Unix Regular Expressions

Stephen Corbesero
corbesero@cs.moravian.edu

Computer Science
Moravian College
I. Introduction
II. Filenames
III. Regular Expression Syntax and Examples
IV. Using Regular Expressions in C
V. SED
VI. AWK
VII. References
Regular Expressions (REGEXPs) have been an integral part of Unix since the beginning. They are called “regular” because they describe a language with very “regular” properties.
Filename Patterns

- ? for any single character
- * for zero or more characters
- [...] and [^...] classes

Examples

* all files
*.* files with a .
*.c files that end with .c
*.? .c, .h, .o, ... files
.[A-Z]* .Xdefaults, ...
*.[ch] files ending in .c or .h
**Unix Commands**

- **grep** is the “find” string command in Unix. It’s used to find a string which can be specified by a REGEXP.
- It’s name comes from the `ed` command `g/RE/p`.
- There are actually several greps: `grep`, `fgrep`, and `egrep`.
- The editors: `ed`, `ex`, `vi`, `sed`, ...
- Text processing languages: `awk`, `perl`, ...
- Sadly, different tools use slightly different regexp syntaxes.
Full Regular Expressions

- Character
  - ordinary characters
  - the backslash
  - special characters: . * []

- Character Sequences
  - concatenation
  - * to allow zero or more occurrences of the previous RE
  - \{n\}, \{n, \}, \{n,m\}
  - \(...\) for grouping
  - \n for memory
Small RE Examples

cat  
c.t  
c\.t  
c[a-eiou]t  
c[^a-z]t  
ca*t  
ca\{5\}t  
ca\{5,10\}t  
ca\{5,\}t  
a.*z
Word and Line REs

- **Words**
  - `< and >`

- **Lines**
  - `^` matches the bol
  - `$` matches the eol

- **Extensions recognized by some commands**
  - No `\`’s for braces and parentheses
  - `|` for or
  - `?` for `{0,1}`
  - `+` for `{1,}`
Word Examples

- To match *the*, but not *then* or *therefore*

  \(<\text{the}\>)

- Words that start with *the*

  \(<\text{the[a-z]*}\)>
Line Examples

- Match *the* at the beginning of a line
  ^the

- or end
  the$

- the line must only contain *the*
  ^the$
REs for Program Source

- Match decimal integers, with an optional leading sign
  \[\{0, 1\} [1-9] [0-9]^*\]

- Match real numbers, using extended syntax
  \[\{+-\}? [0-9]+ \.[0-9]+ ([Ee] [+-] [0-9]+) ?\]

- Match a legal variable name in C
  \[a-zA-Z_][a-zA-Z_0-9]^*\]
RE Memory

- Match lines that have a word at the beginning and ending, but not necessarily the same word in both places
 
  `^[a-z][a-z]*.*[a-z][a-z]*$`

- Match only lines that have the same word at the beginning and the end using.

  `\(([a-z][a-z]*)\).*\1$`
The `regcomp()` function “compiles” a regular expression.

The `regexec()` function lets you execute the regular expressions by applying it to strings.
#include <sys/types.h>
#include <regex.h>

int regcomp(regex_t *preg,
    const char * patt, int cflags);
int regexec(const regex_t *preg,
    const char *str, size_t nmatch,
    regmatch_t pmatch[], int eflags);
size_t regerror(int errcode,
    const regex_t *preg,
    char *buf, size_t errbuf_size);
void regfree(regex_t *preg);
sed, The Stream Editor

- “batch” editor for processing a stream of characters

```
sed [options] [ file ... ]
```

- The `-f` option specifies a file of sed commands
- The `-e` option can specify a single command.
- Both `-e` and `-f` can be repeated.
- `-e` not needed if just one command

- simple command structure

```
[addr [, addr]] cmd [args]
```
- a single line number, like 5
- lines that match a `/re/`
- the last line, `$`
- `address, address`
SED Commands

- blocks: {}, !
- text insertion: a, i
- script branching: b, q, t, :
- line changing: c, d, D
- holding: g, G, H, x
- skipping: n, N
- printing: p, P
- files: r, w
- substitutions: s, y
sed Examples

```bash
sed -n 1,10p file
sed s/cat/dog/
sed s/cat/&amaran/
sed s/\<\([a-z][a-z]*\)\> */\1/\1/g
sed s-/usr/local/-/usr/-g
```
A HTML processor

```
sed -f htmlify.sed ...

where htmlify.sed is
    1i\n        <HTML><HEAD>\n            <TITLE>title</TITLE>\n        </TITLE></HEAD>\n        <BODY>\n        <PRE>\n            $a\n        </PRE>\n    </BODY>\n</HTML>
```
The AWK Language

- Written by Aho, Weinberger, and Kernighan
- Useful for pattern scanning, matching, and processing
- AWK is an example of a “tiny” language
- Very good at string and field processing
  - Input is automatically chopped into fields
  - $1, $2, ...
- Command structure is very regular

\textit{pattern \{ action \}}
BEGIN
empty
/RE/
any expression
pattern, pattern
END
if ( expr ) stat [ else stat ]
while ( expr ) stat
do stat while ( expr )
for ( expr ; expr ; expr ) stat
for ( var in array ) stat
break
continue
{ [ stat ] ... }
expr    # commonly var = expr
print [ expr-list ] [ >expr ]
printf format [ ,expr-list ] [ >expr ]
next
exit [expr]
# AWK Special Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Usage</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILENAME</td>
<td>input file name</td>
<td></td>
</tr>
<tr>
<td>FS</td>
<td>input field separator /re/</td>
<td>blank and tab</td>
</tr>
<tr>
<td>NF</td>
<td>number of fields</td>
<td></td>
</tr>
<tr>
<td>NR</td>
<td>number of the record</td>
<td></td>
</tr>
<tr>
<td>OFMT</td>
<td>output format for numbers</td>
<td>%.6g</td>
</tr>
<tr>
<td>OFS</td>
<td>output field separator</td>
<td>blank</td>
</tr>
<tr>
<td>ORS</td>
<td>output record separator</td>
<td>newline</td>
</tr>
<tr>
<td>RS</td>
<td>input record separator</td>
<td>new-line</td>
</tr>
</tbody>
</table>
**AWK Data Types**

- floats and strings, *interchangeably*
- one dimensional arrays, but more dimensions can be simulated
  - `building[2]`
  - `map[row, col]`
- associative arrays
  - `state["PA"] = "Pennsylvania";`
  - `pop["PA"] = 12000000;`
AWK Functions

**Numeric**
- \( \cos(x), \sin(x) \)
- \( \exp(x), \log(x) \)
- \( \sqrt{x} \)
- \( \text{int}(x) \)

**String**
- \( \text{index}(s, t), \text{match}(s, \text{re}) \)
- \( \text{int}(s) \)
- \( \text{length}(s) \)
- \( \text{sprintf}(\text{fmt}, \text{expr}, \text{expr}, \ldots) \)
- \( \text{substr}(s, m, n), \text{split}(s, a, \text{fs}) \)
**AWK Examples**

- Print the only first two columns of every line in reverse order
  
  ```
  { print $2, $1 }
  ```

- Print the sum and average of numbers in the first field of each line
  
  ```
  { s += $1 }
  ```

  ```
  END { print "sum is", s;
  print " average is", s/NR }
  ```
Print every line with all of its fields reversed from left to right

    { for (i = NF; i > 0; --i)
      print $i }

Print every line between lines containing the words BEGIN and END. (No action defaults to print).

/BEGIN/, /END/
Word Counter

- Count & print the number of times a word occurs in a file
- The `tr` commands convert the file to all lowercase and remove everything except letters, spaces, dashes, and newlines.

```
tr A-Z a-z <file | tr -c -d 'a-z \n-' | awk '{ for(i=1; i<=NF; i++)
    count[+$i]++; } END { for(word in count)
    printf "%-10s %5d\n", word, count[word];}
```
A Simple Banking Calculator

BEGIN {balance=0;}
/deposit/ {balance+=2}
/withdraw/ {balance-=2}
END {printf "Balance: %.2f
", balance}
References

- *Mastering Regular Expressions* by Friedl (O’Reilly)
- *SED and AWK* by Dougherty and Robbins (O’Reilly)
- *Effective AWK Programming* by Robbins (O’Reilly)
- `info regex`
- `man regex, man re_format or man -s 5 regexp`
- The Perl documentation includes a good regular expression tutorial.