as a duplication.

Note that the input string includes the substring "this? This". However, because of the intervening punctuation mark, it is not identified as a duplication.

Example 3: Dynamically Building a Culture-Sensitive Regular Expression

The following example illustrates the power of regular expressions combined with the flexibility offered by .NET's globalization features. It uses the NumberFormatInfo object to determine the format of currency values in the system's current culture. It then uses that information to dynamically construct a regular expression that extracts currency values from the text. For each match, it extracts the subgroup that contains the numeric string only, converts it to a Decimal value, and calculates a running total.

```
using System;
using System.Collections.Generic;
using System.Globalization;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      // Define text to be parsed.
      string input = "Office expenses on 2/13/2008:\n" +
                     "Paper (500 sheets)                      $3.95\n" +
                     "Pencils (box of 10)                     $1.00\n" +
                     "Pens (box of 10)                        $4.49\n" +
                     "Erasers                                 $2.19\n" +
                     "Ink jet printer                        $69.95\n\n" +
                     "Total Expenses                        $ 81.58\n";

      // Get current culture's NumberFormatInfo object.
      NumberFormatInfo nfi = CultureInfo.CurrentCulture.NumberFormat;
      // Assign needed property values to variables.
      string currencySymbol = nfi.CurrencySymbol;
      bool symbolPrecedesIfPositive = nfi.CurrencyPositivePattern % 2 == 0;
      string groupSeparator = nfi.CurrencyGroupSeparator;
      string decimalSeparator = nfi.CurrencyDecimalSeparator;

      // Form regular expression pattern.
      string pattern = Regex.Escape( symbolPrecedesIfPositive ? currencySymbol : "")
+
                       @"\s*[-+]?" + "([0-9]{0,3}(" + groupSeparator + "[0-9]{3})*("
+
                       Regex.Escape(decimalSeparator) + "[0-9]+)?)" +
                       (! symbolPrecedesIfPositive ? currencySymbol : "");
      Console.WriteLine( "The regular expression pattern is:");
      Console.WriteLine("    " + pattern);

      // Get text that matches regular expression pattern.
      MatchCollection matches = Regex.Matches(input, pattern,
                                              RegexOptions.IgnorePatternWhitespace);
      Console.WriteLine("Found {0} matches.", matches.Count);

      // Get numeric string, convert it to a value, and add it to List object.
      List<decimal> expenses = new List<Decimal>();

      foreach (Match match in matches)
         expenses.Add(Decimal.Parse(match.Groups[1].Value));

      // Determine whether total is present and if present, whether it is correct.
      decimal total = 0;
      foreach (decimal value in expenses)
         total += value;

      if (total / 2 == expenses[expenses.Count - 1])
         Console.WriteLine("The expenses total {0:C2}.", expenses[expenses.Count -
1]);
      else
         Console.WriteLine("The expenses total {0:C2}.", total);
   }
}
// The example displays the following output:
//       The regular expression pattern is:
//          \$\s*[-+]?([0-9]{0,3}(,[0-9]{3})*(\.[0-9]+)?)
//       Found 6 matches.
//       The expenses total $81.58.
```

On a computer whose current culture is English - United States (en-US), the example dynamically builds the regular expression `\$\s*[-+]?([0-9]{0,3}(,[0-9]{3})*(\.[0-9]+)?)`. This regular expression pattern can be interpreted as follows:

`\$`	Look for a single occurrence of the dollar symbol ($) in the input string. The regular expression pattern string includes a backslash to indicate that the dollar symbol is to be interpreted literally rather than as a regular expression anchor. (The $ symbol alone would indicate that the regular expression engine should try to begin its match at the end of a string.) To ensure that the current culture's currency symbol is not misinterpreted as a regular expression symbol, the example calls the Escape method to escape the character.
`\s*`	Look for zero or more occurrences of a white-space character.
`[-+]?`	Look for zero or one occurrence of either a positive sign or a negative sign.
`([0-9]{0,3}(,[0-9]{3})*(\.[0-9]+)?)`	The outer parentheses around this expression define it as a capturing group or a subexpression. If a match is found, information about this part of the matching string can be retrieved from the second Group object in the GroupCollection object returned by the Match.Groups property. (The first element in the collection represents the entire match.)
`[0-9]{0,3}`	Look for zero to three occurrences of the decimal digits 0 through 9.
`(,[0-9]{3})*`	Look for zero or more occurrences of a group separator followed by three decimal digits.
`\.`	Look for a single occurrence of the decimal separator.
`[0-9]+`	Look for one or more decimal digits.
`(\.[0-9]+)?`	Look for zero or one occurrence of the decimal separator followed by at least one decimal digit.

If each of these subpatterns is found in the input string, the match succeeds, and a Match object that contains information about the match is added to the MatchCollection object.

The Regex Object

Because the `DumpHRefs` method can be called multiple times from user code, it uses the `static` (`Shared` in Visual Basic) Regex.Match(String, String, RegexOptions) method. This enables the regular expression engine to cache the regular expression and avoids the overhead of instantiating a new Regex object each time the method is called. A Match object is then used to iterate through all matches in the string.

```
private static void DumpHRefs(string inputString)
{
   Match m;
   string HRefPattern = "href\\s*=\\s*(?:[\"'](?<1>[^\"']*)[\"']|(?<1>\\S+))";

   try {
      m = Regex.Match(inputString, HRefPattern,
                      RegexOptions.IgnoreCase | RegexOptions.Compiled,
                      TimeSpan.FromSeconds(1));
      while (m.Success)
      {
         Console.WriteLine("Found href " + m.Groups[1] + " at "
            + m.Groups[1].Index);
         m = m.NextMatch();
      }
   }
   catch (RegexMatchTimeoutException) {
      Console.WriteLine("The matching operation timed out.");
   }
}
```

The following example then illustrates a call to the `DumpHRefs` method.

```
public static void Main()
{
   string inputString = "My favorite web sites include:</P>" +
                        "<A HREF=\"http://msdn2.microsoft.com\">" +
                        "MSDN Home Page</A></P>" +
                        "<A HREF=\"http://www.microsoft.com\">" +
                        "Microsoft Corporation Home Page</A></P>" +
                        "<A HREF=\"http://blogs.msdn.com/bclteam\">" +
                        ".NET Base Class Library blog</A></P>";
   DumpHRefs(inputString);

}
// The example displays the following output:
```

```
//       Found href http://msdn2.microsoft.com at 43
//       Found href http://www.microsoft.com at 102
//       Found href http://blogs.msdn.com/bclteam at 176
```

The regular expression pattern `href\s*=\s*(?:["'](?<1>[^"']*)["']|(?<1>\S+))` is interpreted as shown in the following table.

Pattern	Description
`href`	Match the literal string "`href`". The match is case-insensitive.
`\s*`	Match zero or more white-space characters.
`=`	Match the equals sign.
`\s*`	Match zero or more white-space characters.
`(?:["'](?<1>[^"']*)"|(?<1>\S+))`	Match one of the following without assigning the result to a captured group: • A quotation mark or apostrophe, followed by zero or more occurrences of any character other than a quotation mark or apostrophe, followed by a quotation mark or apostrophe. The group named `1` is included in this pattern. • One or more non-white-space characters. The group named `1` is included in this pattern.
`(?<1>[^"']*)`	Assign zero or more occurrences of any character other than a quotation mark or apostrophe to the capturing group named `1`.
`"(?<1>\S+)`	Assign one or more non-white-space characters to the capturing group named `1`.

Match Result Class

The results of a search are stored in the Match class, which provides access to all the substrings extracted by the search. It also remembers the string being searched and the regular expression being used, so it can call the Match.NextMatch method to perform another search starting where the last one ended.

Review Questions:

1. Write a regular expression to match “Jone” or “John”.
2. Write a regular expression to group “Sea” or “See”.
3. Explain this regular expression: `favou?r`
4. What is the range for the following :

- ^([01][0-9][0-9]|2[0-4][0-9]|25[0-5])$
- ^([01]?[0-9]?[0-9]|2[0-4][0-9]|25[0-5])$
- ^(0?[0-9]?[0-9]|1[01][0-9]|12[0-7])$
- ^([0-9]|[1-9][0-9]|[1-9][0-9][0-9])$
- ^[0-9]{3}$
- ^[0-9]{1,3}$
- ^([1-9]|[1-9][0-9]|[1-9][0-9][0-9])$
- ^(00[1-9]|0[1-9][0-9]|[1-9][0-9][0-9])$
- ^(0{0,2}[1-9]|0?[1-9][0-9]|[1-9][0-9][0-9])$
- ^[0-5]?[0-9]$
- ^(0?[0-9]?[0-9]|[1-2][0-9][0-9]|3[0-5][0-9]|36[0-6])$

5. Write a regular expression to restricting the four IP Address Numbers without capturing them?
6. Write a regular expression for a valid data in the format yyyy-mm-dd
7. Write a regular expression to validate MasterCard numbers either start with the numbers 51 through 55 or with the numbers 2221 through 2720. All have 16 digits.

8. Explain `[hc]?at`
9. Explain `[hc]*at`
10. Write a C# code for replace (Dr. or Prof.) by “Your Highness”
11. Write a C# code for count the duplicated words in the sentence. “I live here, here I perhaps live, but here not there”