Regular Expressions with ‘egrep’

Objectives

- Introduce regular expressions
- Learn how to create basic and more advanced regular expressions
- Introduce the ‘egrep’ program
- Learn how to search the contents of text files for patterns

Regular Expressions

Introduction to Regular Expressions

- A regular expression is a search pattern
- Regular expressions were introduced by the American mathematician Stephen Kleene in the 1950s
- Initially used with Unix text processing utilities (`ed` and `grep`)
- Nowadays many programming languages (including Java) support regular expressions
- Common abbreviations: `regex` or `regexp`
- Similar to the *globs* introduced in the previous lecture, but much more advanced

Overview of Regular Expressions

- The intention behind regular expressions is to provide a way to *match* certain parts of a string or a text
- Regular expressions are represented by sequences of characters, including
  - literal text: `string`
  - bracket expressions: `[aeiou]`, `[a-m]`, etc.
  - character classes: `[:lower:]`, `[:digits:]`, etc.
  - quantifiers: `*`, `?`, `+`, `{n}`, `{n,m}`, etc.
  - further special 'meta' characters: `^`, `$`, `.`, `\`, `|`, etc.

Matching Literal Text

- The simplest regex represents one single character: `s`
- Another simple regex matches several consecutive characters: `string`
- By default, regex matching is case sensitive
- By default, all occurrences of the corresponding regex are matched
- Example:
  - Regular expression: `t`
  - Input string: `Text technology`
  - Match: ⇑ ⇑
- Note that several characters have special meanings (meta character), which will be introduced on the following slides

Bracket Expressions

- A bracket expression is a list of characters enclosed by `[` and `]`
- The expression matches any single character in that list
  - `[ab]`: matches either `a` or `b`
  - `[12]`: matches either `1` or `2`
- If the first character of the list is the caret (‘`^`’), then it matches any character not in the list
  - `[^ab]`: matches any character except `a` and `b`
  - `[^12]`: matches any character except `1` and `2`

Ranges in Bracket Expressions

- Within bracket expressions, the hyphen can be used to represent character ranges
  - `[0-9]`: matches every number from `0` to `9`
  - `[a-z]`: supposed to match all (small case) letters from `a` to `z`
  - `[A-Z]`: supposed to match all (upper case) letters from `A` to `Z`
  - Remember the dependence on your locale for ranges that represent letters: better use character classes!

Character Classes

- Character classes represent ranges or types of characters
- Useful predefined character classes
  - `[:alpha:]`: alphabetic characters
  - `[:digit:]`: numeric characters
  - `[:alnum:]`: alphanumeric characters
  - `[:lower:]`: lower-case alphabetic characters
  - `[:upper:]`: upper-case characters
  - `[:punct:]`: punctuation characters
  - `[:blank:]`: horizontal whitespace characters (empty spaces and tabs)
  - `[:space:]`: horizontal and vertical space characters (including empty spaces, tabs, and newlines)

Character Classes (2)

- When used in a bracket expression, the brackets around the character classes must be included
  - match any upper case character: `[[:upper:]]`
  - match any character from character classes `[:punct:]` and `[:blank:]`: `[[:punct:][:blank:]]`

Quantifiers

- Quantifiers indicate repetition of the preceding regex item
- `+`: the preceding regex item will be matched one or more times
  - `t+` matches `t`, `tt`, `ttt`, etc.
  - `ab+c` matches `abc`, `abbc`, `abbbc`, etc.
  - `a+b+c+` matches `abc`, `aabc`, `abbc`, `abcc`, `aabbcc`, etc.

Quantifiers (2)

- `*`: the preceding item will be matched zero or more times
  - `t*` matches `t`, `tt`, `ttt`, etc., or nothing
  - `ab*c` matches `ac`, `abc`, `abbc`, `abbbc`, etc.
- `?`: the preceding item is optional and will be matched at most once
  - `t?` matches `t` or nothing
  - `colou?r` matches `color` or `colour`
- Be careful: characters `?` and `*` have a (slightly) different function in globs

More Quantifiers

- `{n}`: the preceding item is matched exactly `n` times
  - `t{6}` matches `tttttt`
  - `to{2}l` matches `tool`
- `{n,m}`: the preceding item is matched at least `n` times, but not more than `m` times
  - `t{4,5}` matches `tttt` and `ttttt`
  - `se{1,2}d` matches `sed` and `seed`

More Quantifiers (2)

- `{n,}`: the preceding item is matched `n` or more times
  - `t{2,}` matches `tt`, `ttt`, `tttt`, etc.
- `{,m}`: the preceding item is matched at most `m` times (this is a GNU extension)
  - `t{,3}` matches `t`, `tt`, `ttt`, or nothing

Start and End of Lines

- The two meta characters `^` (caret symbol) and `$` (dollar sign) can be used to refer to the beginning and the end of a line
- `^`: matches the empty string at the beginning of a line
  - `^Text` matches lines starting with `Text`
  - Be careful: within bracket expressions the `^` symbol has a different meaning
- `$`: matches the empty string at the end of a line
  - `logy$` matches lines ending with `logy`

Start and End of Lines (2)

- Both `^` and `$` are termed *anchors*, since they force the match to be *anchored* to beginning or end of a line, respectively
- `^Text.*logy$` matches lines starting with `Text` and ending with `logy`, such as

```
Textlogy
Text WHAT3V3R 3L53 logy
Text technology
```

Matching Any Single Character

- `.`: the period functions like a placeholder that matches any single character
  - `a.c` matches `aac`, `abc`, `aXc`, `a3c`, `a-c`, etc.

Combining Regular Expressions

- `|`: combining two regular expressions with a pipe symbol means that the whole regex matches any string matching either regex
  - `a|b` matches `a` or `b`
  - `text|sound` matches `text` or `sound`

Escaping Meta Characters

- \: to interpret special characters as literals, escape them with a backslash
  - * matches *
  - \. matches .
  - \\ matches \
  - 3\+6=9 matches 3+6=9

Matching Whole Words

- `\b`: match the empty string at the edge (i.e., at the beginning or at the end) of a word
  - `\bap` matches words starting with `ap` such as `apple`, `apricot`, or `application`
  - `ing\b` matches words ending in `ing` such as `string`, `thing`, or `accepting`
  - `\bship\b` matches `ship` as an entire word, i.e., excluding `ships` or `shipment`

Subexpressions

- Subexpressions are parts of larger regular expressions
- A whole expression may be enclosed in parentheses `(` and `)` to form a subexpression
  - `(T)ext`
  - `a(bc)d`
  - `(0[[:digit:]]{3,4})/([[:digit:]]{4,})`
- An unmatched `)` matches just itself
- Subexpressions are useful:
  - subexpressions can control precedence
  - back-reference can be used to refer to subexpressions

Precedence

- Repetition (e.g., '`*`', '`+`', '`?`', or '`{n}`') takes precedence over concatenation, which in turn takes precedence over alternation ('`|`')
  - repetition >> concatenation >> alternation
- Subexpressions can override these precedence rules
  - `cha+` matches `cha`, `chaa`, `chaaa`, etc.
  - `(cha)+` matches `cha`, `chacha`, `chachacha`, etc.
  - `ana{2}s` matches `anaas`
  - `a(na){2}s` matches `ananas`
  - `any|nobody` matches `any` and `nobody`
  - `(any|no)body` matches `anybody` and `nobody`

Back-Reference

- We can use back-references to refer to subexpressions
  - the subexpression has to appear first, before we can refer to it by a back-reference
- A back-reference is written as `\n`, where `n` is a single digit referring to the `n`-th subexpression
  - `(t)\1` matches `tt`
  - `ba(na)\1` matches `banana`
  - '`(red|big) is \1`' matches '`red is red`' and '`big is big`'
  - '`(h?)(a)(n)\3\2\1`' matches '`hannah`' and '`anna`'

Greedy and Lazy Matches

- Regular expressions distinguish greedy versus lazy matches
  - relevant in combination with quantifiers
- `greedy`: the regex matches as many characters as possible
  - this is usually the default mode (e.g., for the 'egrep' program)
- `lazy` (`non-greedy`): the regex matches as few characters as necessary
  - if you want a lazy search, insert a '`?`' after the relevant quantifier: `*?`, `+?`, `??`
  - unfortunately, egrep does not support lazy expressions by default

Greedy and Lazy Matches – Examples

- Input string: `aabb`
  - greedy pattern `a*` matches `aa`
  - lazy pattern `a*?` matches `a`
- Input string: `"This" is what we "search" for.`
  - greedy pattern `".*"` matches `"This" is what we "search"`
  - lazy pattern `".*?"` matches `"This"` and `"search"`

Example Regular Expressions

- Referring to all text files
  - The glob from previous lecture: `*.txt`
  - The equivalent regular expression: `.*\.txt$`
- Match all lines that have exactly five characters
  - `^.{5}$`
- Match IP addresses such as `192.168.1.22` or `10.0.1.255`
  - `([:digit:]{1,3}\.){3}[:digit:]{1,3}`

The ‘egrep’ Program

Overview of 'egrep'

- **grep**: **g**lobally search a **r**egular **e**xpression and **p**rint
- **egrep**: '**e**xtended' version of **grep**
- `egrep` searches input files for lines matching a specified pattern
- When a matching line is found, it is copied to egreps standard output channel
- You should use GNU's egrep for all exercises
  - Note explanations from introductory lecture
  - Remember: on OSX use `gegrep`

Searching in (Text) Files with egrep

- `egrep <pattern>`: search standard input for lines containing the specified `pattern`
  - `pattern` can represent any regular expression
  - `$ <some commands> | egrep 'wom[ae]n'`
  - `$ egrep wom[ae]n < RomeoAndJuliet.txt`
- `egrep <pattern> [file(s)]`: search provided files for lines containing the specified `pattern`
  - `$ egrep wom[ae]n RomeoAndJuliet.txt`

Important Command Line Options for egrep

- `egrep` has a lot of command options
- Usage: `egrep [options] <pattern> [file(s)]`
- Most important options are `-i` and `-v`
- `-i`: search ignoring case
  - search for '`day`' (ignoring case) in file `RomeoAndJuliet.txt`:
    `$ egrep -i day RomeoAndJuliet.txt`
- `-v`: search with the inverted meaning of the match
  - search for all lines **without** '`Romeo`':
    `$ egrep -v Romeo RomeoAndJuliet.txt`

Further Command Line Options for egrep

- `-o`: print only the parts that match the pattern (not the whole line)
  - search for digits and only print those matching digits:
    `$ egrep -o '[[:digit:]]+' RomeoAndJuliet.txt`
- `-w`: search for whole words only
  - search for '`rose`' as a whole word:
    `$ egrep -w rose RomeoAndJuliet.txt`
    note that this search is equivalent to:
    `$ egrep '\brose\b' RomeoAndJuliet.txt`

Further Command Line Options for egrep (2)

- `-x`: pattern must match the whole line
  - search for lines solely containing
    `ACT I`
    `ACT III`
    `ACT VI`
    etc.:
    `$ egrep -x 'ACT [IV]+' RomeoAndJuliet.txt`
    note that this search is equivalent to:
    `$ egrep '^ACT [IV]+$' RomeoAndJuliet.txt`
- `-c`: count the number of matching lines
  - count lines in which '`ROMEO AND JULIET`' occurs:
    `$ egrep -c 'ROMEO AND JULIET' RomeoAndJuliet.txt`

Combining Command Line Options

- It is possible to combine multiple command line options
  - of course, not all options make sense in combination, but it is generally possible
- Search for all lines **without** '`Romeo`' (as one word, ignoring case) and count the resulting lines:
  - `$ egrep -ivwc Romeo RomeoAndJuliet.txt`