Stream EDitor
Instructor Handout

Author: Reuben Francis Cornel

Contents

1 Regular Expressions

- **Take the session a bit slowly**
- **Be a bit more elaborate when explaining**
- A regular expression is a pattern which describes a certain amount of text.
- A regular expression is processed using a "Regular Expression Engine"
- It is important to realise that the regular expression engine
- **Explain how the basic matching process occurs Show example0.pl to illustrate this.** will always return the left most match, even if there is a better match to the right of the string.

1.1 Literal Characters

- The most basic regular expression is a Single literal character.
- A Literal character is nothing but a simple printable or a non printable character

1.2 Special Characters

- Some characters are reserved for special purposes. These characters could be used to select one or more that one character, negate the selection so on and so forth.

1.3 Character Classes

- You can use a character class to match a single character from a list of several characters.
- They are represented using [ ](square brakets)
- **Show example1 and ask what the output will be. This is a chocolate question**

  Example: the regular expression `gr[ae]y` will match `grey` or `gray`.

- You can use the hypen "-" to specify a range of characters.

  Example: The expression [0-9] will match a single digit. But can you tell me what sort of character will the expression [0-9a-fA-F] match.

1.3.1 Negated Character Classes

- Typing a caret after the opening square bracket will negate the character class. The result of this is that the character class will not match the characters which are present in the character class.

1.3.2 Special Characters in Character Classes

- The only special characters, which have meaning, inside the
- **Show example2 and ask what the output will be. This is a chocolate question. The output should be only the line having the * or t preceded by "ca"** character class are
  - Closing square bracket
  - Backslash
  - Caret
  - Hypen

1.4 The '.' Operator

- The dot is used to match any characters which the exception of new line characters.
- **Can you think as to why new line character matching was not present? Simple reason, in earlier days the tools were line based so the regular expression would be applied to every line seperately, and genrally the new line was stripped of.**
- You have to be careful when using the dot, because sometimes it might match something which you don't intend it to match.
- **Show example 3 as a example of matching a date pattern.**

1.5 Anchors

- Unlike literal and special characters, the regular expression engine will try and match positions, such as the begining and end of strings.
- The ^(caret) matches the start of the line
- The $ (dollar) matches the end of line.
- One of the useful cases, where anchors are used is to clean up blank lines.

  Example: The command shown below can be used clean up blank line in vi

```
:%/^$/d
```

1.6 Optionals

- The question mark(?) symbol tell the regular expression engine that the preceding expression is optional. That is the expression may or may not occur.

  Example: The expression `colou?r` will match colour or color
- **Show example5. Chocolate Question**

1.7 Repetitions with + and *

- We have two other repetition operators as far as regular expression are concerned. First is the * (asterix) that matches **zero or more occurences** of the expression. Then we have the + (plus) sign which matches **one or more occurences** of the regular expression.
- **Show example6**
- WARNING: remember that both + and * are "greedy". That means that they will continue looking for the largest possible match even if they have found a match.
- **Show example6a. Show example6b as a solution.**
- Could you suggest a better Regular Expression to get the tag values?

1.7.1 Limiting repetition

- You can use the flower bracket({}) to specify the number of times a regular expression may be repeated
- **Show example21**
- The syntax of using {} is as follows

$$<\text{Regular Expression}>\{min, max\}$$

- *min* represents the minimum number of times a regular expression may be repeated.
- *max* represents the maximum number of times a regular expression may be repeated.
- Mentioning the minimum number is mandatory, where as you need not mention the maximum number. If you don't mention the maximum number
- **Ask how can you emulate * and + using {}**

1.8 Alternating

- Using alternation you can match a single regular expression out of several regular expressions.
- This is represented using the `|` (pipe symbol).

  Example: You can use (cat|dog) to match either cat or dog in a input string.

- **Show example4 and ask for output**
- WARNING: Remember the regular expression engine is eager it will stop the minute it finds the first match. So you have to make sure that your regular expression describes the right match.
- **Show example4a.pl**

  Example: If my regular expression is `Get|GetValue|Set|SetValue` Can anyone explain how this regular expression matches the string "SetValue".

- **Ask if anyone can explain.**

1.9 Back references

- Round brackets are used to create groupings of regular expression.
- **Show example7.pl**
- Apart from grouping regular expressions, these bracket also create "Back References".
- A *Back Reference* is nothing but a section of text that has been stored since it matches a regular expression.

- **Show example 6**
- Generally the text is stored in variables whose names are numbered for example, in sed the backreference text is stored in variables of name \1, \2 etc, in perl back references are stored in variable with names such as $1, $2 etc.

1.10 Exercises

- Write a regular expression to match ip addresses.

```
(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.
(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.
(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.
(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)
```

- **Pass the sheet out, containing mail address**

2 Sed

2.1 Uses Of Sed

- Text modifications on the fly.

2.2 Command Line Arguments

- **-n** This command line argument tells `sed` not to copy the input to the output.
- **Show example8 then show example6 to see difference**
- **-e** This command tells `sed` to take the next argument as another editing command.
- **Show this by writing on the board**
- **-f** This command tells `sed` to take argument as the file and interpret it line after line.

2.3 Basic Operation

- The basic structure of the editing command in `sed` is given as follows.

  [address1, address2][function][arguments]

- One or both of the addresses may be omitted.
- The function must be present
- The argument is either mandatory or optional based on the type of function the argument is used for.
- The order of applying the is sequential. But this can be changed based on the branching commands.

2.3.1 The Pattern Space

- The line which is read in by `sed` by default is put into a buffer called the pattern space. This is the place where all sorts of butchery is performed on the text.

2.4 Addresses

- Lines to which the editing commands are to be applied can be selected by addresses.
- These addresses may be *context based addresses* or *Line based addresses*

2.4.1 Line Addressing

- As the name suggests line based addressing means that you directly refer to the line number for which you want to apply your editing command.
- **Show example9**
- The line number is reset to zero when a new file is opened.
- The special case for line number addressing is $ which represents the last line.
- **Show example9a**

2.4.2 Context Based Addressing

- A context based address looks for the lines which match the regular expression which is being searched.
- The regular expressions used by `sed` is as discussed above in the regular expressions section of this handout.
- If there is a need to use special characters in the regular expression, those characters would have to be preceded by a \
- **Show example10**

2.4.3 Number of addresses

- The `sed` commands can take 0, 1 or 2 addresses. Below we will see the affects of giving the specified number of addresses.
- **No Address**: When you give no address the command is applied to every line.
- **Show example11**
- **1 Address**: The command is applied to all lines that match that address.
- **Show example10**
- **2 Addresses**: The command is applied to the first line which matches the address and all subsequent lines till the line which matches the second address.
- **Show example9**

2.5 Functions

2.5.1 Whole Line Oriented Functions

- **d** – This function deletes the lines from the input, which match the address given to it. But there is no change done to the original file.
- When the "d" function is executed for a line, a new line is read in from the input the list of command is restarted for the newline read in.
- **n** – This function reads the next line from the input, replacing the current line in the pattern space.
- The flow of editing commands continues on the new line read in.
- **a** – This function outputs the given lines of text after the line which matches the pattern.
- This function can append any number of lines to the output.
- These lines can be seperated by newlines but must have a \at the end.

  Example:
  a\
  Test

- **Show Example12, show example12a for demo of what the command will do with out "d"**
- **i** – This function adds lines to the output just before the line which matches the address. All other comments are as for the previous command.
- **Show Example14**
- **c** – This function changes the lines which have been selected by the address. It deletes the original lines and replaces them with the text which is supplied.
- **Show Example13**

2.5.2 Substitute Function

- The function changes parts of lines selected by a context search within the line. The general syntax of this command is given below.

  ```
  s/<pattern>/<replacement>/<flags>
  ```

- The delimiter used to seperated the pattern, replacement and flags fields in the above example is a \. But this can be replaced by anything as long as you use the same character as delimiter.
- The pattern is a regular expression.
- The replacement is not a regular expression. The only characters which have special meanings are
  - & which signifies the pattern and
  - \d where d represents an integer for the corresponding pattern enclosed in \(\)

- The flags may contain any of the following flags
  - **g** – substitute all occurences of the pattern.
  - **p** – print the line if the substitution was successful.
  - **Show Example16**
  - **w <filename>** – writes the substituted line to a file.
  - **Show Example15**

2.5.3 Input-Output Functions

- **p** – This command prints the lines to the output as soon as this command is encountered.
- **w <filename>** – This command writes the addressed lines a file. There must be exactly **one space** seperating w and the filename. A max of 10 different filename can be mentioned in different w commands.
- **r <filename>** – This command command reads the given file and prints the lines to the output.
- **Show Example17**
- **Ask what will happen if the file is not found.**

2.5.4 Multiple Input Line Functions

- **N** – The next line is appended to the pattern space and the two lines are seperated by \n. This can be used to match patterns across lines.
- **Show Example18**
- **P** – This command prints the part of the text from the pattern space till the first \n
- **Show Example19**

2.5.5 Hold and Get Functions

- `sed` has a buffer space called the "hold" space which can be used to save text for further modification. The functions that help us use this space is called the hold and get functions.
- **h** – copies contents of the pattern space to the hold space destroying its previous contents.
- **H** – copies and appends the copied contents to the contents of the hold space. The lines are seperated using \n.
- **g** – get contents of the hold area into pattern space, destroying what was previously present.
- **G** – get contents of the hold area into pattern space, appending to what was previously present in the pattern space.
- **x** – this command exchanges the contents of the pattern and hold spaces.
- **Show Example20**

2.5.6 Control Flow Functions

- These functions do not do any editing to the selected lines instead they control the flow of how the functions are applied to the lines.
- **!** – This command is applied to all lines which are *not* selected by the address lines.
- **{}** – This command helps you group all editing commands into blocks. These blocks can be nested.
- **:<label>** – This command places a label to which branch commands can jump. The label at max can have only 8 characters. If a label already exists an error will be generated.
- **:b <label>** – This command causes the execution to jump to the label specified unconditionally
- **Show Example19**
- **t <label>** – This command branches to a label only is a successful substitution has been made previously on the same line.

2.5.7 Miscellaneous Functions

- =: This function writes the line number to the standard output.
- **q**: This function causes execution of sed to be terminated, this function will print any lines which have to be printed before quitting.

2.6 References

- www.regular-expressions.info
- http://sed.sourceforge.net/grabbag/