COSC 3750

Regex and the Standard Functions; tar

Kim Buckner

University of Wyoming

Feb. 17, 2022

Basics

- Read “man 7 regex,” you’ll love it.
- Especially –
  Within a bracket expression, a collating element enclosed in “[=” and “=]” is an equivalence class, standing for the sequences of characters of all collating elements equivalent to that one, including itself.

(more . . .)

- Actually the ones we will discuss are the "modern" POSIX.2 REs.
- These are "handier" than what we have been using, the "obsolete" REs.
- These are the ones also referred to as POSIX *extended* REs.

Big Differences

- The operators *, +, ? do NOT need a backslash.
- They mean:
  - ***** : 0 or more of the previous *atom*,
  - **+** : 1 or more of the previous *atom*, and
  - **?** : 0 or 1 of the previous *atom*.

(more . . .)

- An *atom* is:
  - an RE enclosed in (), also called a *subexpression*
  - a *bracket* expression,
  - ^ which matches the beginning of a line,
  - $ which matches the end of a line,
  - a special character following a \ , or
  - any other character by itself.

(more . . .)

- The bounds {m,n} work the same as we have already seen and require no backslash.
- Values must be unsigned integers between 0 and 255.
- First must not be larger than second.

(more . . .)

- Bracket expressions are like we have seen.
- To include ']', make it the first character.
- To include –, make it the first, last or second endpoint of a range.
- If a range starts with –, enclose it in "[." and ".]"
- Pretty much everything else loses its special meaning.

(more . . .)

- In a character class we can have a multicharacter sequence.
- Enclose it in "[." and ".]"
- Can use "[:" and ":]" to enclose the following: `alnum`, `alpha`, `blank`, `cntrl`, `digit`, `graph`, `lower`, `print`, `punct`, `space`, `upper`, `xdigit`.

(more . . .)

- Do not make REs longer than 256 bytes, not portable.
- Read the man page carefully and especially the BUGS.

So what?

- Can use C library functions to use regex on strings.
- There are four functions:
  - regcomp()
  - regexec()
  - regerror()
  - regfree()

REGEX functions

- MUST compile regex using regcomp().
- MUST CHECK RETURN VALUES!!!!!!!!
- Must make sure there is enough space for the matches.
- If reusing the compiled pattern need to regfree() them.

(more . . .)

- Do not depend on the case insensitive search being portable.
- It should work right but . . .
- Lets look at some simple code.

§reg.c

Compilation

- **regcomp()** takes three arguments.
- The first is a *regex_t** which will be used with **regexec()** to actually do the matching.
- Next slide is regex_t BEFORE compilation printed using gdb.

regex_t Before

```
{__buffer = 0x0, __allocated = 0, __used = 0,
  __syntax = 64870022513707 24, __fastmap = 0x0,
  __translate = 0x7ffff7ffe150 "",
  re_nsub = 140737354129744, __can_be_null = 0,
  __regs_allocated = 2, __fastmap_accurate = 1,
  __no_sub = 1, __not_bol = 0, __not_eol = 1,
  __newline_anchor = 0}
```

regex_t After

```
{__buffer = 0x602370, __allocated = 224, __used = 224,
__syntax = 242428, __fastmap = 0x602260 "",
__translate = 0x0, re_nsub = 0, __can_be_null = 0,
__regs_allocated = 0, __fastmap_accurate = 1,
__no_sub = 0, __not_bol = 0, __not_eol = 0,
__newline_anchor = 0}
```

(more . . .)

- The second argument to regcomp() is the actual pattern you want to match.
- The third is a set of flags. The man page lists them.
- The flags listed under POSIX regex matching are only for the regexec() function eflags.

regexec()

- This function does the work. It takes the compiled regex and returns any matches.
- The array of *regmatch_t* has to be allocated and the *nmatch* is the size of that array.
- The point of the array is to provide information on subexpressions.
- Subexpressions are those parts of the regex enclosed in parentheses.

(more . . .)

- The *eflags* argument is made up by ORing those flags mentioned above.
- regexec() can return three values, according to the gnu.org page discussing this.
- 0 for success (some match).
- REG_NOMATCH which is not an error.
- REG_ESPACE ran out of space.

regerror(), regfree()

- **regerror()** can give you exact (precise, verbose) error messages about compilation problems.
- **regfree()** cleans up a compiled regex_t so that it can properly be reused.

Using these.

- If the number of regular expressions you need to match are few, like say with the grep program, then these make sense.
- If they are potentionally a large number of them, then maybe you should consider something like flex.

Tape Achives

- That is what the utility **tar** creates.
- But, fortunately for us, it will also create archive files.
- Up until recently, if you did not specify a file to create, **tar** would fail with an error that meant it could not open the tape drive.
- Now that error is that **tar** refuses to write to the terminal.

Concept

- **tar** was designed to back up the file system to tape.
- That means that some things about it are a little odd.
- Each archive component it writes out must have enough data to reconstruct the object on disk.
- That means the format is very specific.

Format of archive

- Each component consists of a header block and then bytes from the disk.
- Have to understand that this is meant to be portable.
- So the documentation talks about “if supported” and so on.

The header

- Won't go into a lot of detail about the header at this time.
- It obviously contains the filename. That has some restrictions but mostly for very large names.
- Contains things like owner, group, permissions and so on.
- The archives also store links and directories.
- The header is 512 bytes.

Data

- File data is stored just as read off disk.
- This is NO formatting or modification. Allows even executables to be stored this way.
- This data is "padded" so that it is in even 512 byte increments.
- Because of the "tape" part.

Compression

- **tar** supports several compression algorithms.
- The most comman is *gzip*.
- Others are bzip2, xz, lzip, lzma, lzop, and compress.
- The standard file extension of gzip compression is ".gz" as in *archive.tar.gz*.
- More convenient is just ".tgz," this is what you will use.

Operation

- Three different versions of options, traditional (I usually use this), UNIX, and GNU
- The traditional and UNIX are similar but not the same.
- Traditional have no dashes and are a block of letters, UNIX uses dashes in the normal, and GNU is the "long" version like **--create**.

(more . . .)

- Common options
  - c – create
  - x – extract
  - t – tell
  - f – specify the archive filename
  - z – compress(ed) with gzip
  - u – update

(more . . .)

- If you use the GNU or UNIX versions and have an option that takes an argument, the argument MUST follow the option.
- Like:

```
tar -f filename -c -z file_list
```

- The traditional version does not require this BUT the arguments MUST be in the same order as the option

```
tar bfcz 2 archive.tgz file_list
```

Usage

- Tar by default is recursive. It includes all files in all subdirectories.
- Do NOT use

```
tar cf myarchive.tar *
```

  This may include the archive file in the archive.
- Normal operation is

```
tar zcf archive.tgz file_list
```

  to create a gzip compressed archive.

(more . . .)

- Don't know what is in an archive? Use tell.

```
tar tf archive.tgz
```

- I have found that recently **tar** will figure out that the file is compressed and NOT complain that you did not specify "z".

(more . . .)

- Extract is opposite of create

```
tar xf archive.tar
```

- And you can specify that just one or a few files be extracted by listing them after the archive name.

```
tar xf archive.tar file_list
```

(more . . .)

- And suppose you changed a file (or files) that are in the archive or want to add some new files

```
tar uf archive.tar file_list
```

- You can actually add ALL the original files and only the ones which have a newer modification time will be updated.