

```

*****
16906 Sat Jun 13 17:15:03 2015
new/usr/src/cmd/spell/spellprog.c
3727 british people can't spell
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License, Version 1.0 only
6  * (the "License"). You may not use this file except in compliance
7  * with the License.
8  *
9  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
10 * or http://www.opensolaris.org/os/licensing.
11 * See the License for the specific language governing permissions
12 * and limitations under the License.
13 *
14 * When distributing Covered Code, include this CDDL HEADER in each
15 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
16 * If applicable, add the following below this CDDL HEADER, with the
17 * fields enclosed by brackets "[]" replaced with your own identifying
18 * information: Portions Copyright [yyyy] [name of copyright owner]
19 *
20 * CDDL HEADER END
21 */
22 /*
23 * Copyright 2015 Gary Mills
24 * Copyright 2005 Sun Microsystems, Inc. All rights reserved.
25 * Use is subject to license terms.
26 */

28 /*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
29 /*      All Rights Reserved      */

30 #pragma ident      "%Z%M% %I%      %E% SMI"

31 #include <stdlib.h>
32 #include <unistd.h>
33 #include <limits.h>
34 #include <string.h>
35 #include <stdio.h>
36 #include <ctype.h>
37 #include <locale.h>
38 #include "hash.h"

40 #define Tolower(c) (isupper(c)?tolower(c):c)
41 #define DLEV 2

43 /*
44  * ANSI prototypes
45  */
46 static int      ily(char *, char *, char *, int);
47 static int      s(char *, char *, char *, int);
48 static int      es(char *, char *, char *, int);
49 static int      subst(char *, char *, char *, int);
50 static int      nop(void);
51 static int      bility(char *, char *, char *, int);
52 static int      i_to_y(char *, char *, char *, int);
53 static int      CCe(char *, char *, char *, int);
54 static int      y_to_e(char *, char *, char *, int);
55 static int      strip(char *, char *, char *, int);
56 static int      ize(char *, char *, char *, int);
57 static int      tion(char *, char *, char *, int);
58 static int      an(char *, char *, char *, int);
59 int      prime(char *);

```

```

61 static void      ise(void);
60 static int      tryword(char *, char *, int);
61 static int      trypref(char *, char *, int);
62 static int      trysuff(char *, int);
63 static int      vowel(int);
64 static int      dict(char *, char *);
65 static int      monosyl(char *, char *);
66 static int      VCe(char *, char *, char *, int);
67 static char      *skipv(char *);
70 static void      ztos(char *);

69 struct suftab {
72 static struct suftab {
70     char *suf;
71     int (*pl)();
72     int nl;
73     char *dl;
74     char *al;
75     int (*p2)();
76     int n2;
77     char *d2;
78     char *a2;
79 };

81 static struct suftab sufa[] = {
82 } suftab[] = {
82     {"ssen", ily, 4, "-y+iness", "+ness"},
83     {"ssel", ily, 4, "-y+i+less", "+less"},
84     {"se", s, 1, "", "+s", es, 2, "-y+ies", "+es"},
85     {"s'", s, 2, "", "+s'"},
86     {"s", s, 1, "", "+s"},
87     {"ecn", subst, 1, "-t+ce", ""},
88     {"ycn", subst, 1, "-t+cy", ""},
89     {"ytilb", nop, 0, "", ""},
90     {"ytilib", bility, 5, "-le+ility", ""},
91     {"elbaif", i_to_y, 4, "-y+i+able", ""},
92     {"elba", CCe, 4, "-e+able", "+able"},
93     {"yti", CCe, 3, "-e+ity", "+ity"},
94     {"ylb", y_to_e, 1, "-e+y", ""},
95     {"yl", ily, 2, "-y+ily", "+ly"},
96     {"laci", strip, 2, "", "+al"},
97     {"latnem", strip, 2, "", "+al"},
98     {"lanoi", strip, 2, "", "+al"},
99     {"tnem", strip, 4, "", "+ment"},
100    {"gni", CCe, 3, "-e+ing", "+ing"},
101    {"reta", nop, 0, "", ""},
102    {"retc", nop, 0, "", ""},
103    {"re", strip, 1, "", "+r", i_to_y, 2, "-y+ier", "+er"},
104    {"de", strip, 1, "", "+d", i_to_y, 2, "-y+ied", "+ed"},
105    {"citsi", strip, 2, "", "+ic"},
106    {"citi", ize, 1, "-ic+e", ""},
107    {"cihparg", i_to_y, 1, "-y+ic", ""},
108    {"tse", strip, 2, "", "+st", i_to_y, 3, "-y+iest", "+est"},
109    {"cirtem", i_to_y, 1, "-y+ic", ""},
110    {"yrtem", subst, 0, "-er+ry", ""},
111    {"cigol", i_to_y, 1, "-y+ic", ""},
112    {"tsigol", i_to_y, 2, "-y+ist", ""},
113    {"tsi", CCe, 3, "-e+ist", "+ist"},
114    {"msi", CCe, 3, "-e+ism", "+ist"},
115    {"noitacifi", i_to_y, 6, "-y+ication", ""},
116    {"noitazi", ize, 4, "-e+ation", ""},
117    {"rota", tion, 2, "-e+or", ""},
118    {"rotc", tion, 2, "", "+or"},
119    {"noit", tion, 3, "-e+ion", "+ion"},
120    {"naino", an, 3, "", "+ian"},
121    {"na", an, 1, "", "+n"},

```

```

122     {"evi", subst, 0, "-ion+ive", ""},
123     {"ezi", CCe, 3, "-e+ize", "+ize"},
124     {"pihs", strip, 4, "", "+ship"},
125     {"dooh", ily, 4, "-y+ihood", "+hood"},
126     {"luf", ily, 3, "-y+iful", "+ful"},
127     {"ekil", strip, 4, "", "+like"},
128     0
129 };

131 static struct suftab sufb[] = {
132     {"ssen", ily, 4, "-y+iness", "+ness"},
133     {"ssel", ily, 4, "-y+i+less", "+less"},
134     {"se", s, 1, "", "+s", es, 2, "-y+ies", "+es"},
135     {"s'", s, 2, "", "+'s"},
136     {"s", s, 1, "", "+s"},
137     {"ecn", subst, 1, "-t+ce", ""},
138     {"ycn", subst, 1, "-t+cy", ""},
139     {"ytilb", nop, 0, "", ""},
140     {"ytilib", bility, 5, "-le+ility", ""},
141     {"elbaif", i_to_y, 4, "-y+i+able", ""},
142     {"elba", CCe, 4, "-e+able", "+able"},
143     {"yti", CCe, 3, "-e+ity", "+ity"},
144     {"ylb", y_to_e, 1, "-e+y", ""},
145     {"yl", ily, 2, "-y+ily", "+ly"},
146     {"laci", strip, 2, "", "+al"},
147     {"latnem", strip, 2, "", "+al"},
148     {"lanoi", strip, 2, "", "+al"},
149     {"tnem", strip, 4, "", "+ment"},
150     {"gni", CCe, 3, "-e+ing", "+ing"},
151     {"reta", nop, 0, "", ""},
152     {"retc", nop, 0, "", ""},
153     {"re", strip, 1, "", "+r", i_to_y, 2, "-y+ier", "+er"},
154     {"de", strip, 1, "", "+d", i_to_y, 2, "-y+ied", "+ed"},
155     {"citsi", strip, 2, "", "+ic"},
156     {"citi", ize, 1, "-ic+e", ""},
157     {"cihparg", i_to_y, 1, "-y+ic", ""},
158     {"tse", strip, 2, "", "+st", i_to_y, 3, "-y+iest", "+est"},
159     {"cirtem", i_to_y, 1, "-y+ic", ""},
160     {"yrtem", subst, 0, "-er+ry", ""},
161     {"cigol", i_to_y, 1, "-y+ic", ""},
162     {"tsigol", i_to_y, 2, "-y+ist", ""},
163     {"tsi", CCe, 3, "-e+ist", "+ist"},
164     {"msi", CCe, 3, "-e+ism", "+ist"},
165     {"noitacifi", i_to_y, 6, "-y+ication", ""},
166     {"noitasi", ize, 4, "-e+ation", ""},
167     {"rota", tion, 2, "-e+or", ""},
168     {"rotc", tion, 2, "", "+or"},
169     {"noit", tion, 3, "-e+ion", "+ion"},
170     {"naino", an, 3, "", "+ian"},
171     {"na", an, 1, "", "+n"},
172     {"evi", subst, 0, "-ion+ive", ""},
173     {"esi", CCe, 3, "-e+ise", "+ise"},
174     {"pihs", strip, 4, "", "+ship"},
175     {"dooh", ily, 4, "-y+ihood", "+hood"},
176     {"luf", ily, 3, "-y+iful", "+ful"},
177     {"ekil", strip, 4, "", "+like"},
178     0
179 };

181 static char *preftab[] = {
182     "anti",
183     "auto",
184     "bio",
185     "counter",
186     "dis",
187     "electro",

```

```

188     "en",
189     "fore",
190     "geo",
191     "hyper",
192     "intra",
193     "inter",
194     "iso",
195     "kilo",
196     "magneto",
197     "meta",
198     "micro",
199     "mid",
200     "milli",
201     "mis",
202     "mono",
203     "multi",
204     "non",
205     "out",
206     "over",
207     "photo",
208     "poly",
209     "pre",
210     "pseudo",
211     "psycho",
212     "re",
213     "semi",
214     "stereo",
215     "sub",
216     "super",
217     "tele",
218     "thermo",
219     "ultra",
220     "under", /* must precede un */
221     "un",
222     0
223 };

225 static int bflag;
226 static int vflag;
227 static int xflag;
228 static struct suftab *suftab;
229 static char *prog;
230 static char word[LINE_MAX];
231 static char original[LINE_MAX];
232 static char *deriv[LINE_MAX];
233 static char affix[LINE_MAX];
234 static FILE *file, *found;
235 /*
236 *     deriv is stack of pointers to notes like +micro +ed
237 *     affix is concatenated string of notes
238 *     the buffer size 141 stems from the sizes of original and affix.
239 */

241 /*
242 *     in an attempt to defray future maintenance misunderstandings, here is
243 *     an attempt to describe the input/output expectations of the spell
244 *     program.
245 *
246 *     spellprog is intended to be called from the shell file spell.
247 *     because of this, there is little error checking (this is historical, not
248 *     necessarily advisable).
249 *
250 *     spellprog options hashed-list pass
251 *
252 *     the hashed-list is a list of the form made by spellin.
253 *     there are 2 types of hashed lists:

```

```

254 *      1. a stop list: this specifies words that by the rules embodied
255 *      in spellprog would be recognized as correct, BUT are really
256 *      errors.
257 *      2. a dictionary of correctly spelled words.
258 *      the pass number determines how the words found in the specified
259 *      hashed-list are treated. If the pass number is 1, the hashed-list is
260 *      treated as the stop-list, otherwise, it is treated as the regular
261 *      dictionary list. in this case, the value of "pass" is a filename. Found
262 *      words are written to this file.
263 *
264 *      In the normal case, the filename = /dev/null. However, if the v option
265 *      is specified, the derivations are written to this file.
266 *      The spellprog looks up words in the hashed-list; if a word is found, it
267 *      is printed to the stdout. If the hashed-list was the stop-list, the
268 *      words found are presumed to be misspellings. in this case,
269 *      a control character is printed ( a "-" is appended to the word.
270 *      a hyphen will never occur naturally in the input list because deroff
271 *      is used in the shell file before calling spellprog.)
272 *      If the regular spelling list was used (hlista or hlistb), the words
273 *      are correct, and may be ditched. (unless the -v option was used -
274 *      see the manual page).
275 *
276 *      spellprog should be called twice : first with the stop-list, to flag all
277 *      a priori incorrectly spelled words; second with the dictionary.
278 *
279 *      spellprog hstop 1 |\
280 *      spellprog hlista /dev/null
281 *
282 *      for a complete scenario, see the shell file: spell.
283 *
284 */

286 int
287 main(int argc, char **argv)
288 {
289     char *ep, *cp;
290     char *dp;
291     int fold;
292     int c, j;
293     int pass;

295     /* Set locale environment variables local definitions */
296     (void) setlocale(LC_ALL, "");
297     #if !defined(TEXT_DOMAIN) /* Should be defined by cc -D */
298     #define TEXT_DOMAIN "SYS_TEST" /* Use this only if it wasn't */
299     #endif
300     (void) textdomain(TEXT_DOMAIN);

303     prog = argv[0];
304     while ((c = getopt(argc, argv, "bvxn")) != EOF) {
305         switch (c) {
306             case 'b':
307                 bflag++;
308                 ise();
309                 break;
310             case 'v':
311                 vflag++;
312                 break;
313             case 'x':
314                 xflag++;
315                 break;
316         }
317     }

318     argc -= optind;

```

```

319     argv = &argv[optind];

321     if ((argc < 2) || !prime(*argv)) {
322         (void) fprintf(stderr,
323             gettext("%s: cannot initialize hash table\n"), prog);
324         exit(1);
325     }
326     argc--;
327     argv++;

329     /* Select the correct suffix table */
330     suftab = (bflag == 0) ? sufa : sufb;

332 /*
333 *      if pass is not 1, it is assumed to be a filename.
334 *      found words are written to this file.
335 */
336     pass = **argv;
337     if (pass != '1')
338         found = fopen(*argv, "w");

340     for (;;) {
341         affix[0] = 0;
342         file = stdout;
343         for (ep = word; (*ep = j = getchar()) != '\n'; ep++)
344             if (j == EOF)
345                 exit(0);
346 /*
347 *      here is the hyphen processing. these words were found in the stop
348 *      list. however, if they exist as is, (no derivations tried) in the
349 *      dictionary, let them through as correct.
350 */
351 /*
352         if (ep[-1] == '-') {
353             *--ep = 0;
354             if (!tryword(word, ep, 0))
355                 (void) fprintf(file, "%s\n", word);
356             continue;
357         }
358         for (cp = word, dp = original; cp < ep; )
359             *dp++ = *cp++;
360         *dp = 0;
361         fold = 0;
362         for (cp = word; cp < ep; cp++)
363             if (islower(*cp))
364                 goto lcase;
365         if (((ep - word) == 1) &&
366             ((word[0] == 'A') || (word[0] == 'I')))
367             continue;
368         if (trypref(ep, ".", 0))
369             goto foundit;
370         ++fold;
371         for (cp = original+1, dp = word+1; dp < ep; dp++, cp++)
372             *dp = Tolower(*cp);
373 lcase:
374         if (((ep - word) == 1) && (word[0] == 'a'))
375             continue;
376         if (trypref(ep, ".", 0) || trysuff(ep, 0))
377             goto foundit;
378         if (isupper(word[0])) {
379             for (cp = original, dp = word; *dp = *cp++; dp++)
380                 if (fold) *dp = Tolower(*dp);
381             word[0] = Tolower(word[0]);
382             goto lcase;
383         }
384         (void) fprintf(file, "%s\n", original);

```

```

385         continue;
387 foundit:
388         if (pass == 'l')
389             (void) fprintf(file, "%s-\n", original);
390         else if (affix[0] != 0 && affix[0] != '.') {
391             file = found;
392             (void) fprintf(file, "%s\t%s\n", affix,
393                 original);
394         }
395     }
396 }

398 /*
399  * strip exactly one suffix and do
400  * indicated routine(s), which may recursively
401  * strip suffixes
402  */

404 static int
405 trysuff(char *ep, int lev)
406 {
407     struct suftab *t;
408     char *cp, *sp;

410     lev += DLEV;
411     deriv[lev] = deriv[lev-1] = 0;
412     for (t = &suftab[0]; t != 0 && (sp = t->suf) != 0; t++) {
358     for (t = &suftab[0]; (sp = t->suf) != 0; t++) {
413         cp = ep;
414         while (*sp)
415             if (*--cp != *sp++)
416                 goto next;
417         for (sp = cp; --sp >= word && !vowel(*sp); )
418             ;
363     for (sp = cp; --sp >= word && !vowel(*sp); );
419     if (sp < word)
420         return (0);
421     if ((*t->pl)(ep-t->n1, t->d1, t->a1, lev+1))
422         return (1);
423     if (t->p2 != 0) {
424         deriv[lev] = deriv[lev+1] = 0;
425         return ((*t->p2)(ep-t->n2, t->d2, t->a2, lev));
426     }
427     return (0);
428 next:;
429     }
430     return (0);
431 }
    unchanged_portion_omitted

706 /* crummy way to Britishise */
707 static void
708 ise(void)
709 {
710     struct suftab *p;

712     for (p = suftab; p->suf; p++) {
713         ztos(p->suf);
714         ztos(p->d1);
715         ztos(p->a1);
716     }
717 }

719 static void
720 ztos(char *s)

```

```

721 {
722     for (; *s; s++)
723         if (*s == 'z')
724             *s = 's';
725 }

761 static int
762 dict(char *bp, char *ep)
763 {
764     int temp, result;
765     if (xflag)
766         (void) fprintf(stdout, "=%.*s\n", ep-bp, bp);
767     temp = *ep;
768     *ep = 0;
769     result = hashlook(bp);
770     *ep = temp;
771     return (result);
772 }
    unchanged_portion_omitted

```