Regular Expression parsing in C

Ben Klemens

7 March 2014

Regular expressions are a means of expressing a pattern in text, like (a number followed by one or more letters) or (number-comma-space-number, with nothing else on the line); in basic regex-ese, these could be expressed as `[0-9]\+[[:alpha:]]\+` and `^[0-9]\+, [0-9]\+\$`. The POSIX standard specifies a set of C functions to parse the regular expressions whose grammar it defines, and those functions have been wrapped by hundreds of tools. I think it is literally true that I use them every day, either at the command line via POSIX-standard tools like `sed`, `awk`, and `grep`, or to deal with little text-parsing details in code. Maybe I need to find somebody's name in a file, or somebody sent me date ranges in single strings like `04Apr2009-12Jun2010` and I want to break that down into six usable fields, or I have a fictionalized treatise on cetology and need to find the chapter markers.

A reminder: if you want to break a string down into tokens before and after a single-character delimiter, `strtok` (entry #70) will work for you.

However, I resolved to not include a regular expression tutorial in this book. My Internet search for *"regular expression tutorial"* gives me 12,900 hits. On a Linux box, `man 7 regex` should give you a rundown, and if you have Perl installed, you have `man perlre` summarizing Perl-compatible regular expressions (PCREs). Or, Mastering Regular Expressions[1] gives an excellent book-length discussion of the topic. Here, I will cover how they work in POSIX's C library.

There are three major types of regular expression:

- Basic regular expressions (BREs) were the first draft, with only a few of the more common symbols having special meaning, like the `*` meaning zero or more of the previous atom, as in `[0-9]*` to represent an optional integer. Additional features required a backslash to indicate a special character: *one or more digits* is expressed via `\+`, so an integer preceded by a plus sign would be `+[0-9]\+`.
- Extended regular expressions (EREs) were the second draft, mostly taking special characters to be special without the backslashes, and plain text with a backslash. Now an integer preceded by a plus sign is `\+[0-9]+`.
- Perl has regular expressions at the core of the language, and its authors made several significant additions to the regex grammar, including a lookahead/lookbehind feature, non-greedy quantifiers that match the smallest possible match, and in-regex comments.

[1] `http://www.amazon.com/exec/obidos/tg/detail/-/0596528124/qid=1120157199/sr=8-1/ref=pd_bbs_ur_1?v=glance&s=books&n=507846`

The first two types of regular expression are implemented via a small set of functions. Being defined in POSIX, they are probably part of your standard library. PCREs are available via libpcre, which you can download from online or via your package manager. See `man pcreapi` for details of its functions. Glib provides a convenient wrapper for libpcre.

The example below, for example, compiles on Linux and Mac without any compiler flags beyond the usual necessities:

```
CFLAGS="−g −Wall −O3 −−std=gnu11" make regex
```

The POSIX and PCRE interfaces have a common four-step procedure:

- Compile the regex via `regcomp` or `pcre_compile`
- Run a string through the compiled regex via `regexec` or `pcre_exec`.
- If you marked substrings in the regular expression to pull out (see below), copy them from the base string using the offsets returned by the `regexec` or `pcre_exec` function.
- Free the internal memory used by the compiled regex.

The first two steps and the last step can be executed with a single line of code each, so if your question is only whether a string matches a given regular expression, then life is easy. I won't go into great detail about the flags and details of usage of `regcomp`, `regexec`, and `regfree` here because the page of the POSIX standard about them is reprinted in the Linux and BSD man pages (try `man regexec`), and there are *many* websites devoted to reprinting those man pages.

If you need to pull substrings, things get a little more complicated. Parens in a regex indicate that the parser should retrieve the match within that string (even if it only matches the null string). Thus, the ERE pattern `"(.*)o"` matches the string `"hello"`, and as a side-effect, stores the largest possible match for the `.*`, which is `hell`. BREs work the same but with backslashes : `"\(.*\)o"`. The third argument to the `regexec` function is the number of parenthesized subexpressions in the pattern; I call it `matchcount` in the example below. The fourth argument to `regexec` is an array of `matchcount+1 regmatch_t` elements. The `regmatch_t` has two elements: `rm_so` marking the start of the match and `rm_eo` marking the end. The zeroth element of the array will have the start and end of the match of the entire regex (imagine parens around the entire pattern), and subsequent elements have the start and end of each parenthesized subexpression, ordered by where their open-parens are in the pattern.

By way of foreshadowing, here is the header, listing the two utility functions provided by the sample code at the end of this segment. The `regex_match` function+macro+struct allows named, optional arguments, as per a previous blog post (entry #100).

```
typedef struct {
    const char *string;
    const char *regex;
    char ***substrings;
    _Bool use_case;
} regex_fn_s;

#define regex_match(...) regex_match_base((regex_fn_s){__VA_ARGS__})

int regex_match_base(regex_fn_s in);

char * search_and_replace(char const *base, char const*search, char const *replace);
```

We need a separate search and replace function, because POSIX doesn't provide one. Unless the replacement is exactly the same length as what it is replacing, the operation requires reallocating the original string. But we already have the tools to break a string into substrings, so the `search_and_replace` uses parenthesized substrings to break down a function into substrings, and then rebuilds a new string, inserting the replacement part at the appropriate point.

It returns `NULL` on no match, so you could do a global search and replace via

```
char *s2;
while((s2 = search_and_replace(long_string, pattern))){
    char *tmp = long_string;
    long_string = s2;
    free(tmp);
}
```

There are inefficiencies here: the `regex_match` function recompiles the string every time, and the global search-and-replace would be more efficient if it used the fact that everything up to `result[1].rm_eo` does not need to be re-searched. In this case, we can use C as a prototyping language for C: write the easy version, and if the profiler shows that these inefficiencies are a problem, replace them with more efficient code.

Here is the code. The lines where key events in the above discussion occur are marked, with some additional notes at the end. There is a test function at the end showing some simple uses of the provided functions.

To compile this, you'll need `stopif.h`; get it from Github[2].

```
#define _GNU_SOURCE //cause stdio.h to include asprintf
#include "stopif.h"
#include <regex.h>
#include "regex_fns.h"
#include <string.h> //strlen
#include <stdlib.h> //malloc, memcpy

static int count_parens(const char *string){ //(1)
    int out = 0;
    int last_was_backslash = 0;
    for(const char *step=string; *step !='\0'; step++){
        if (*step == '\\' && !last_was_backslash){
            last_was_backslash = 1;
            continue;
        }
```

[2]`https://github.com/b-k/21st-Century-Examples/blob/master/stopif.h`

```
        if (*step == ')' && !last_was_backslash)
            out++;
        last_was_backslash = 0;
    }
    return out;
}

int regex_match_base(regex_fn_s in){
    Stopif(!in.string, return -1, "NULL string input");
    Stopif(!in.regex, return -2, "NULL regex input");

    regex_t re;
    int matchcount = 0;
    if (in.substrings) matchcount = count_parens(in.regex);
    regmatch_t result[matchcount+1];
    int compiled_ok = !regcomp(&re, in.regex, REG_EXTENDED //(2)
                                            + (in.use_case ? 0 : REG_ICASE)
                                            + (in.substrings ? 0 : REG_NOSUB) );
    Stopif(!compiled_ok, return -3, "This regular expression didn't compile: \"%s\"", in.
         regex);

    int found = !regexec(&re, in.string, matchcount+1, result, 0); //(3)
    if (!found) return 0;
    if (in.substrings){
         *in.substrings = malloc(sizeof(char*) * matchcount);
         char **substrings = *in.substrings;
        //match zero is the whole string; ignore it.
        for (int i=0; i< matchcount; i++){
            if (result[i+1].rm_eo > 0){//GNU peculiarity: match-to-empty marked with -1.
                int length_of_match = result[i+1].rm_eo - result[i+1].rm_so;
                substrings[i] = malloc(strlen(in.string)+1);
                memcpy(substrings[i], in.string + result[i+1].rm_so, length_of_match);
                substrings[i][length_of_match] = '\0';
            } else { //empty match
                substrings[i] = malloc(1);
                substrings[i][0] = '\0';
            }
        }
        in.string += result[0].rm_eo; //end of whole match;
    }
    regfree(&re); //(4)
    return matchcount;
}

char * search_and_replace(char const *base, char const*search, char const *replace){
    char *regex, *out;
    asprintf(&regex, "(.*)(%s)(.*)", search); //(5)
    char **substrings;
    int match_ct = regex_match(base, regex, &substrings);
    if(match_ct < 3) return NULL;
    asprintf(&out, "%s%s%s", substrings[0], replace, substrings[2]);
    for (int i=0; i< match_ct; i++)
        free(substrings[i]);
    free(substrings);
    return out;
}

#ifdef test_regexes
int main(){
    char **substrings;

    int match_ct = regex_match("Hedonism by the alps, savory foods at every meal.",
                "([He]*)do.*a(.*)s, (.*)or.* ([em]*)al", &substrings);
    printf("%i matches:\n", match_ct);
    for (int i=0; i< match_ct; i++){
        printf("[%s] ", substrings[i]);
        free(substrings[i]);
    }
    free(substrings);
    printf("\n\n");

    match_ct = regex_match("", "([[:alpha:]]+) ([[:alpha:]]+)", &substrings);
    Stopif(match_ct != 0, return 1, "Error: matched a blank");

    printf("%s", search_and_replace("Plants\n", "l", ""));
}
#endif
```

(1) You need to send `regexec` an allocated array to hold substring matches, and its length, meaning that you need to know how many substrings there are. This function takes in an ERE and counts open-parens that aren't escaped by a backslash.

(2) Here we compile the regex to a `regex_t`. The function is inefficient because the regex gets recompiled every time. It is left as an exercise to the reader to cache already-compiled regular expressions.

(3) Here is the `regexec` use. If you just want to know whether there is a match or not, you can send NULL and 0 as the list of matches and its length.

(4) Don't forget to free the internal memory used by the `regex_t`.

(5) The search-and-replace works by breaking down the input string into (everything before the match)(the match)(everything after the match). This is the regex representing that.