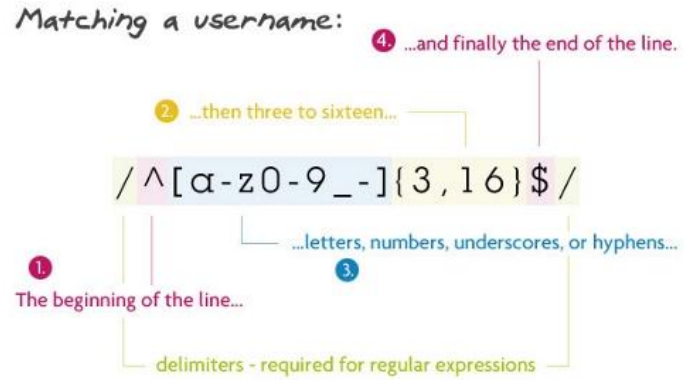
COMP 2021

Unix and Script Programming



Regular Expression

What is a Regular Expression?

- A *regular expression* is a string of characters (or pattern) to be matched.
- Matching either succeeds or fails.
- You can also replace a matched pattern with another string.
- Regular expressions are used by many other Unix commands and programs, such as `grep`, `sed`, `awk`, `vi`, `emacs`, and even some shells.

Regular Expression Operators within Perl

- Match regular expression: `m/PATTERN/`
- Substitute regular expression: `s/PATTERN/REPLACEMENT/`
- Translate regular expression: `tr/SEARCHLIST/REPLACEMENT/`

- The forward slashes (/) act as delimiters for regular expression (regex)
- You can use any other delimiter in place of /

Basic Matching and Substitution

```
$ regular_helloworld.pl
#!/usr/local/bin/perl5 -w
my $string = "Hello world!";
print "Original string: $string\n";

print "Does the string contain \"World\"?\n";
if ($string =~ m/World/) {print "Yes\n";}
else {print "No\n";}

print "Doe the string contain \"World\", ignoring case?\n";
if ($string =~ m#World#i) {print "Yes\n";}
else {print "No\n";}

print "Change to \"Hello mom!\"\n";
$string =~ s/world/mom/;
print "Updated string: $string\n";
```

`m//` is match operator,

Regular expression operator works with `$_` by default, binding operator `=~`, `!~` let them work with other variables

Use a different pattern delimiter, can be any non-alphanumeric character

Note if a match is not successful, nothing happens

Binding Operator =~

- An example using =~ to match <STDIN>:

```
$ regular_quit.pl
#!/usr/local/bin/perl5 -w
while (1){
        print "Quit (y/n)?\n";
        if (<STDIN> =~ /^[yY]/){
                print "Quitting...\n";
                exit;
        }
        print "Continuing\n";
}
```

Output:

```
Quit (y/n)? y
Quitting
```

Match Operator Modifiers

- The match operator supports its own set of modifiers

Modifier	Description
i	Makes the match case insensitive
m	Specifies that if the string has newline or carriage return characters, the ^ and $ operators will now match against a newline boundary, instead of a string boundary
o	Evaluates the expression only once
s	Allows use of . to match a newline character
x	Allows you to use white space in the expression for clarity
g	Globally finds all matches

Slash and Backslash

- If your pattern has a slash character (/), you must precede each with a backslash (\):
- You can use other delimiters if pattern contains lots of /

```
$ regular_slash.pl
#!/usr/local/bin/perl5 -w
print "Enter path:\n";
$searchpath = <STDIN>;
if ($searchpath =~ m#\/usr\/local\/bin#){
        print "Path is /usr/local/bin \n";
}
else{
        print "Path doesn't match.\n";
}
```

Basic Translation

- tr/ SEARCHLIST/REPLACEMENTLIST

Translation (or transliteration) replaces all occurrences of characters in SEARCHLIST with the corresponding characters in REPLANCEMENTLIST

```
$ cat regular_tr.pl
#!/usr/local/bin/perl5 -w
my $string = "The cat sat on the mat";
$string =~ tr/a/o/;
print "$string\n";
```

Output: `The cot sot on the mot`

More complex Regular Expressions

- You don't just have to match on fixed strings
- You can match on just anything by using complex regular expressions

- Example: matching an email

```
/^([a-z0-9_\.-]+)@([\da-z\.-]+)\.([a-z\.]{2,6})$/
```

Use of $_ in Regular Expression

- Search in a movie file

```
$cat movie
Titanic
Saving Private Ryan
Shakespeare in Love
Life is Beautiful
```

- Check whether "Shakespeare" is in the file

```
while(<>){
                if(/Shakespeare/){ print;}
}
```

`$_` is tested in if

- I'm not quite sure how to spell Shakespeare, it starts with Shak..

```
if (/Shak.*/)
```

`.*` means zero or more of any characters

Single-Character Patterns

- The dot "`.`" matches any single character except the newline (`\n`)
  - `/a./` matches any two-letter sequence that starts with `a` and is not "`a\n`".
- Use `\.` to match the period
- Specify one out of a group of characters to match use [ ]
  - `/[abcde]/` matches a string containing any one of the first 5 lowercase letters
  - `/[aeiouAEIOU]/` matches any of the 5 vowels in either upper or lower case.
- Use – for ranges of characters (like `a` through `z`)
  - `/[0123456789]/` matches any single digit
  - `/[0-9]/` is the same
- Backslash] or – if you want them in the list, put a backslash before it
  - `/[X\-Z]/` matches X, -, Z

Single-Character Patterns (cont.)

- More range examples:
  - `/[0-9\-]/` matches 0-9, or minus
  - `/[0-9a-z]/` matches any digit or lowercase letter
  - `/[a-zA-Z0-9_]/` matches any letter, digit, underscore

- Negated character group starts with a ^ immediately after the left bracket. This matches any single character not in the list.
- `/[^0123456789]/` matches any single non-digit
- `/[^0-9]/` same as above
- `/[^aeiouAEIOU]/` matches any single non-vowel
- `/[^\^]/` matches any single character except ^

Single-Character Groups

- For convenience, some common character groups are predefined:

PredefinedGroup	Negated	Negated Group
\d (a digit)	[0-9]	\D (non-digit)	[^0-9]
\w (word char)	[a-zA-Z0-9_]	\W (non-word)	[^a-zA-Z0-9_]
\s (space char)	[\t\n]	\S (non-space)	[^ \t\n]

  - \d matches any digit
  - \w matches any letter, digit, underscore
  - \s matches any space, tab, newline
- You can use these predefined groups in other groups:

```
/[\da-fA-F]/     # match any hexadecimal digit
```

Multipliers

- Multipliers allows you to say "one or more of these" or "up to four of these"
  - `*` zero or more of the immediately previous character (or character group).
  - `+` one or more of the immediately previous character (or character group).
  - `?` means zero or one of the immediately previous character (or character group).
- `*`, `+`, and `?` are greedy, and will match as many characters as possible

Repetition Examples

`/x{5,10}/` five to ten `x`'s

`/x{5,}/` five or more `x`'s

`/x{5}/` exactly five `x`'s"

`/x{0,5}/` up to five `x`'s"?

`/c.{5}d/` `c` followed by any 5 characters (which can be different) and ending with `d`

- `*` is the same as `{0,}`
- `+` is the same as `{1,}`
- `?` is the same as `{0,1}`

Alteration

- Picking from a set of alternatives when there is more than one character in the patterns.
- `/Gates|Clinton|Shakespeare/` matches either `Gates` or `Clinton` or `Shakespeare`
- For single character alternatives,

  `/[abc]/`

  is the same as

  `/a|b|c/`

Anchoring Patterns

- Anchors requires that the pattern be at the beginning or end of the line.
- ^ matches the beginning of the line (only if ^ is the first character of the pattern):

```
/^COMP/   # match lines that begin with COMP
/COMP\^/  # match lines containing COMP^ somewhere
/\^/      # match lines containing ^
```

- $ matches the end of the line (only if $ is the last character of the pattern):

```
/2021$/   # match lines that end with 2021
/$2021/   # match with contents of scalar $2021
/\$/      # match lines containing $
```

Summary: Metacharacters and Repetition

Metacharacters

char	meaning
^	beginning of string
$	end of string
.	any character except newline
*	match 0 or more times
+	match 1 or more times
?	match 0 or 1 times; *or*: shortest match
|	alternative
( )	grouping; "storing"
[ ]	set of characters
{ }	repetition modifier
\	quote or special

To present a metacharacter as a data character standing for itself, precede it with \ (e.g. \. matches the full stop character . only).

Repetition

Quantifiers

*a**	zero or more *a*'s
a+	one or more *a*'s
a?	zero or one *a*'s (i.e., optional *a*)
a{*m*}	exactly *m* *a*'s
a{*m*,}	at least *m* *a*'s
a{*m*,*n*}	at least *m* but at most *n* *a*'s
repetition?	same as *repetition* but the *shortest* match is taken

Read the notation *a*'s as "occurrences of strings, each of which matches the pattern *a*". Read *repetition* as any of the repetition expressions listed above it. Shortest match means that the shortest string matching the pattern is taken. The default is "greedy matching", which finds the longest match. The *repetition*? construct was introduced in Perl version 5.

Some Examples

Examples

expression	matches...
abc	abc (that exact character sequence, but anywhere in the string)
^abc	abc at the *beginning* of the string
abc$	abc at the *end* of the string
a|b	either of a and b
^abc|abc$	the string abc at the beginning or at the end of the string
ab{2,4}c	an a followed by two, three or four b's followed by a c
ab{2,}c	an a followed by at least two b's followed by a c
ab*c	an a followed by any number (zero or more) of b's followed by a c
ab+c	an a followed by one or more b's followed by a c
ab?c	an a followed by an optional b followed by a c; that is, either abc or ac
a.c	an a followed by any single character (not newline) followed by a c
a\.c	a.c exactly
[abc]	any one of a, b and c
[Aa]bc	either of Abc and abc
[abc]+	any (nonempty) string of a's, b's and c's (such as a, abba, acbabcacaa)
[^abc]+	any (nonempty) string which does *not* contain any of a, b and c (such as defg)
\d\d	any two decimal digits, such as 42; same as \d{2}
\w+	a "word": a nonempty sequence of alphanumeric characters and low lines (underscores), such as foo and 12bar8 and foo_1
100\s*mk	the strings 100 and mk optionally separated by any amount of white space (spaces, tabs, newlines)
abc\b	abc when followed by a word boundary (e.g. in abc! but not in abcd)
perl\B	perl when *not* followed by a word boundary (e.g. in perlert but not in perl stuff)

Pattern Memory

- How would we match a pattern that starts and ends with the same letter or word
- For this, we need to remember the pattern.
- Use ( ) around any pattern to put that part of the string into memory (it has no effect on the pattern itself)
- To recall memory, use backslash with an integer

Pattern Memory Example

- `/Bill(.)Gates\1/`

  Matches a string starting with `Bill`, followed by any single non-newline character, followed by `Gates`, followed by that same single character.

  matches: `Bill!Gates!` `Bill-Gates-`

  does not match: `Bill?Gates!` `Bill-Gates_`

  note that `/Bill.Gates./` would match all four

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

  matches a string starting with a, a character (#1), followed by b, another single character (#2), c, the character #2, d, and the character #1.

  matches: `a-b!c!d-`.

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

  matches an `a`, followed by any number of characters (even zero), followed by `b`, followed by the same sequence of characters, followed by `c`.

  matches: `aBillbBillc` and `abc`

  does not match `aBillbBillGatesc`.

Special Read-Only Variables

- After a successful pattern match, the variables `$1`, `$2`, `$3`,… are set to the same values as `\1`, `\2`, `\3`,…
- You can use `$1`, `$2`, `$3`,… later in your program.

```
$_ = "Shakespeare     in Love";
/(\w+)\W+(\w+)/; # match first two words
# $1 is now “Shakespeare" and $2 is now “in"
print "The first words are: $1, $2\n";

$date = “12/31/1999”;
$date =~ s#(\d+)/(\d+)/(\d+)#$3/$1/$2#;
print "$date"; # 1999/12/31
```

Other Read-Only Variables

- `$&` is the part of the string that matched the pattern.
- `$`` is the part of the string before the match
- `$'` is the part of the string after the match

```
$string = "The food is in the salad bar";
$string =~ m/foo/;
print "Before: $`\n";
print "Matched: $&\n";
print "After: $'\n";
```

output:

```
Before: The
Matched: foo
After: d is in the salad bar
```

Pattern Memory Example

```
$ cat regular_memory.pl
#!/usr/local/bin/perl5 -w
print 'Match pattern /Bill(.)Gates\1', "\n";
$_ = "Bill!Gates!";
print;
if (/Bill(.)Gates\1/) {print ": match\n";}
else {print ":does not match\n"; }
$_ = "Bill?Gates!";
print;
if (/Bill(.)Gates\1/) {print ": match\n";}
else {print ":does not match\n";}
$_ = "AAA BBB CCC";
print "String: AAA BBB CCC\n";
print 'Match with m/(\w+)', "\n";
m/(\w+)/;
print "$1\n";
print 'Match with m/(\w+)/g', "\n";
@matches = m/(\w+)/g;
print "@matches\n";
```

Output:

```
Match pattern /Bill(.)Gates\1
Bill!Gates!: match
Bill?Gates!:does not match
String: AAA BBB CCC
Match with m/(\w+)
AAA
Match with m/(\w+)/g
AAA BBB CCC
```

g option replaces all matches instead of just the first match

Precedence

- What happens with the pattern: `a|b*`
- Is this `(a|b)*` or

  `a|(b*)` ?
- Precedence of patterns from highest to lowest

Name	Representation
Parentheses	( )
Multipliers	? + * {m,n}
Sequence & anchoring	abc ^ $
Alternation	|

- By the table, `*` has higher precedence than `|`, so it is interpreted as `a|(b*)` .

Precedence

- Use parentheses if want the other interpretation in the previous example?
- Example: `(a|b)*`

- Use parentheses in ambiguous cases to improve clarity, even if not strictly needed.
- When you use parentheses for precedence, they also go into memory (`\1`, `\2`, `\3`).

More Precedence Examples

```
abc*          # matches ab, abc, abcc, abccc,…
(abc)*        # matches "", abc, abcabc, abcabcabc,…
^a|b          # matches a at beginning of line, or b
              anywhere
^(a|b)        # matches either a or b at the beginning of
              line
a|bc|d        # a, or bc, or d
(a|b)(c|d)    # ac, ad, bc, or bd
```

split

- The `split` function allows you to cut up a string into pieces (explode)
- Syntax `Split REGEX, STRING`
- Split the `STRING` at every match of the `REGEX`
- Works with `$_` if no `STRING` is given
- `split` defaults to look for space delimiters

```
$_ = “banana apple peach mango";
@fields = split;
# split $_ using space (default) as delimiter
```

split Example

```
$cat regular_split.pl
#!/usr/local/bin/perl5 -w

print "split with white space\n";
my $string = "ab cd ef gh ij";
my @strings = split / /, $string;



print "split with multiple patterns:\n";
$string= "10:10:10, 12/1/2011";
@strings = split /[:,\s\/]+/, $string;
#split with :, or ,, or any types of space character \s, and
#forward slashes \/
# + is needed to match on 1 or more of the character immediately
#proceding it
```

Output:

```
split with white space
ab
cd
ef
gh
ij
split with multiple patterns:
10:10:10, 12/1/2011
After splitting:
10
10
10
12
1
2011
```

Quick Quiz

```
# extract hours, minutes, seconds
if ($time =~ /(\d{2}):(\d{2}):(\d{2})/) {
        $hours = $1;
        $minutes = $2;
        $seconds = $3;
}
```

What does this match? `^[ \t]+`

How to match a floating point number? Integers or floating point number without integer part should be matched too. (+3.14159, 2, .618, -1.5)

Is this correct `[-+]?[0-9]*\.?[0-9]*`

`[-+]?([0-9]*\.[0-9]+|[0-9]+)`

`[-+]?[0-9]*\.?[0-9]+`

`^[-+]?[0-9]*\.?[0-9]+$`