Regular Expressions

Podstawy programowania

Jolanta Bachan
2008-01-16

Zaliczenie

- Zaliczenie może dostać student, który:
  - regularnie chodził na zajęcia lub przed końcem zajęć przedstawił projekt zaliczeniowy;
  - zaliczył projekt good2bStudent.pl;
  - napisał program na zaliczenie.
- Zaliczenie odbędzie się dnia 23.01.2008.
- Jeszcze kilka osób nie przesłało programu good2bStudent. Czekam do końca tego tygodnia, tj. sobota, 19. stycznia, godz. 24.00.
- Pytania?

Proszę przynieść INDEKSY!

What Is a *Regular Expression*?

A *regular expression* is a pattern – a template – to be *matched* against a string. Matching a regular expression against a string either succeeds or fails.

Sometimes you just want to check if the string matches the *pattern*, but sometimes you want to replace the string with another string.

To save time, *regular expression* is often abbreviated as *regexp* or *regex*.

Simple Use of Regular Expressions

To find all the lines of a file that contain the string abc we may use Windows NT `findstr` command:

```
>findstr “abc” somefile > results
```

Simple Use of Regular Expressions

To find all the lines of a file that contain the string abc we may use Windows NT `findstr` command:

```
>findstr “abc” somefile > results
```

E.g.

```
>findstr “abc” words.txt > results.txt
```

where:

`abc` is the regular expression

`findsrt` is the command which tests the regular expression against each input line.

Lines that match are sent to standard output, and end up in the `results.txt` file because of the command-line redirection.

Simple Use of Regular Expressions

In Perl, we can speak of the string `abc` as a regular expression by enclosing the string in slashes:

```
if (/abc/) {
   print $_ ;
}
```

What is tested against the regular expression `abc`?

When the regular expression is enclosed in slashes, the `$_` variable is tested against the regular expression. If the regular expression matches, the *match* operator returns true and prints the `$_`. Otherwise, it returns false.

Simple Use of Regular Expressions

In Perl, we can speak of the string `abc` as a regular expression by enclosing the string in slashes:

```
if (/abc/) {
   print $_ ;
}
```

Unlike the `findstr` command, which is operating on all of the lines of a file, this Perl fragment is looking at just one line.

Simple Use of Regular Expressions

If you want to test all the lines of a fine, add a loop:

```
while (<>) {
  if (/abc/) {
    print $_ ;
  }
}
```

Simple Use of Regular Expressions

If you want to test all the lines of a fine, add a loop:

```
while (<>) {
  if (/abc/) {
    print $_ ;
  }
}
```

Remember:

“<>” is called a diamond operator. It gets its data from the file which is either specified in the command line or in the program.

Simple Use of Regular Expressions

If you do not know the number of b's between a and c, you put an asterisk “*”:

* means zero or more.

For Windows NT:

```
>findstr “ab*c” words.txt > results.txt
```

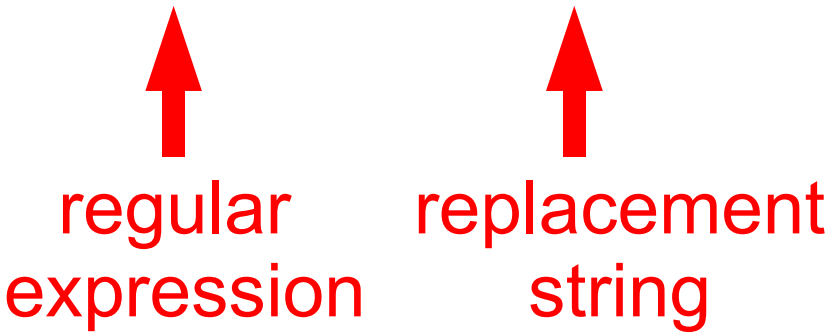
For Perl:

```
while (<>) {
   if (/ab*c/) {
      print $_ ;
   }
}
```

Substitute Operator

Substitute operator replaces the part of a string that matches the regular expression with another string.

s/ab*c/somestring/ ;



Patterns

Single-Character Patterns

- The simplest and the most common pattern-matching character is a single character that matches itself. In other words, putting a letter `a` in a regular expression requires a corresponding letter a in the string.
- The dot "`.`" matches any single character *except* newline (`\n`). E.g.

`/a./` matches any two-letter sequence that starts with a and is not a`\n`.

Single-Character Patterns: Character Class

- A pattern-matching *character class* is represented by a pair of open and close square brackets and a list of characters between the brackets. One and only one of these characters must be present at the corresponding part of the string for the pattern to match.

`/[abcde]/` matches a string containing any of the first five letters of the lowercase character, while
`/[aeiouAEIOU]/` matches any of the five vowels in either lower- or uppercase.

Single-Character Patterns: Character Class

- Ranges of characters (like a through z) can be abbreviated by showing the end points of the range separated by a dash (-). E.g.

```
[0123456789] # match any single
 digit
[0-9] # same thing
[0-9\-] # match 0-9, or minus
[a-z0-9] # match any single
 lowercase letter or digit
[a-zA-Z0-9_] # match any single
 letter, digit, or underscore
```

Single-Character Patterns: Character Class

- A leading up arrow (or caret ^) immediately after the left bracket matches any single character that is *NOT* in the list. E.g.

```
[^0-9] # match any single non-digit
[^aeiouAEIOU] # match any single
  non-vowel
[^\^] # match any single character
  except an up-arrow
```

Single-Character Patterns: Character Class

- A leading up arrow (or caret ^) immediately after the left bracket matches any single character that is *NOT* in the list. E.g.

```
[^0-9] # match any single non-digit
[^aeiouAEIOU] # match any single
  non-vowel
[^\^] # match any single character
  except an up-arrow
```

Remember to `chomp` the string.

Single-Character Patterns: Predefined Character Class Abbreviations

Construct	Equivalent Class	Negated Construct	Equivalent Negated Class
\d (a digit)	[0-9]	\D (digits, not!)	[^0-9]
\w (word char)	[a-zA-Z0-9_]	\W (words, not!)	[^a-zA-Z0-9_]
\s (space char)	[\t\n]	\S (space, not)	[^ \t\n]

Grouping Patterns

By grouping patterns we understand regular expressions which mean “one or more of these” or “up to (three, four, five...) of those”, etc.

Grouping Patterns: Sequence

`/abc/` matches a followed by b followed by c.

Grouping Patterns: Multipliers

* indicates zero or more of the immediately previous character (or character class). E.g.

```
/ab*c/
/[abc]*/
```

+ indicates one or more of the immediately previous character. E.g.

`/ab+c/` means a followed by one or more b's, followed by c

? indicates zero or one of the immediately previous character. E.g.

`/fo+ba?r/` matches an f, followed by one or more o's, followed by b, followed by an optional a, followed by an r.

Grouping Patterns: Multipliers

* `*` indicates zero or more
* `+` indicates one or more
* `?` indicates zero or one

Exercise: Write 4 different strings which will be matched by `/h*ugg?iez+/`.

Grouping Patterns: Multipliers

* indicates zero or more
+ indicates one or more
? indicates zero or one

In all three of these grouping patterns, the patterns are *greedy*. If such a multiplier has a chance to match between five and ten characters, it'll pick the ten-character string every time. E.g.

```
$_ = “fred xxxxxxxxxx barney” ;
s/x+/boom/ ;
```

Result: `fred boom barney`

Grouping Patterns: Multipliers

The term "*greedy*" is usually referred to as "*the longest match principle*".

Example:

```
$ = "a xxx c xxxxxxxx c xxx d" ;
/a.*c.*d/ ;
```

Question: What does the first `.*` match?

Grouping Patterns: Multipliers

The term "*greedy*" is usually referred to as "*the longest match principle*".

Example:

```
$ = "a xxx c xxxxxxxx c xxx d" ;
/a.*c.*d/ ;
```

Question: What does the first `.*` match?

Answer: It matches all the characters up to the second `c`, even though matching only the characters up to the first `c` would still allow the entire regular expression to match.

Grouping Patterns: Multipliers

To force any multiplier (`*`, `+`, `?` and `{m,n}`) to be nongreedy (or *lazy*), follow it by a question mark.

Example:

```
$ = “a xxx c xxxxxxxx c xxx d” ;
/a.*?c.*d/ ;
```

Question: What does the `.*?` match?

Grouping Patterns: Multipliers

To force any multiplier (`*`, `+`, `?` and `{m,n}`) to be nongreedy (or *lazy*), follow it by a question mark.

Example:

```
$ = “a xxx c xxxxxxxx c xxx d” ;
/a.*?c.*d/ ;
```

Question: What does the `.*?` match?

Answer: `.*?` matches the fewest characters between the a and c, not the most characters.

Grouping Patterns: Multipliers

To force any multiplier (`*`, `+`, `?` and `{m,n}`) to be nongreedy (or *lazy*), follow it by a question mark.

Example:

```
$ = “a xxx c xxxxxxxx c xxx d” ;
/a.*?c.*d/ ;
```

Another way of formulating `/a.*?c.*d/` is `/a[^c|\n]*c.*d/`.

Grouping Patterns: Multipliers

Example:

```
$ = “a xxx ce xxxxxxxx ci xxx d” ;
/a.*ce.*d/ ;
```

If the `.*` matches the most characters possible before the next `c`, the next regular expression character (`e`) does not match the next character of the string (`i`). In this case, we get automatic *backtracking*. The multiplier is unwounded, stopping at some place earlier.

Grouping Patterns: Multipliers: general multiplier

The *general multiplier* consists of a pair of matching curly brackets with one or two numbers inside, as in `/x{5,10}/`. The immediately preceding character (here “x”) must be found within the indicated number of repetitions (five through ten)

- Leaving off the second number means “that many or more”, e.g. `/x{5,}/` - five x's or more
- Leaving off the comma means “exactly this many”, e.g. `/x{5}/` - five x's
- To get some amount or fewer, you must put the zero in, e.g. `/x{0,5}/` - five or fewer x's

Grouping Patterns: Multipliers: general multiplier

In other words,

* `*` corresponds to `{0,}`
* `+` corresponds to `{1,}`
* `?` corresponds to `{0,1}`

Grouping Patterns: Parentheses as memory

- The *parentheses as memory* construct does not change the way an expression matches but causes the enclosed text part to be remembered, so that it may be referred to later on in the expression.
- To recall the memorised part of a string, you must include a backslash followed by an integer. This pattern construct represents the same sequence of characters matched earlier in the same-numbered pair of parentheses (counting from one).

Grouping Patterns: Parentheses as memory

- For example:

```
/fred(.)barney\1/ ;
```

matches a string consisting of `fred`, followed by any single non-newline character, followed by `barney`, followed by the same single character. So, the matching string is `fredxbarneyx`, but not `fredxbarneyy`. Compare that with `/fred.barney./`

Grouping Patterns: Parentheses as memory

- For example:

```
/fred(.)barney\1/ ;
```

matches a string consisting of `fred`, followed by any single non-newline character, followed by `barney`, followed by the same single character. So, the matching string is `fredxbarneyx`, but not `fredxbarneyy`. Compare that with `/fred.barney./`

`1` indicates the first parenthesised part of the regular expression.

Grouping Patterns: Parentheses as memory

`1` indicates the first parenthesised part of the regular expression. If there is more than one, the second part (counting from the left) is referenced as `\2`, third as `\3`, and so on.

Example:

`/a(.)b(.)c\2d\1/` matches `axbycydx`

Grouping Patterns: Parentheses as memory

`1` indicates the first parenthesised part of the regular expression. If there is more than one, the second part (counting from the left) is referenced as `\2`, third as `\3`, and so on.

Example:

`/a(.)b(.)c\2d\1/` matches `axbycydx`

Exercise: Give another example of a sting which is matched by this regular expression.

Grouping Patterns: Parentheses as memory

The referenced part may be more than a single character.

Example:

`/a(.*)b\1c/`

matches an `a`, followed by any number of characters (even zero), followed by `b`, followed by that same sequence of characters, followed by `c`. So the string would match `aFREDbFREDc`, but not `aXXbXXXc`.

Grouping Patterns: Parentheses as memory

The referenced part may be more than a single character.

Example:

`/a(.*)b\1c/`

matches an `a`, followed by any number of characters (even zero), followed by `b`, followed by that same sequence of characters, followed by `c`. So the string would match `aFREDbFREDc`, but not `aXXbXXXc`.

Exercise: Give another example of a string which is matched by this regular expression.

Grouping Patterns: Alternation

Alternation, as in `a|b|c` matches exactly one of the alternatives (a or b or c, in this case).

Example:

`/song|blue/` matches either `song` or `blue`.

Anchoring patterns

The word anchor \b matches a boundary between a word character and a non-word character \w\W or \W\w :

```
$_ = "Housecat catenates house and cat";
/cat/;    # matches cat in “housecat”
/\bcat/;  # matches cat in “catenates”
/cat\b/;  # matches cat in “housecat”
/\bcat\b/;  # matches “cat” at end of string
```

Note in the last example, the end of the string is considered a word boundary.

Anchoring patterns

`\B` requires that there is no a word boundary at the indicated point :

```
/\bFred\B/;  # matches “Frederick”,
  but not “Fred Flinstone”
```

Anchoring patterns

The caret (^) matches the beginning of the string.
The $ matches the end of the string.

```
$_ = "housekeeper" ;
/keeper/;     # matches
/^keeper/;    # doesn't match
/keeper$/;    # matches

$_ = "housekeeper\n"
/keeper$/;    # matches
```

Anchoring patterns

When both `^` and `$` are used at the same time, the regexp has to match both the beginning and the end of the string, i.e., the regexp matches the whole string.

```
$_ = "keeper" ;
/^keep$/;   # doesn't match
/^keeper$/; # matches

$_ = "" ;
/^$/;  # ^$ matches an empty string
```

The =~ Operator

The operator =~ associates the string with the regexp match and produces a true value if the regexp matched, or false if the regexp did not match.

```
"Hello World" =~ /World/;  # matches

$a = "hello, world" ;
$a =~ /^he/ ; true
$a =~ /(.)\1/ ; true (matches the doubble l)

if ($a =~ /(.)\1/) { # true, so
   print “It matches!” ;
}
```

The !~ Operator

The !~ operator has the reversed function of the =~.

```
if ("Hello World" !~ /World/) {
    print "It doesn't match\n";
} else {
    print "It matches\n";
}
```

Ignoring case

`i` means “ignore case”. `i` is appended to the closing slash of the regexp.

```
/somepattern/i ;

s/oldstring/newstring/i ;
```

Substitution

Substitute operator replaces the part of a string that matches the regular expression with another string.

s/regexp/somestring/ ;

Substitution

If you want the replacement to operate on all possible matches instead of just the first match, append a g to the substitution. For example:

```
$_ = “foot fool buffoon” ;
s/foo/bar/g ;  # $_ is now “bart
  barl bufbarn”
```

Substitution

If you want the replacement to operate on all possible matches instead of just the first match, append a g to the substitution. For example:

```
$_ = “foot fool buffoon” ;
s/foo/bar/g ;  # $_ is now “bart
  barl bufbarn”
```

Substitution

Example:

```
$_ = “Hello, world” ;
$new = “Goodbye” ;
s/Hello/$new/ ; # replaces “Hello”
  with “Goodbye”
```

Substitution

More examples:

```
$which = “this is a test” ;
$which =~ s/test/quiz/ ; # $which is
  now “this is a quiz”

$someplace[$here] =~ s/left/right/ ;
```

A construct similar to s/// Transliteration: tr Operator

The tr operator takes two arguments: an old string and a new string, and looks for the characters of the old string and replaces them with the corresponding characters in the new string. For example:

```
$x = “Linguistics and Information Science”;
$x =~ tr/A-Z/a-z/ ; # $x is lowercased
```

The `split` Function

The `split` function separates the string operand into a list of substrings and returns that list. The regexp must be designed to match whatever constitutes the separators for the desired substrings.

```
$x = "Fred and Barney";
@words = split (/\s+/, $x);
   # $word[0] = “Fred”
   # $word[1] = “and”
   # $word[2] = “Barney”
```

where `/\s+/` matches one or more whitespace characters together

The `split` Function

The `split` function separates the string operand into a list of substrings and returns that list. The regexp must be designed to match whatever constitutes the separators for the desired substrings.

```
$x = "Fred and Barney";
($first, $second, $third) = split(/\s+/,$x);

   # $first = “Fred”
   # $second = “and”
   # $third = “Barney”
```

The `join` Function

The `join` function takes a list of values and glues them together with a glue string between each list element.

```
$longstring = join($glue, @list) ;
```

The `join` Function

The `join` function takes a list of values and glues them together with a glue string between each list element.

For example, rebuilt the sentence “Fred and Barney”

```
$sentence = join (“ ”,  @words) ;
```

Exercise

- Construct a regular expression that matches:
  - at least one a followed by any number of b's
  - any number of backslashes followed by any number of asterisks (any number might be zero). (Hint: Remember of the special feature of a *backslash* which makes special characters neutral.)
  - Three consecutive copies of whatever is contained in `$whatever`.
  - Any five characters, including newline.

MakeDictionary.pl Project

Make a dictionary

1 - ...
2 - ...
3 - ...
4 - ?

Make a dictionary

1 - Input text
2 - ...
3 - ...
4 - ?

Make a dictionary

1 - Input text
2 - Lowercase everything
3 - ...
4 - ?

Make a dictionary

1 - Input text
2 - Lowercase everything
3 - Remove punctuation marks
4 - ?

Make a dictionary

1 - Input text
2 - Lowercase everything
3 - Remove punctuation marks
4 - Normalise two spaces and more by one space
5 - ?

Make a dictionary

1 - Input text
2 - Lowercase everything
3 - Remove punctuation marks
4 - Normalise two spaces and more by one space
5 - Produce a list
6 - ?

Make a dictionary

1 - Input text
2 - Lowercase everything
3 - Remove punctuation marks
4 - Normalise two spaces and more by one space
5 - Produce a list
6 - Sort the words alphabetically
7 - ?

Make a dictionary

1 - Input text
2 - Lowercase everything
3 - Remove punctuation marks
4 - Normalise two spaces and more by one space
5 - Produce a list
6 - Sort the words alphabetically
7 - Remove the words which appear twice
8 - ?

Make a dictionary

1 - Input text
2 - Lowercase everything
3 - Remove punctuation marks
4 - Normalise two spaces and more by one space
5 - Produce a list
6 - Sort the words alphabetically
7 - Remove the words which appear twice
8 - Count the number of units
9 - ?

Make a dictionary

1 - Input text
2 - Lowercase everything
3 - Remove punctuation marks
4 - Normalise two spaces and more by one space
5 - Produce a list
6 - Sort the words alphabetically
7 - Remove the words which appear twice
8 - Count the number of units
9 - Print the list with the frequency information

Make a dictionary

Let's start with creating a new file, first... :-)

Homework

- Modify the MyDictionary.pl program in such a way that it takes a text file as the input, and gets the information about which file to open from the standard input.
- Prepare for the test!

Prepare for the test!
&
See you next week!