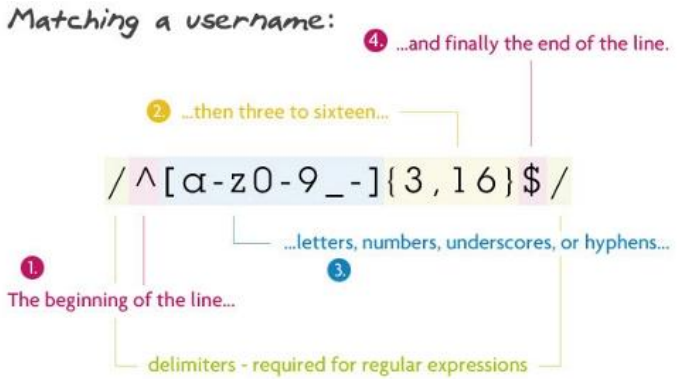
COMP 2021

Unix and Script Programming

Regular Expression



What is a Regular Expression?

- A *regular expression (regex)* describes a set of possible input strings
- Descend from a fundamental concept in Computer Science called *finite automata theory*
- Regular expressions are endemic to Unix
  - Shell, vim, emacs
  - awk, sed, grep
  - Perl and Python
  - Compilers

Simple Regular Expressions

- The simplest regular expressions are a string of literal characters to match
- The string *matches* the regular expression if it contains the substring
- Example: regular expression: `c k s`
  - String 1 matches: Unix Tools ro**cks**.
  - String 2 matches: Unix Tools su**cks**.
  - String 3 no match: Unix Tools is Okay.
- Regular expression can match a string in more than one place.
- Example: regular expression: `a p p l e`
  - String: Scr**apple** from the **apple**.

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})$
```

Single Character Pattern

- The **.** regular expression matches any single character except the newline (`\n`)
  - Example: regular expression

o	.	

  - String: F**or** me to po**op** on
- Character class [ ] can be used to match any specific set of characters.
  - Example: regular expression

b	[eor]	a	t

  - String: **beat** a **brat** on a **boat**
  - `[aeiouAEIOU]` matches any of the 5 vowels in either upper or lower case.
- Character class can be negated with the [^] syntax
  - Example: regular expression

b	[^eo]	a	t

  - String: beat a **brat** on a boat

Single-Character Pattern (cont.)

- Use – for range 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
    - `[X\-Z]` matches X, -, Z

- 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
  - `[^0123456789]` matches any single non-digit
  - `[^0-9]` same as above
  - `[^aeiouAEIOU]` matches any single non-vowel
  - `[^\^]` matches any single character except ^

Named Character Classes

- Commonly used character classes can be referred to by name:

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

Anchors

- Anchors are used to match beginning or end of the line (or both)
- ^ means beginning of the line
- $ means end of the line

  - Example: regular expression

^	b	[eor]	a	t

  - String: **beat** a brat on the boat
  - Example: regular expression

b	[eor]	a	t	$

  - String: beat a brat on the **boat**

Repetition

- The * is used to define zero or more occurrences of the single regular expression preceding it

- Example: regular expression

y	a	*	y

- String: I got a mail, **yaaaaaaaaaaay!**

- Example: regular expression

o	a	*	o

- String: For me to p**oo**p on.

Repetition Ranges

- Ranges can also be specified
  - `{ }` notation can specify a range of repetitions for the immediately preceding regex
  - `{n}` means exactly n occurrences
  - `{n,}` means at least n occurrences
  - `{n,m}` means at least n occurrences but no more than m occurrences

- Example
  - `x{5,10}` five to ten x's
  - `x{5}` exactly five x's
  - `x{5,}` five or more 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
  - `*` same as {0,}

Subexpressions

- If you want to group part of an expression so that * or { } applies to more than just the previous character, use ( ) notation
- Subexpresssion is treated like a single character
- Example:
  - `a*` matches 0 or more occurrences of a
  - `abc*` matches ab, abc, abcc, abccc, …
  - `(abc)*` matches abc, abcabc, abcabcabc, …
  - `(abc){2,3}` matches abcabc or abcabcabc

Escaping Special Characters

- Even though we are single quoting our regexs so the shell won't interpret the special characters, some characters are special *to grep* (eg * and .)
- To get literal characters, we *escape* the character with a \ (backslash)
- Example: search `'a*b*'`
  - Unless we do something special, this will match zero or more 'a's followed by zero or more 'b's, not what we want
  - `'a\*b\*'` will fix this

Protecting Regex Metacharacters

- Since many of the special characters used in regexs also have special meaning to the shell, it's a good idea to get in the habit of single quoting your regexs
- This will protect special characters from being operated on by the shell
  - Single quote `''` : take the string as is

grep

- **grep** comes from the *ed* (Unix text editor) search command "global regular expression print"
- This was such a useful command that it was written as a standalone utility
- grep is the answer to the moments where you know you want the file that contains a specific phrase but you can't remember its name

grep, fgrep, egrep

- **`grep`** uses regular expressions for pattern matching
- **`fgrep`** file grep, does not use regular expressions, only matches fixed strings but can get search strings from a file
- **`egrep`** extended grep, uses a more powerful set of regular expressions but does not support backreferencing
  - Acronym: extended global regular expressions print
  - `egrep` = `grep -E` (extended regular expressions, which treats `+`, `?`, `|`, `(`, and `)` as meta-characters

egrep: 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

Metacharacters and Repetition

Quantifiers: specify how many instances of a character, group, or character class must be present in the input for a match to be found.

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

*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.

egrep: Alteration

- Alternation character | for matching one or more subexpression
  - `(b|c)at` matches 'bat' or 'cat'
  - `^(From|Subject):` matches the From and Subject lines of a typical email message
- For single character alternatives, `[abc]` is the same as `(a|b|c)`
- Subexpressions are used to limit the scope of the alternation
  - `At(ten|nine)tion` matches "Attention" or "Atninetion", not "Atten" or "ninetion" as would happen without the parenthesis - `Atten|ninetion`

grep: 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, you can backreference using backslash with an integer
  - `\n` is the backreference specifier, where *n* is a number
  - Looks for *n*th subexpression

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`.

Practical Regex Examples

- Variable names in C
  - `[a-zA-Z_][a-zA-Z_0-9]*`
- Dollar amount with optional cents
  - `\$[0-9]+(\.[0-9][0-9])?`
- Time of day
  - `(1[012]|[1-9]):[0-5][0-9] (am|pm)`
- HTML headers <h1> <H1> <h2> …
  - `<[hH][1-4]>`

More 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)

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	|

- Use parentheses
  - If want the other interpretation
  - 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`)

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
```

Fun with Dictionary

- `/usr/share/dict/words` contains about 48,000 words (in CSE lab 2 machine)
  - `grep '^.a...x.$' /usr/share/dict/words`
  - `grep '^\(.*\)\1$' /usr/share/dict/words`

```
cachexy
carboxy
martext
panmixy
```

- egrep as a simple spelling checker, specify plausible alternatives you know
  - `egrep 'n(ie|ei)ther' /usr/share/dict/words`
- How many words have 3 a's one letter apart?
  - `egrep 'a.a.a' /usr/share/dict/words | wc -l`
- Palindrome?
  - Find out all 4-letter palindromes
  - How about 5-letter palindromes

Quick Quiz

- 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]+$`