

TALUG Meeting Notes

July 21, 2007

Meeting Location

Upon arriving at Stautzenberger College, we found that it had closed. Stautzenberger College had moved to a new facility in Maumee, and closed their Southwick location. This was previously unknown to all members in attendance and we were unsure of what to do. Fortunately Jason came through in the clutch and got us a room at the Maumee Library.

The meeting started approximately 35 minutes late due to the building mix-up, but otherwise went as scheduled.

Regular Expressions: Presented by Jason Bechtel

A regular expression cheat sheet was provided, and can be found [here](#).

These notes are a summary of the presentation, examples, and questions. For Jason's full presentation, please see the above link in the title.

Background

On the grand scale of things, regular expressions are relatively low order expressions, but in terms of computing, regular expressions are the highest order expressions that are still computationally feasible. Don't get the wrong idea, regular expressions are still *very* powerful.

Regular expressions are patterns for describing and or validating text. Regular expressions can also be used to produce new text. For our purposes, they are generally used for validating.

Regular expressions are different the globbing. An example of globbing is *.txt where this expression matches any string ending in .txt This is not a regular expression.

Regular expressions use two kinds of characters, Literal characters and Meta-characters. Example: asterisk *, it has a meaning beyond (meta) it's value. The other kind of character is the literal character, which means exactly what it is.

Meta-characters

Meta-characters consist of: . ^ \$ + ? { } [] | \ There are 11 meta-characters in total. If a meta-character is to be used as a literal character, it needs to be backslash escaped.

Square Braces []

Square braces are character sets. They are used to match any *one* of the characters enclosed.

Example: `gr[ae]y` Will match the literal `gr`, followed by *either* `a` or `e` and then the literal `y`.

Meta-characters can be used within the square braces, although they have different meanings than when used outside of the square braces. There are only three meta-characters that can be used inside square braces, these are:

- Dash indicates a range
- ^ Caret means compliment, everything other than.
- \ Backslash is an escape character for meta-characters.

Examples: Square brace use.

[a-zA-Z0-9] will match any alphanumeric character. Will only match one character.

[^] will match everything that is not a space. Note that tab is not a space.

Square brackets can use predefined character sets:

\w word characters

\s whitespace

\d digits

The predefined character sets have complements, which just use the capital letter. For example, all non-word characters would be \W.

asterisk * , plus +, questionmark ?

Repetition meta-characters. These meta-characters should follow the object to be repeated.

* Match 0 or more

+ Match 1 or more

? Match 0 or 1

Examples:

gr[ae]*y will match gry graay graey graaeaeaeay

[0-9]* will match a string of digits

Greedy Matching

Greedy matching tries to match as much as possible. Greedy matching is the default matching scheme in regular expressions. This can be best explained with an example:

Example: Step by step explanation of greedy matching.

String: abc**bd**

Regexp: a[**bcd**]*b

1. Matches the literal a
2. Matches **abcd**
3. Fails to match literal b
4. Backs up a character, tries to match b
5. Fails to match literal b
6. Backs up a character, tries to match b
7. Succeeds in matching b

The end result is that **abcb** has been matched.

Lazy Matching

Lazy matching is the opposite of greedy matching. It tries to match as little as possible. Because regular expression defaults to greedy matching, in order to use lazy matching the meta-characters need to be modified. The modifier for lazy matching is the question mark ?

Once again, this is best explained with an example:

Example: HTML matching

String: This `` is a test ``

Regex: `<. +>`

This regular expression would match the entire string.

If we wanted to only match the first ``, we would have to use lazy matching such as `<. +?>` where the ? is a modifier for the meta-character +

Practice: Write a regular expression to match all of the following:

```
abc
Abbc
abbbC
Abbbbbbc
```

To match these characters, we can use the following regular expression `[a-zA-Z]*`, but if we want to enforce the order a to b to c, use `[aA][b]+[cC]` or `[aA]b+[cC]`

More Practice: Explain the following regular expressions:

`[A-Z]?[a-z]+` Will match capitalized or non capitalized words.

`[.0-9]` The dot (.) is a literal character inside square braces, so it will match a decimalized number. For example, it would match 1.1 or 1.1.21.

`\$?[0-9]*\.[0-9][0-9]` Will match monetary values which have 2 decimal places, can have a dollar sign or not, and can have 0 or more digits before the decimal. Note: the backslashes are used to 'escape' a meta-character, and cause it to be interpreted as a literal character.

`[\w.=]+@\w+\.\w+` Would match email addresses. Note: word characters `\w` include underscores.

`<[^<>]+>` Would match one HTML tag.

Anchors ^\$

Anchors will 'anchor' a regular expression to a specific location. Anchors include the hat ^ which anchors to the beginning of the line, and the dollar sign \$ which anchors to the end of the line.

Example: We want to only match lines starting with comment #

Regex: `^#`

This regular expression will match lines beginning with the pound sign #.

Example: Find blank lines:

Regex: `^$`

This regular expression will match a line containing nothing in between the beginning and end of that line.

Example: Use grep to find what is in a file, excluding the comments:

```
grep -v "^#" file
```

Parentheses and Pipes

Parentheses are blocks. They “block” off a section of regular expressions, which can then be repeated. In addition to blocking off sections, they also create a sub-match which can be recalled as a variable. The entire regular expression can be recalled with `$0` (Usually, but it depends on the environment), and the first block can be recalled as `$1`.

Example: Repeated matching

Regexp: `a(bcd)*e`

This regular expression will match the literal `a`, the literal string `bcd` *repeated* and then the literal `e`

Example: Sub matching

Regexp: `a([bcd])e`

This regular expression will match the literal `a`, and any *one* of `b`, `c`, or `d`, and then the literal `e`. In addition, the character matched by the square brackets (`b`, `c`, or `d`) can be recalled with `$1`.

Pipes (`|`) are used as a logical ‘or’. For example `all|one` will match `all` *or* `one`. It does not behave as `al[lo]ne`

Example: `reg(ular expressions?|ex(p|es)?)` Will match `regular expressions` or `regex` or `regexes` or `regexp`

Example: `([01][0-9]|2[0-3]):[0-5][0-9]` Will match 12 and 24 hour time. It can be recalled and used later with `$0`.

Curly Braces { }

Curly Braces `{ }` allow us to repeat a regular expression more than 0 or 1 times, but less than infinite. The general usage is `(some regular expression){m,n}` where the regular expression will be repeated `m` through `n` times inclusive.

Example: Match all contents from column 270 to the end of the line.

Regexp: `^.{269}(.*)$`

Programs that do visual regexping

There are programs which will generate regular expressions so that you don’t have to, several common programs are:

- [kodos](#)
- [visual-regexp](#)

Using Regular Expressions

The most common programs that use regular expressions are `grep`, `sed`, and `awk`. `Find` does *not* use regular expressions, it uses file globbing instead.

Notes on `grep`: `grep` can be used with the `-lr` option to list file names that contain matches recursively. To find configuration files that match some pattern:

`find /etc -iname '*.*conf' -exec grep -l 'pattern' {} \:` Also, instead of `-exec`, `| xargs` could have been used. `xargs` builds a list from the output, and formats it to pass as command line arguments.

Notes on sed: sed is a stream editor. Jason most commonly uses `sed -e 's/ /g'` where `-e` is for expressions. The `s` stands for substitute. The forward slashes are simply separators, other separators can be used if your expression contains forward slashes. The field in between the first two separators is a regular expression. The field between the second two separators is what is to be substituted upon a match of the regular expression. The `g` option specifies global *within* a line of text, rather than just the first match on the line.

References

[Regular Expression Howto](#)

[Regular-Expression.info](#)

Software Freedom Day Discussion

A planning session was held for [Software Freedom Day](#), which is scheduled to occur on Saturday September 15, 2007. In the planning session we brainstormed locations, advertising, and presentations. This session was a brainstorming session. The outcomes of this brainstorming session are outlined below.

Location

Several locations were proposed, including Westfield Shopping Center, the Main Library, and the University of Toledo Student Union. Each location had advantages and disadvantages. Westfield Shopping Center had the advantage of being in a very high traffic area, as well as having an audience who are already looking to acquire new things (shopping). The downsides of Westfield Shopping Center were cost and demographic. In a previous year's attempt, Westfield Shopping Center had wanted \$250 for the day. Also, as Jason suggested, the people who would benefit from this event most are those who don't have new, fast computers, but rather those who have older computer are do not know how to get new software such as Microsoft Word without paying \$200 for it.

The Main Library had the advantage of being free, and having a nice auditorium with a projector microphone etc ... The downside of the Main Library was that we would have to convince people to come and listen, rather than being in the middle of a high traffic area.

The University of Toledo Student Union is another possibility. The University of Toledo provides a relatively high traffic area (Student Union) that would be free of cost, but comes with a catch 22. On Friday, there is high traffic, but due to parking restrictions (parking passes) only UT students will be able to attend. On Saturday, the campus will have plenty of parking, but will be relatively empty.

A solution was proposed were TALUG would have a booth in the Student Union for few hours Friday and do presentation at the Main Library on Saturday.

Advertising

In order to have a successful SFD, the event will need to be advertised. This could include handing out fliers and posting fliers. If we have the event at the Main Library, there would be advertising opportunities are the branch libraries. Bookstores were also suggested, as well as at all the regional colleges (UT, Owens, Stautzenberger...). Details of advertising will need to be decided at a future meeting.

Events

University of Toledo

The University of Toledo part of SFD (Friday) will consist of a booth somewhere on main campus in a high traffic area. Computers will be set up showing FOSS programs and distributions. We will be handing out

CD's of FOSS programs and distributions. It was suggested to set up some sort of game, such as a 'choose your distro' version of cornhole, in order to increase receptiveness to the CD's, as well as to generate interest. It was suggested that people are more likely to keep and use a CD that they won as a 'prize' than ones being given away. This event will also serve as advertising for Saturday's main library event.

Main Library

The Main Library portion of the event, on Saturday (Official SFD Day) will be set up in an auditorium at the Main Library. We will need someone in the lobby directing people to the proper room. The presentations should be short (10-15 minutes?) and should be given on a rotating basis. Presentation topics could include productivity applications (open-office), image manipulation (gimp, gimpshop, blender, scribus), programs for kids, programs / distros for lower end hardware, visual bling (xgl etc).

General Notes

- We should be handing out pressed CD's, as they look more official.
- Demonstrated programs and operating systems should be visually pleasing. Fluxbox without GTK themes might be useful and fast, but it looks a bit archaic.
- No command line. As cool and powerful as this tool is, it tends to scare away new users.