JavaScript Shell Command Line Processing

Finnbarr P. Murphy

(fpm@fpmurphy.com)

I like to use a JavaScript shell when developing custom JavaScript functions which contain more than a few lines of code. Logic and coding errors are easier to spot. The development cycle is also faster because you can avoid the cycle of loading code onto a web server, reloading a web page, etc.

What is a JavaScript shell you may ask? First some background. JavaScript is a complex full-featured weakly typed object- based functional programming language originally developed by Brendan Eich in 1995 while working on the Netscape Navigator browser. It is most frequently used in client-side web applications but is also used to enable scripting access to embedded objects in other applications. The langauge has been standardized in the ECMA-262 (ECMAScript) specification. The current version is Edition 5 which was published in December 2009. Formally, Javascript is a dialect of ECMAScript whose langauge specification is controlled by the Mozilla Foundation. There are other dialects including ActionScript which the scripting language used in Adobe Flash. Javascript is still evolving as a language and several versions are in daily use. The current version is JavaScript 1.8.

The Mozilla Firefox browser has a C language JavaScript engine. It was orginally called Javascript Reference (JSRef) but nowadays is known as SpiderMonkey. Other Mozilla products also use this engine and it is available to the public under a MPL/GPL/LGPL tri-license. The current version, SpiderMonkey 1.7, conforms to JavaScript 1.8 which is a superset of ECMA-262 Edition 3. It consists of a library (or DLL) containing the JavaScript runtime (compiler, interpreter, decompiler, garbage collector, atom manager and standard classes) engine. The SpiderMonkey codebase has no dependencies on the rest of the Mozilla codebase. The codebase also contains the routines for a simple user interface which can be linked to the runtime library in order to make a command line shell.

The SpiderMonkey JavaScript shell does not include built-in support for command line argument processing. Recently I decided to add such support so that I did not have to keep writing essentially similar code to parse a script's command line arguments to check for valid arguments and options. The main design decision I made was to follow the traditional BSD and POSIX command line argument format and parse the command line arguments and options using a *getopts*--like function.

Rather than re-invent the wheel I decided to try to port a version of the BSD *getopt(3)* function to JavaScript. The BSD *getopt* function has stood the test of time and it's internal logic has proven to be extremely robust. Here the C language source:

```
/*
 * Copyright (c) 1987, 1993, 1994
 *    The Regents of the University of California.  All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
```



```
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 4. Neither the name of the University nor the names of its contributors
 *    may be used to endorse or promote products derived from this software
 *    without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 */
#include &amp;lt;stdio.h&amp;gt;
#include &amp;lt;stdlib.h&amp;gt;
#include &amp;lt;string.h&amp;gt;
#include &amp;lt;unistd.h&amp;gt;
int    opterr = 1,        /* if error message should be printed */
       optind = 1,        /* index into parent argv vector */
       optopt,            /* character checked for validity */
       optreset;          /* reset getopt */
char    *optarg;          /* argument associated with option */
#define    BADCH    (int)'?'
#define    BADARG    (int)':'
#define    EMSG    ""
/*
 * getopt --
 *    Parse argc/argv argument vector.
 */
int
getopt(nargc, nargv, ostr)
    int nargc;
    char * const nargv[];
    const char *ostr;
{
    static char *place = EMSG;            /* option letter processing */
    char *oli;                            /* option letter list index */
    if (optreset || *place == 0) {        /* update scanning pointer */
        optreset = 0;
        place = nargv[optind];
        if (optind >= nargc || *place++ != '-') {
            /* Argument is absent or is not an option */
            place = EMSG;
            return (-1);
        }
        optopt = *place++;
        if (optopt == '-' &amp;&amp; *place == 0) {
            /* "--" => end of options */
            ++optind;
            place = EMSG;
            return (-1);
        }
        if (optopt == 0) {
            /* Solitary '-', treat as a '-' option
               if the program (eg su) is looking for it. */
            place = EMSG;
            if (strchr(ostr, '-') == NULL)
                return (-1);
            optopt = '-';
        }
    } else
        optopt = *place++;
    /* See if option letter is one the caller wanted... */
```



```
    if (optopt == ':' || (oli = strchr(ostr, optopt)) == NULL) {
        if (*place == 0)
            ++optind;
        if (opterr &amp;amp;&amp;amp; *ostr != ':')
            (void)fprintf(stderr,
                "%s: illegal option -- %c\n", _getprogname(),
                optopt);
        return (BADCH);
    }
    /* Does this option need an argument? */
    if (oli[1] != ':') {
        /* don't need argument */
        optarg = NULL;
        if (*place == 0)
            ++optind;
    } else {
        /* Option-argument is either the rest of this argument or the
           entire next argument. */
        if (*place)
            optarg = place;
        else if (nargc > ++optind)
            optarg = nargv[optind];
        else {
            /* option-argument absent */
            place = EMSG;
            if (*ostr == ':')
                return (BADARG);
            if (opterr)
                (void)fprintf(stderr,
                    "%s: option requires an argument -- %c\n",
                    _getprogname(), optopt);
            return (BADCH);
        }
        place = EMSG;
        ++optind;
    }
    return (optopt);            /* return option letter */
}
```

And here is the functionally equivalent JavaScript function which I developed:

```


//
//  getopt.js    Finnbarr P. Murphy March 2010
//
//  Based on BSD getopt.c  Use subject to BSD license.
//
//  For details of how to use this function refer to
//  the BSD man page for getopt(3). GNU-style long
//  options are not supported.
//
var opterr = 1;                         // print error message
var optind = 0;                         // index into parent argv array
var optopt = "";                        // character checked for validity
var optreset = 0;                       // reset getopt
var optarg = "";                        // option argument
function getopt(nargv, ostr)
{
    if ( typeof getopt.place == 'undefined' ) {
        getopt.place =  "";             // static string, option letter processing
        getopt.iplace = 0;              // index into string
    }
    var oli;                            // option letter list index
    if (optreset > 0 || getopt.iplace == getopt.place.length) {
        optreset = 0;
        getopt.place = nargv[optind]; getopt.iplace = 0;
```



```
        if (optind >= nargv.length || getopt.place.charAt(getopt.iplace++) != "-") {
            // argument is absent or is not an option
            getopt.place = ""; getopt.iplace = 0;
            return("");
        }
        optopt = getopt.place.charAt(getopt.iplace++);
        if (optopt == '-' &amp;&amp; getopt.iplace == getopt.place.length) {
            // "--" => end of options
            ++optind;
            getopt.getopt.place = ""; getopt.getopt.iplace = 0;
            return("");
        }
        if (optopt == 0) {
            // Solitary '-', treat as a '-' option
            getopt.place = ""; getopt.iplace = 0;
            if (ostr.indexOf('-') == -1)
                return("");
            optopt = '-';
        }
    } else
         optopt = getopt.place.charAt(getopt.iplace++);
    // see if option letter is what is wanted
    if (optopt == ':' || (oli = ostr.indexOf(optopt)) == -1) {
        if (getopt.iplace == getopt.place.length)
            ++optind;
        if (opterr &amp;&amp; ostr.charAt(0) != ':')
            print("illegal option -- " + optopt);
        return ('?');
    }
    // does this option require an argument?
    if (ostr.charAt(oli + 1) != ':') {
         // does not need argument
         optarg = null;
         if (getopt.iplace == getopt.place.length)
             ++optind;
    } else {
       //  Option-argument is either the rest of this argument or the entire next argument.
       if (getopt.iplace < getopt.place.length) {
            optarg = getopt.place.substr(getopt.iplace);
       } else if (nargv.length > ++optind) {
            optarg = nargv[optind];
       } else {
            // option argument absent
            getopt.place = ""; getopt.iplace = 0;
            if (ostr.charAt(0) == ':') {
                return (':');
           }
            if (opterr)
                print("option requires an argument -- " + optopt);
            return('?');
        }
        getopt.place = ""; getopt.iplace = 0;
        ++optind;
    }
    return (optopt);
}
```

The C language version of the *getopt* function requires the use of a static pointer variable, *place* to perform its magic (see line 54 of *getopt.c*.) Static variables such as *place* when used in functions maintain their value between function calls. The JavaScript language does not support static variables or pointers however. The workaround for the static variable issue was fairly trivial. In JavaScript functions are also objects and thus we can create a variable that is a member of a function. Since the variable is now part of an object, it's value is retained between function calls. See lines 19 – 22 of *getopt.js*. The workaround for the no pointers issue was to replace the pointer with a string variable *place* together with a variable, *iplace*, which is used to index into this string.

Here is a trivial example of how to use the *getopt* function in a Javscript shell script.

```
$ cat test.js
#!/bin/js
load("getopt.js");
var opt;
while ((opt = getopt(arguments, "ghf:v")) != '') {
    switch (opt) {
       case 'f':
           print("f option found. Argument is: " + optarg);
           break;
       case 'g':
           print("g option found");
           break;
       case 'h':
           print("usage: " + environment["_"] + " [-f filename] [-g] [-v]");
           quit(0);
       case 'v':
           print("v option found");
           break;
       case ':':
           print("Error - Option needs a value: " + optopt);
           quit(1);
       case '?':
           print("Error - No such option: " + optopt);
           quit(1);
     }
}
quit(0);
$
$ ./test.js
$ ./test.js -h
usage: ./test.js [-f filename] [-g] [-v]
$ ./test.js -j
illegal option -- j
Error - No such option: j
$ ./test.js -gvf finn
g option found
v option found
f option found. Argument is: finn
$ ./test.js -g -v -f finn
g option found
v option found
f option found. Argument is: finn
$
```


If you look closely at this script, you may notice a number of unusual JavaScript constructs. First of all, there are a number of calls to a *quit()* function. The JavaScript shell does not natively support the classical *exit()* function. Instead the developers choose to supply a function called *quit()* which is essentially, for all practical purposes the same as the C *exit()* function. Another unusual construct is how the *getopts()* function is made available to this script using the *load()* function. The *load()* is another function which is only available in the JavaScript shell and is used to include other source code files in a script. In this case, *load("getopts.js")* makes the *getopts()* function and related variables, i.e. *optarg*, *optreset*, etc., available to this script. Another function provided by the JavaScript shell is the *print()* function which writes to *stdout*.

If you look at the first line of this script you will see that the *shebang* method of invoking a script, in this case a JavaScript script, is supported just like with regular shell scripts. Note the use of *environment["_"]* in the usage message on line 17 to output the name of the script. It turns out that the *callee* property of the *arguments* object was deprecated in JavaScript 1.4 and the *callee* property is marked obsolete. Thus neither of those properties are available in the JavaScript shell

to return the name of the calling script. Fortunately, the developers of the shell decided to make the user's environment available to the shell by means of the *environment* object.

Here is simple script to list out environment strings available to you in the JavaScript shell:

```
#!/bin/js
var key;
for ( key in environment ) {
   if (typeof environment[key] == 'string') {
      print(key + ' ---> ' + environment[key]);
   }
}
```

And here is a portion of the output when this script is run on a GNU/Linux platform:

```
USERNAME ---> fpm
GDM_KEYBOARD_LAYOUT ---> us
LANG ---> en_US.UTF-8
SHLVL ---> 3
HOME ---> /home/fpm
LOGNAME ---> fpm
COLORTERM ---> gnome-terminal
PWD --> /work/js/tmp
_ ---> ./env.js
OLDPWD ---> /work/js
```

From this output, you can see that I am logged in as *fpm*, my current directory is */home/fpm/js/tmp* and the script that I executed to obtain this output is called *env.js* and was executed from my current directory. Thus, in the above getopts example script I used the _ key to retrieve the name of the script.

Well, that is all I have time to write about this topic at the moment. I encourage you to add the JavaScript shell to your tool set and become familiar with it. I hope that you too will find the *getopts* functionality useful when you are writing your own JavaScript shell scripts. I would appreciate it if you would let me know if you find any bugs in the code.

