

Regular Expressions and How to Wield Them

a presentation for
The Toledo Area Linux Users Group (TALUG)

by Jason M. Bechtel

July 21, 2007

What are Regular Expressions?

A regular expression (**regex** or **regexp** for short) is a pattern describing a certain amount of text. Think of regular expressions as **wildcards on steroids**. You are probably familiar with wildcard notations such as `*.txt` in a shell (like Bash). This is sometimes referred to as **globbing** and is another topic unto itself. Regexes are **much more powerful**.

Literals vs. Metacharacters

Regexes are just strings of characters. Some of the characters represent themselves, but some characters have special meaning in regexes. These are the metacharacters: `.` `^` `$` `*` `+` `?` `{` `[` `]` `\` `|` `(` `)`

To match one of these characters as a literal in a regex, you need to escape it using the backslash metacharacter (`\`), eg: `"1 \+ 1= 2"`

Character Sets: []

A character set (or character class) matches **any ONE** of a list of characters (eg: [a e] matches 'a' *or* 'e').

Sets have their own rules for special characters!

- '-' (dash) is used to specify a range, eg: [a - e]
- '^' (caret/hat) at the beginning means "complement", eg: [^ a - e] (= "anything but a-e")
- '\' (backslash) is an escape for metacharacters
- other metacharacters are just literals

Some Example Character Sets

- $[a e]$ = either a or e
- $[a - e]$ = a, b, c, d, or e
- $[a - zA - Z 0 - 9]$ = any alphanumeric character
- $[- _ \backslash \backslash \wedge \$ \cdot \backslash [\backslash]]$ = -, _, \, ^, \$, ., [, or]
- $[\wedge]$ = any character that is *not* a space
- $[\wedge \backslash s]$ = any character that is *not* whitespace

Predefined Character Sets

- $\backslash d$ = any decimal digit = $[0-9]$
- $\backslash D$ = any non-digit = $[^0-9]$
- $\backslash s$ = whitespace = $[\backslash t \backslash n \backslash r \backslash f \backslash v]$
- $\backslash S$ = any non-whitespace character
- $\backslash w$ = any "word" character = $[a-zA-Z0-9_]$
- $\backslash W$ = any "non-word" character

Matching Anything: '.' (dot)

What if you want to match **any character** (like * or ? in file globbing)? In regular expressions...

- Star (*) and question mark (?) have other meanings
- **Dot (.) is the match-anything character**
- **Exception**: Dot (.) usually doesn't match line break characters (\n and \r), except in special modes.

Matching Multiple Times

What if you want to match something more than once?

- Star (*) means *"match zero or more times"*
- Plus (+) means *"match one or more times"*
- Question mark (?) means *"match zero or one times"*

Match what? Whatever expression precedes the

metacharacter... eg: [0-9]* \s+ .?

Greedy vs. Lazy Matching

- The repetition metacharacters are "greedy"!
 - expand as far as possible and only give back as needed to match the rest of the expression
 - eg: `<. +>` matching against "this ``is a`` test"
 - matches "``is a``", not "``"!
- "Lazy" matching is turned on with a `?` after the repeat metacharacter
 - eg: `<. +?>` matches "``"

Greedy Matching, Step-by-Step

Example: pattern a [bc d] * b vs. string "abcbd"...

<u>Matched</u>	<u>Explanation</u>
1. a	▪ 'a' in the RE matches
2. abcbd	▪ [b c d] * matches to end of line
3. failure	▪ tries to match b, but can't, fails
4. abcb	▪ backs up one character
5. failure	▪ tries to match b again, fails
6. abc	▪ backs up again
7. abcb	▪ matches b, done

Practice Welding

How would you construct a regex to match all of the following strings?

- abc
- Abbc
- abbbC
- Abbbbbbbbbbbbbbbbc
- without matching "ac"!

Practice Wielding

What strings would the following regexes match?

- `gr[ae]y`
- `[A-Z]?[a-z]+`
- `[.0-9]+`
- `\$?[0-9]*\.[0-9][0-9]`
- `[\w.-]+@\w+\.\w+`
- `<[^<>]+>`

Anchors

What if you want to match something at the beginning or end of a line **only**?

- `^` anchors the pattern to the beginning of the line
- `$` anchors the pattern to the end of the line

Examples:

- `^#` (comment lines in config files)
- `^$` (empty lines)

Parentheses and Pipes are Powerful!

What if you want to match a whole pattern multiple times, allow any of just a handful of possibilities?

- `()` creates a "sub-pattern"
 - can access contents later (how depends on environment)
 - allows for creation of repeatable blocks, eg:
 - `a (b cd) *e` (any number of 'bcd's between 'a' and 'e')
- `all|one` allows for 'all' or 'one'
 - NOTE: *not* `al[lo]ne`

Parentheses and Pipes are Powerful!

Examples:

- `a (bcd) * e` (any number of 'bcd's between a & e)
- `reg (ular expressions ? | ex (p | es) ?)`
- `([0 1] [0 - 9] | 2 [0 - 3]) : [0 - 5] [0 - 9]`

Curly Braces (for complex repeats)

What if you want to match a pattern a precise number of times (more than twice)?

- $\{m, n\}$ only recognizes expression repeated from m to n times, *inclusive*
 - omitting m implies zero (0)
 - omitting n implies infinity (actual limit depends)
 - $\{0, \}$ = *
 - $\{1, \}$ = +
 - $\{0, 1\}$ = ?

Go Practice!

Now, let's go wield some regular expressions!