```
**********************************************************
   47387 Fri Aug  9 07:47:42 2013
new/usr/src/cmd/find/find.c
3965 find does not support -delete option
**********************************************************
```

```
   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 (the "License").
   6  * You may not use this file except in compliance with the License.
   7  *
   8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
   9  * or http://www.opensolaris.org/os/licensing.
  10  * See the License for the specific language governing permissions
  11  * and limitations under the License.
  12  *
  13  * When distributing Covered Code, include this CDDL HEADER in each
  14  * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  15  * If applicable, add the following below this CDDL HEADER, with the
  16  * fields enclosed by brackets "[]" replaced with your own identifying
  17  * information: Portions Copyright [yyyy] [name of copyright owner]
  18  *
  19  * CDDL HEADER END
  20  */
  21 /*
  22  * Copyright (c) 1988, 2010, Oracle and/or its affiliates. All rights reserved.
  23  * Copyright 2012 Nexenta Systems, Inc. All rights reserved.
  24  * Copyright (c) 2013 Andrew Stormont.  All rights reserved.
  25  */


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


  32 /*      Parts of this product may be derived from              */
  33 /*      Mortice Kern Systems Inc. and Berkeley 4.3 BSD systems. */
  34 /*      licensed from  Mortice Kern Systems Inc. and           */
  35 /*      the University of California.                          */

  37 /*
  38  * Copyright 1985, 1990 by Mortice Kern Systems Inc.  All rights reserved.
  39  */

  41 #include <stdio.h>
  42 #include <errno.h>
  43 #include <pwd.h>
  44 #include <grp.h>
  45 #include <sys/types.h>
  46 #include <sys/stat.h>
  47 #include <sys/param.h>
  48 #include <sys/acl.h>
  49 #include <limits.h>
  50 #include <unistd.h>
  51 #include <stdlib.h>
  52 #include <locale.h>
  53 #include <string.h>
  54 #include <strings.h>
  55 #include <ctype.h>
  56 #include <wait.h>
  57 #include <fnmatch.h>
  58 #include <langinfo.h>
  59 #include <ftw.h>
  60 #include <libgen.h>
  61 #include <err.h>
```



```
  62 #include <regex.h>
  63 #include "getresponse.h"

  65 #define A_DAY           (long)(60*60*24)        /* a day full of seconds */
  66 #define A_MIN           (long)(60)
  67 #define BLKSIZ          512
  68 #define round(x, s)     (((x)+(s)-1)&~((s)-1))
  69 #ifndef FTW_SLN
  70 #define FTW_SLN         7
  71 #endif
  72 #define LINEBUF_SIZE            LINE_MAX        /* input or output lines */
  73 #define REMOTE_FS               "/etc/dfs/fstypes"
  74 #define N_FSTYPES               20
  75 #define SHELL_MAXARGS           253     /* see doexec() for description */

  77 /*
  78  * This is the list of operations
  79  * F_USER and F_GROUP are named to avoid conflict with USER and GROUP defined
  80  * in sys/acl.h
  81  */

  83 enum Command
  84 {
  85         PRINT,
  86         ACL, AMIN, AND, ATIME, CMIN, CPIO, CSIZE, CTIME, DEPTH, EXEC, F_GROUP,
  87         F_GROUPACL, F_USER, F_USERACL, FOLLOW, FSTYPE, INAME, INUM, IPATH,
  88         IREGEX, LINKS, LOCAL, LPAREN, LS, MAXDEPTH, MINDEPTH, MMIN, MOUNT,
  89         MTIME, NAME, NCPIO, NEWER, NOGRP, NOT, NOUSER, OK, OR, PATH, PERM,
  90         PRINT0, PRUNE, REGEX, RPAREN, SIZE, TYPE, VARARGS, XATTR, DELETE
  90         PRINT0, PRUNE, REGEX, RPAREN, SIZE, TYPE, VARARGS, XATTR
  91 };
_____unchanged_portion_omitted_

 105 /*
 106  * Except for pathnames, these are the only legal arguments
 107  */
 108 static struct Args commands[] =
 109 {
 110         "!",            NOT,            Op,
 111         "(",            LPAREN,         Unary,
 112         ")",            RPAREN,         Unary,
 113         "-a",           AND,            Op,
 114         "-acl",         ACL,            Unary,
 115         "-amin",        AMIN,           Num,
 116         "-and",         AND,            Op,
 117         "-atime",       ATIME,          Num,
 118         "-cmin",        CMIN,           Num,
 119         "-cpio",        CPIO,           Cpio,
 120         "-ctime",       CTIME,          Num,
 121         "-depth",       DEPTH,          Unary,
 122         "-delete",      DELETE,         Unary,
 123         "-exec",        EXEC,           Exec,
 124         "-follow",      FOLLOW,         Unary,
 125         "-fstype",      FSTYPE,         Str,
 126         "-group",       F_GROUP,        Num,
 127         "-groupacl",    F_GROUPACL,     Num,
 128         "-iname",       INAME,          Str,
 129         "-inum",        INUM,           Num,
 130         "-ipath",       IPATH,          Str,
 131         "-iregex",      IREGEX,         Str,
 132         "-links",       LINKS,          Num,
 133         "-local",       LOCAL,          Unary,
 134         "-ls",          LS,             Unary,
 135         "-maxdepth",    MAXDEPTH,       Num,
 136         "-mindepth",    MINDEPTH,       Num,
 137         "-mmin",        MMIN,           Num,
```

```
 138         "-mount",       MOUNT,          Unary,
 139         "-mtime",       MTIME,          Num,
 140         "-name",        NAME,           Str,
 141         "-ncpio",       NCPIO,          Cpio,
 142         "-newer",       NEWER,          Str,
 143         "-nogroup",     NOGRP,          Unary,
 144         "-not",         NOT,            Op,
 145         "-nouser",      NOUSER,         Unary,
 146         "-o",           OR,             Op,
 147         "-ok",          OK,             Exec,
 148         "-or",          OR,             Op,
 149         "-path",        PATH,           Str,
 150         "-perm",        PERM,           Num,
 151         "-print",       PRINT,          Unary,
 152         "-print0",      PRINT0,         Unary,
 153         "-prune",       PRUNE,          Unary,
 154         "-regex",       REGEX,          Str,
 155         "-size",        SIZE,           Num,
 156         "-type",        TYPE,           Num,
 157         "-user",        F_USER,         Num,
 158         "-useracl",     F_USERACL,      Num,
 159         "-xattr",       XATTR,          Unary,
 160         "-xdev",        MOUNT,          Unary,
 161         NULL,           0,              0
 162 };
```

*unchanged_portion_omitted_*

```
 205 static int              compile();
 206 static int              execute();
 207 static int              doexec(char *, char **, int *);
 208 static int              dodelete(char *, struct stat *, struct FTW *);
 209 static struct Args      *lookup();
 210 static int              ok();
 211 static void             usage(void)     __NORETURN;
 212 static struct Arglist   *varargs();
 213 static int              list();
 214 static char             *getgroup();
 215 static FILE             *cmdopen();
 216 static int              cmdclose();
 217 static char             *getshell();
 218 static void             init_remote_fs();
 219 static char             *getname();
 220 static int              readmode();
 221 static mode_t           getmode();
 222 static char             *gettail();


 225 static int walkflags = FTW_CHDIR|FTW_PHYS|FTW_ANYERR|FTW_NOLOOP;
 226 static struct Node      *topnode;
 227 static struct Node      *freenode;      /* next free node we may use later */
 228 static char             *cpio[] = { "cpio", "-o", 0 };
 229 static char             *ncpio[] = { "cpio", "-oc", 0 };
 230 static char             *cpiol[] = { "cpio", "-oL", 0 };
 231 static char             *ncpiol[] = { "cpio", "-ocL", 0 };
 232 static time_t           now;
 233 static FILE             *output;
 234 static char             *dummyarg = (char *)-1;
 235 static int              lastval;
 236 static int              varsize;
 237 static struct Arglist   *lastlist;
 238 static char             *cmdname;
 239 static char             *remote_fstypes[N_FSTYPES+1];
 240 static int              fstype_index = 0;
 241 static int              action_expression = 0;  /* -print, -exec, or -ok */
 242 static int              error = 0;
```



```
 243 static int              paren_cnt = 0;  /* keeps track of parentheses */
 244 static int              Eflag = 0;
 245 static int              hflag = 0;
 246 static int              lflag = 0;
 247 /* set when doexec()-invoked utility returns non-zero */
 248 static int              exec_exitcode = 0;
 249 static regex_t          *preg = NULL;
 250 static int              npreg = 0;
 251 static int              mindepth = -1, maxdepth = -1;
 252 extern char             **environ;

 254 int
 255 main(int argc, char **argv)
 256 {
 257         char *cp;
 258         int c;
 259         int paths;
 260         char *cwdpath;

 262         (void) setlocale(LC_ALL, "");
 263 #if !defined(TEXT_DOMAIN)       /* Should be defined by cc -D */
 264 #define TEXT_DOMAIN "SYS_TEST"  /* Use this only if it weren't */
 265 #endif
 266         (void) textdomain(TEXT_DOMAIN);

 268         cmdname = argv[0];
 269         if (time(&now) == (time_t)(-1)) {
 270                 (void) fprintf(stderr, gettext("%s: time() %s\n"),
 271                     cmdname, strerror(errno));
 272                 exit(1);
 273         }
 274         while ((c = getopt(argc, argv, "EHL")) != -1) {
 275                 switch (c) {
 276                 case 'E':
 277                         Eflag = 1;
 278                         break;
 279                 case 'H':
 280                         hflag = 1;
 281                         lflag = 0;
 282                         break;
 283                 case 'L':
 284                         hflag = 0;
 285                         lflag = 1;
 286                         break;
 287                 case '?':
 288                         usage();
 289                         break;
 290                 }
 291         }

 293         argc -= optind;
 294         argv += optind;

 296         if (argc < 1) {
 297                 (void) fprintf(stderr,
 298                     gettext("%s: insufficient number of arguments\n"), cmdname);
 299                 usage();
 300         }

 302         for (paths = 0; (cp = argv[paths]) != 0; ++paths) {
 303                 if (*cp == '-')
 304                         break;
 305                 else if ((*cp == '!' || *cp == '(') && *(cp+1) == 0)
 306                         break;
 307         }
```

```
309         if (paths == 0) /* no path-list */
310                 usage();

312         output = stdout;

314         /* lflag is the same as -follow */
315         if (lflag)
316                 walkflags &= ~FTW_PHYS;

318         /* allocate enough space for the compiler */
319         topnode = malloc((argc + 1) * sizeof (struct Node));
320         (void) memset(topnode, 0, (argc + 1) * sizeof (struct Node));

322         if (compile(argv + paths, topnode, &action_expression) == 0) {
323                 /* no expression, default to -print */
324                 (void) memcpy(topnode, &PRINT_NODE, sizeof (struct Node));
325         } else if (!action_expression) {
326                 /*
327                  * if no action expression, insert an LPAREN node above topnode,
328                  * with a PRINT node as its next node
329                  */
330                 struct Node *savenode;

332                 if (freenode == NULL) {
333                         (void) fprintf(stderr, gettext("%s: can't append -print"
334                             " implicitly; try explicit -print option\n"),
335                             cmdname);
336                         exit(1);
337                 }
338                 savenode = topnode;
339                 topnode = freenode++;
340                 (void) memcpy(topnode, &LPAREN_NODE, sizeof (struct Node));
341                 topnode->next = freenode;
342                 topnode->first.np = savenode;
343                 (void) memcpy(topnode->next, &PRINT_NODE, sizeof (struct Node));
344         }

346         while (paths--) {
347                 char *curpath;
348                 struct stat sb;

350                 curpath = *(argv++);

352                 /*
353                  * If -H is specified, it means we walk the first
354                  * level (pathname on command line) logically, following
355                  * symlinks, but lower levels are walked physically.
356                  * We use our own secret interface to nftw() to change
357                  * the from stat to lstat after the top level is walked.
358                  */
359                 if (hflag) {
360                         if (stat(curpath, &sb) < 0 && errno == ENOENT)
361                                 walkflags &= ~FTW_HOPTION;
362                         else
363                                 walkflags |= FTW_HOPTION;
364                 }

366                 /*
367                  * We need this check as nftw needs a CWD and we have no
368                  * way of returning back from that code with a meaningful
369                  * error related to this
370                  */
371                 if ((cwdpath = getcwd(NULL, PATH_MAX)) == NULL) {
372                         if ((errno == EACCES) && (walkflags & FTW_CHDIR)) {
373                                 /*
374                                  * A directory above cwd is inaccessible,
```



```
375                                  * so don't do chdir(2)s. Slower, but at least
376                                  * it works.
377                                  */
378                                 walkflags &= ~FTW_CHDIR;
379                                 free(cwdpath);
380                         } else {
381                                 (void) fprintf(stderr,
382                                     gettext("%s : cannot get the current "
383                                     "working directory\n"), cmdname);
384                                 exit(1);
385                         }
386                 } else
387                         free(cwdpath);


390                 if (nftw(curpath, execute, 1000, walkflags)) {
391                         (void) fprintf(stderr,
392                             gettext("%s: cannot open %s: %s\n"),
393                             cmdname, curpath, strerror(errno));
394                         error = 1;
395                 }

397         }

399         /* execute any remaining variable length lists */
400         while (lastlist) {
401                 if (lastlist->end != lastlist->nextstr) {
402                         *lastlist->nextvar = 0;
403                         (void) doexec((char *)0, lastlist->arglist,
404                             &exec_exitcode);
405                 }
406                 lastlist = lastlist->next;
407         }
408         if (output != stdout)
409                 return (cmdclose(output));
410         return ((exec_exitcode != 0) ? exec_exitcode : error);
411 }

413 /*
414  * compile the arguments
415  */

417 static int
418 compile(argv, np, actionp)
419 char **argv;
420 struct Node *np;
421 int *actionp;
422 {
423         char *b;
424         char **av;
425         struct Node *oldnp = topnode;
426         struct Args *argp;
427         char **com;
428         int i;
429         enum Command wasop = PRINT;

431         if (init_yes() < 0) {
432                 (void) fprintf(stderr, gettext(ERR_MSG_INIT_YES),
433                     strerror(errno));
434                 exit(1);
435         }

437         for (av = argv; *av && (argp = lookup(*av)); av++) {
438                 np->next = 0;
439                 np->action = argp->action;
440                 np->type = argp->type;
```

```
441                 np->second.i = 0;
442                 if (argp->type == Op) {
443                         if (wasop == NOT || (wasop && np->action != NOT)) {
444                                 (void) fprintf(stderr,
445                                 gettext("%s: operand follows operand\n"),
446                                                 cmdname);
447                                 exit(1);
448                         }
449                         if (np->action != NOT && oldnp == 0)
450                                 goto err;
451                         wasop = argp->action;
452                 } else {
453                         wasop = PRINT;
454                         if (argp->type != Unary) {
455                                 if (!(b = *++av)) {
456                                         (void) fprintf(stderr,
457                                         gettext("%s: incomplete statement\n"),
458                                                         cmdname);
459                                         exit(1);
460                                 }
461                                 if (argp->type == Num) {
462                                         if (((argp->action == MAXDEPTH) ||
463                                             (argp->action == MINDEPTH)) &&
464                                             ((int)strtol(b, (char **)NULL,
465                                             10) < 0))
466                                                 errx(1,
467                                         gettext("%s: value must be positive"),
468                                                     (argp->action == MAXDEPTH) ?
469                                                     "maxdepth" : "mindepth");
470                                         if ((argp->action != PERM) ||
471                                             (*b != '+')) {
472                                                 if (*b == '+' || *b == '-') {
473                                                         np->second.i = *b;
474                                                         b++;
475                                                 }
476                                         }
477                                 }
478                         }
479                 }
480                 switch (argp->action) {
481                 case AND:
482                         break;
483                 case NOT:
484                         break;
485                 case OR:
486                         np->first.np = topnode;
487                         topnode = np;
488                         oldnp->next = 0;
489                         break;

491                 case LPAREN: {
492                         struct Node *save = topnode;
493                         topnode = np+1;
494                         paren_cnt++;
495                         i = compile(++av, topnode, actionp);
496                         np->first.np = topnode;
497                         topnode = save;
498                         av += i;
499                         oldnp = np;
500                         np += i + 1;
501                         oldnp->next = np;
502                         continue;
503                 }

505                 case RPAREN:
506                         if (paren_cnt <= 0) {
```



```
507                                 (void) fprintf(stderr,
508                                     gettext("%s: unmatched ')'\n"),
509                                     cmdname);
510                                 exit(1);
511                         }
512                         paren_cnt--;
513                         if (oldnp == 0)
514                                 goto err;
515                         if (oldnp->type == Op) {
516                                 (void) fprintf(stderr,
517                                     gettext("%s: cannot immediately"
518                                     " follow an operand with ')'\n"),
519                                     cmdname);
520                                 exit(1);
521                         }
522                         oldnp->next = 0;
523                         return (av-argv);

525                 case FOLLOW:
526                         walkflags &= ~FTW_PHYS;
527                         break;
528                 case MOUNT:
529                         walkflags |= FTW_MOUNT;
530                         break;
531                 case DEPTH:
532                         walkflags |= FTW_DEPTH;
533                         break;
534                 case DELETE:
535                         walkflags |= (FTW_DEPTH | FTW_PHYS);
536                         walkflags &= ~FTW_CHDIR;
537                         (*actionp)++;
538                         break;

540                 case LOCAL:
541                         np->first.l = 0L;
542                         np->first.ll = 0LL;
543                         np->second.i = '+';
544                         /*
545                          * Make it compatible to df -l for
546                          * future enhancement. So, anything
547                          * that is not remote, then it is
548                          * local.
549                          */
550                         init_remote_fs();
551                         break;

553                 case SIZE:
554                         if (b[strlen(b)-1] == 'c')
555                                 np->action = CSIZE;
556                         /*FALLTHROUGH*/
557                 case INUM:
558                         np->first.ll = atoll(b);
559                         break;

561                 case CMIN:
562                 case CTIME:
563                 case MMIN:
564                 case MTIME:
565                 case AMIN:
566                 case ATIME:
567                 case LINKS:
568                         np->first.l = atol(b);
569                         break;

571                 case F_USER:
572                 case F_GROUP:
```

```
573                 case F_USERACL:
574                 case F_GROUPACL: {
575                         struct  passwd  *pw;
576                         struct  group *gr;
577                         long value;
578                         char *q;

580                         value = -1;
581                         if (argp->action == F_USER ||
582                             argp->action == F_USERACL) {
583                                 if ((pw = getpwnam(b)) != 0)
584                                         value = (long)pw->pw_uid;
585                         } else {
586                                 if ((gr = getgrnam(b)) != 0)
587                                         value = (long)gr->gr_gid;
588                         }
589                         if (value == -1) {
590                                 errno = 0;
591                                 value = strtol(b, &q, 10);
592                                 if (errno != 0 || q == b || *q != '\0') {
593                                         (void) fprintf(stderr, gettext(
594                                             "%s: cannot find %s name\n"),
595                                                 cmdname, *av);
596                                         exit(1);
597                                 }
598                         }
599                         np->first.l = value;
600                         break;
601                 }

603                 case EXEC:
604                 case OK:
605                         walkflags &= ~FTW_CHDIR;
606                         np->first.ap = av;
607                         (*actionp)++;
608                         for (;;) {
609                                 if ((b = *av) == 0) {
610                                         (void) fprintf(stderr,
611                                         gettext("%s: incomplete statement\n"),
612                                                 cmdname);
613                                         exit(1);
614                                 }
615                                 if (strcmp(b, ";") == 0) {
616                                         *av = 0;
617                                         break;
618                                 } else if (strcmp(b, "{}") == 0)
619                                         *av = dummyarg;
620                                 else if (strcmp(b, "+") == 0 &&
621                                         av[-1] == dummyarg &&
622                                         np->action == EXEC) {
623                                         av[-1] = 0;
624                                         np->first.vp = varargs(np->first.ap);
625                                         np->action = VARARGS;
626                                         break;
627                                 }
628                                 av++;
629                         }
630                         break;

632                 case NAME:
633                 case INAME:
634                 case PATH:
635                 case IPATH:
636                         np->first.cp = b;
637                         break;
638                 case REGEX:
```



```
639                 case IREGEX: {
640                         int error;
641                         size_t errlen;
642                         char *errmsg;

644                         if ((preg = realloc(preg, (npreg + 1) *
645                             sizeof (regex_t))) == NULL)
646                                 err(1, "realloc");
647                         if ((error = regcomp(&preg[npreg], b,
648                             ((np->action == IREGEX) ? REG_ICASE : 0) |
649                             ((Eflag) ? REG_EXTENDED : 0))) != 0) {
650                                 errlen = regerror(error, &preg[npreg], NULL, 0);
651                                 if ((errmsg = malloc(errlen)) == NULL)
652                                         err(1, "malloc");
653                                 (void) regerror(error, &preg[npreg], errmsg,
654                                     errlen);
655                                 errx(1, gettext("RE error: %s"), errmsg);
656                         }
657                         npreg++;
658                         break;
659                 }
660                 case PERM:
661                         if (*b == '-')
662                                 ++b;

664                         if (readmode(b) != NULL) {
665                                 (void) fprintf(stderr, gettext(
666                                     "find: -perm: Bad permission string\n"));
667                                 usage();
668                         }
669                         np->first.l = (long)getmode((mode_t)0);
670                         break;
671                 case TYPE:
672                         i = *b;
673                         np->first.l =
674                             i == 'd' ? S_IFDIR :
675                             i == 'b' ? S_IFBLK :
676                             i == 'c' ? S_IFCHR :
677 #ifdef S_IFIFO
678                             i == 'p' ? S_IFIFO :
679 #endif
680                             i == 'f' ? S_IFREG :
681 #ifdef S_IFLNK
682                             i == 'l' ? S_IFLNK :
683 #endif
684 #ifdef S_IFSOCK
685                             i == 's' ? S_IFSOCK :
686 #endif
687 #ifdef S_IFDOOR
688                             i == 'D' ? S_IFDOOR :
689 #endif
690                             0;
691                         break;

693                 case CPIO:
694                         if (walkflags & FTW_PHYS)
695                                 com = cpio;
696                         else
697                                 com = cpiol;
698                         goto common;

700                 case NCPIO: {
701                         FILE *fd;

703                         if (walkflags & FTW_PHYS)
704                                 com = ncpio;
```

```
705                         else
706                                 com = ncpiol;
707                 common:
708                         /* set up cpio */
709                         if ((fd = fopen(b, "w")) == NULL) {
710                                 (void) fprintf(stderr,
711                                         gettext("%s: cannot create %s\n"),
712                                         cmdname, b);
713                                 exit(1);
714                         }

716                         np->first.l = (long)cmdopen("cpio", com, "w", fd);
717                         (void) fclose(fd);
718                         walkflags |= FTW_DEPTH;
719                         np->action = CPIO;
720                 }
721                         /*FALLTHROUGH*/
722                 case PRINT:
723                 case PRINT0:
724                         (*actionp)++;
725                         break;

727                 case NEWER: {
728                         struct stat statb;
729                         if (stat(b, &statb) < 0) {
730                                 (void) fprintf(stderr,
731                                         gettext("%s: cannot access %s\n"),
732                                         cmdname, b);
733                                 exit(1);
734                         }
735                         np->first.l = statb.st_mtime;
736                         np->second.i = '+';
737                         break;
738                 }

740                 case PRUNE:
741                 case NOUSER:
742                 case NOGRP:
743                         break;
744                 case FSTYPE:
745                         np->first.cp = b;
746                         break;
747                 case LS:
748                         (*actionp)++;
749                         break;
750                 case XATTR:
751                         break;
752                 case ACL:
753                         break;
754                 case MAXDEPTH:
755                         maxdepth = (int)strtol(b, (char **)NULL, 10);
756                         break;
757                 case MINDEPTH:
758                         mindepth = (int)strtol(b, (char **)NULL, 10);
759                         break;
760                 }

762                 oldnp = np++;
763                 oldnp->next = np;
764         }

766         if ((*av) || (wasop))
767                 goto err;

769         if (paren_cnt != 0) {
770                 (void) fprintf(stderr, gettext("%s: unmatched '('\n"),
```



```
771                         cmdname);
772                 exit(1);
773         }

775         /* just before returning, save next free node from the list */
776         freenode = oldnp->next;
777         oldnp->next = 0;
778         return (av-argv);
779 err:
780         if (*av)
781                 (void) fprintf(stderr,
782                     gettext("%s: bad option %s\n"), cmdname, *av);
783         else
784                 (void) fprintf(stderr, gettext("%s: bad option\n"), cmdname);
785         usage();
786         /*NOTREACHED*/
787 }
```

*unchanged_portion_omitted_*

```
801 /*
802  * This is the function that gets executed at each node
803  */

805 static int
806 execute(name, statb, type, state)
807 char *name;
808 struct stat *statb;
809 int type;
810 struct FTW *state;
811 {
812         struct Node *np = topnode;
813         int val;
814         time_t t;
815         long l;
816         long long ll;
817         int not = 1;
818         char *filename;
819         int cnpreg = 0;

821         if (type == FTW_NS) {
822                 (void) fprintf(stderr, gettext("%s: stat() error %s: %s\n"),
823                         cmdname, name, strerror(errno));
824                 error = 1;
825                 return (0);
826         } else if (type == FTW_DNR) {
827                 (void) fprintf(stderr, gettext("%s: cannot read dir %s: %s\n"),
828                         cmdname, name, strerror(errno));
829                 error = 1;
830         } else if (type == FTW_SLN && lflag == 1) {
831                 (void) fprintf(stderr,
832                         gettext("%s: cannot follow symbolic link %s: %s\n"),
833                         cmdname, name, strerror(errno));
834                 error = 1;
835         } else if (type == FTW_DL) {
836                 (void) fprintf(stderr, gettext("%s: cycle detected for %s\n"),
837                         cmdname, name);
838                 error = 1;
839                 return (0);
840         }

842         if ((maxdepth != -1 && state->level > maxdepth) ||
843             (mindepth != -1 && state->level < mindepth))
844                 return (0);

846         while (np) {
847                 switch (np->action) {
```

```
848                 case NOT:
849                         not = !not;
850                         np = np->next;
851                         continue;
852
853                 case AND:
854                         np = np->next;
855                         continue;
856
857                 case OR:
858                         if (np->first.np == np) {
859                                 /*
860                                  * handle naked OR (no term on left hand side)
861                                  */
862                                 (void) fprintf(stderr,
863                                     gettext("%s: invalid -o construction\n"),
864                                     cmdname);
865                                 exit(2);
866                         }
867                         /* FALLTHROUGH */
868                 case LPAREN: {
869                         struct Node *save = topnode;
870                         topnode = np->first.np;
871                         (void) execute(name, statb, type, state);
872                         val = lastval;
873                         topnode = save;
874                         if (np->action == OR) {
875                                 if (val)
876                                         return (0);
877                                 val = 1;
878                         }
879                         break;
880                 }
881
882                 case LOCAL: {
883                         int     nremfs;
884                         val = 1;
885                         /*
886                          * If file system type matches the remote
887                          * file system type, then it is not local.
888                          */
889                         for (nremfs = 0; nremfs < fstype_index; nremfs++) {
890                                 if (strcmp(remote_fstypes[nremfs],
891                                                 statb->st_fstype) == 0) {
892                                         val = 0;
893                                         break;
894                                 }
895                         }
896                         break;
897                 }
898
899                 case TYPE:
900                         l = (long)statb->st_mode&S_IFMT;
901                         goto num;
902
903                 case PERM:
904                         l = (long)statb->st_mode&07777;
905                         if (np->second.i == '-')
906                                 val = ((l&np->first.l) == np->first.l);
907                         else
908                                 val = (l == np->first.l);
909                         break;
910
911                 case INUM:
912                         ll = (long long)statb->st_ino;
913                         goto llnum;
```



```
914                 case NEWER:
915                         l = statb->st_mtime;
916                         goto num;
917                 case ATIME:
918                         t = statb->st_atime;
919                         goto days;
920                 case CTIME:
921                         t = statb->st_ctime;
922                         goto days;
923                 case MTIME:
924                         t = statb->st_mtime;
925                 days:
926                         l = (now-t)/A_DAY;
927                         goto num;
928                 case MMIN:
929                         t = statb->st_mtime;
930                         goto mins;
931                 case AMIN:
932                         t = statb->st_atime;
933                         goto mins;
934                 case CMIN:
935                         t = statb->st_ctime;
936                         goto mins;
937                 mins:
938                         l = (now-t)/A_MIN;
939                         goto num;
940                 case CSIZE:
941                         ll = (long long)statb->st_size;
942                         goto llnum;
943                 case SIZE:
944                         ll = (long long)round(statb->st_size, BLKSIZ)/BLKSIZ;
945                         goto llnum;
946                 case F_USER:
947                         l = (long)statb->st_uid;
948                         goto num;
949                 case F_GROUP:
950                         l = (long)statb->st_gid;
951                         goto num;
952                 case LINKS:
953                         l = (long)statb->st_nlink;
954                         goto num;
955                 llnum:
956                         if (np->second.i == '+')
957                                 val = (ll > np->first.ll);
958                         else if (np->second.i == '-')
959                                 val = (ll < np->first.ll);
960                         else
961                                 val = (ll == np->first.ll);
962                         break;
963                 num:
964                         if (np->second.i == '+')
965                                 val = (l > np->first.l);
966                         else if (np->second.i == '-')
967                                 val = (l < np->first.l);
968                         else
969                                 val = (l == np->first.l);
970                         break;
971                 case OK:
972                         val = ok(name, np->first.ap);
973                         break;
974                 case EXEC:
975                         val = doexec(name, np->first.ap, NULL);
976                         break;
977                 case DELETE:
978                         val = dodelete(name, statb, state);
979                         break;
```

```
 981                 case VARARGS: {
 982                         struct Arglist *ap = np->first.vp;
 983                         char *cp;
 984                         cp = ap->nextstr - (strlen(name)+1);
 985                         if (cp >= (char *)(ap->nextvar+3)) {
 986                                 /* there is room just copy the name */
 987                                 val = 1;
 988                                 (void) strcpy(cp, name);
 989                                 *ap->nextvar++ = cp;
 990                                 ap->nextstr = cp;
 991                         } else {
 992                                 /* no more room, exec command */
 993                                 *ap->nextvar++ = name;
 994                                 *ap->nextvar = 0;
 995                                 val = 1;
 996                                 (void) doexec((char *)0, ap->arglist,
 997                                     &exec_exitcode);
 998                                 ap->nextstr = ap->end;
 999                                 ap->nextvar = ap->firstvar;
1000                         }
1001                         break;
1002                 }

1004                 case DEPTH:
1005                 case MOUNT:
1006                 case FOLLOW:
1007                         val = 1;
1008                         break;

1010                 case NAME:
1011                 case INAME:
1012                 case PATH:
1013                 case IPATH: {
1014                         char *path;
1015                         int fnmflags = 0;

1017                         if (np->action == INAME || np->action == IPATH)
1018                                 fnmflags = FNM_IGNORECASE;

1020                         /*
1021                          * basename(3c) may modify name, so
1022                          * we need to pass another string
1023                          */
1024                         if ((path = strdup(name)) == NULL) {
1025                                 (void) fprintf(stderr,
1026                                     gettext("%s: cannot strdup() %s: %s\n"),
1027                                     cmdname, name, strerror(errno));
1028                                 exit(2);
1029                         }
1030                         /*
1031                          * XPG4 find should not treat a leading '.' in a
1032                          * filename specially for pattern matching.
1033                          * /usr/bin/find  will not pattern match a leading
1034                          * '.' in a filename, unless '.' is explicitly
1035                          * specified.
1036                          */
1037 #ifndef XPG4
1038                         fnmflags |= FNM_PERIOD;
1039 #endif

1041                         val = !fnmatch(np->first.cp,
1042                             (np->action == NAME || np->action == INAME)
1043                                 ? basename(path) : path, fnmflags);
1044                         free(path);
1045                         break;
```



```
1046                 }

1048                 case PRUNE:
1049                         if (type == FTW_D)
1050                                 state->quit = FTW_PRUNE;
1051                         val = 1;
1052                         break;
1053                 case NOUSER:
1054                         val = ((getpwuid(statb->st_uid)) == 0);
1055                         break;
1056                 case NOGRP:
1057                         val = ((getgrgid(statb->st_gid)) == 0);
1058                         break;
1059                 case FSTYPE:
1060                         val = (strcmp(np->first.cp, statb->st_fstype) == 0);
1061                         break;
1062                 case CPIO:
1063                         output = (FILE *)np->first.l;
1064                         (void) fprintf(output, "%s\n", name);
1065                         val = 1;
1066                         break;
1067                 case PRINT:
1068                 case PRINT0:
1069                         (void) fprintf(stdout, "%s%c", name,
1070                             (np->action == PRINT) ? '\n' : '\0');
1071                         val = 1;
1072                         break;
1073                 case LS:
1074                         (void) list(name, statb);
1075                         val = 1;
1076                         break;
1077                 case XATTR:
1078                         filename = (walkflags & FTW_CHDIR) ?
1079                                 gettail(name) : name;
1080                         val = (pathconf(filename, _PC_XATTR_EXISTS) == 1);
1081                         break;
1082                 case ACL:
1083                         /*
1084                          * Need to get the tail of the file name, since we have
1085                          * already chdir()ed into the directory (performed in
1086                          * nftw()) of the file
1087                          */
1088                         filename = (walkflags & FTW_CHDIR) ?
1089                                 gettail(name) : name;
1090                         val = acl_trivial(filename);
1091                         break;
1092                 case F_USERACL:
1093                 case F_GROUPACL: {
1094                         int i;
1095                         acl_t *acl;
1096                         void *acl_entry;
1097                         aclent_t *p1;
1098                         ace_t *p2;

1100                         filename = (walkflags & FTW_CHDIR) ?
1101                             gettail(name) : name;
1102                         val = 0;
1103                         if (acl_get(filename, 0, &acl) != 0)
1104                                 break;
1105                         for (i = 0, acl_entry = acl->acl_aclp;
1106                             i != acl->acl_cnt; i++) {
1107                                 if (acl->acl_type == ACLENT_T) {
1108                                         p1 = (aclent_t *)acl_entry;
1109                                         if (p1->a_id == np->first.l) {
1110                                                 val = 1;
1111                                                 acl_free(acl);
```

```
1112                                            break;
1113                                    }
1114                            } else {
1115                                    p2 = (ace_t *)acl_entry;
1116                                    if (p2->a_who == np->first.l) {
1117                                            val = 1;
1118                                            acl_free(acl);
1119                                            break;
1120                                    }
1121                            }
1122                            acl_entry = ((char *)acl_entry +
1123                                acl->acl_entry_size);
1124                    }
1125                    acl_free(acl);
1126                    break;
1127            }
1128            case IREGEX:
1129            case REGEX: {
1130                    regmatch_t pmatch;

1132                    val = 0;
1133                    if (regexec(&preg[cnpreg], name, 1, &pmatch, NULL) == 0)
1134                            val = ((pmatch.rm_so == 0) &&
1135                                (pmatch.rm_eo == strlen(name)));
1136                    cnpreg++;
1137                    break;
1138            }
1139            case MAXDEPTH:
1140                    if (state->level == maxdepth && type == FTW_D)
1141                            state->quit = FTW_PRUNE;
1142                    /* FALLTHROUGH */
1143            case MINDEPTH:
1144                    val = 1;
1145                    break;
1146            }
1147            /*
1148             * evaluate 'val' and 'not' (exclusive-or)
1149             * if no inversion (not == 1), return only when val == 0
1150             * (primary not true). Otherwise, invert the primary
1151             * and return when the primary is true.
1152             * 'Lastval' saves the last result (fail or pass) when
1153             * returning back to the calling routine.
1154             */
1155            if (val^not) {
1156                    lastval = 0;
1157                    return (0);
1158            }
1159            lastval = 1;
1160            not = 1;
1161            np = np->next;
1162        }
1163        return (0);
1164 }
```

*unchanged_portion_omitted_*

```
1331 static int
1332 dodelete(char *name, struct stat *statb, struct FTW *state)
1333 {
1334         char *fn;
1335         int rc = 0;

1337         /* restrict symlinks */
1338         if ((walkflags & FTW_PHYS) == 0) {
1339                 (void) fprintf(stderr,
1340                     gettext("-delete is not allowed when symlinks are "
1341                     "followed.\n"));
```



```
1342                 return (1);
1343         }

1345         fn = name + state->base;
1346         if (strcmp(fn, ".") == 0) {
1347                 /* nothing to do */
1348                 return (1);
1349         }

1351         if (strchr(fn, '/') != NULL) {
1352                 (void) fprintf(stderr,
1353                     gettext("-delete with relative path is unsafe."));
1354                 return (1);
1355         }

1357         if (S_ISDIR(statb->st_mode)) {
1358                 /* delete directory */
1359                 rc = rmdir(name);
1360         } else {
1361                 /* delete file */
1362                 rc = unlink(name);
1363         }

1365         if (rc < 0) {
1366                 /* operation failed */
1367                 (void) fprintf(stderr, gettext("delete failed %s: %s\n"),
1368                     name, strerror(errno));
1369                 return (1);
1370         }

1372         return (1);
1373 }
```

```
1375 /*
1376  *  Table lookup routine
1377  */
1378 static struct Args *
1379 lookup(word)
1380 char *word;
1381 {
1382         struct Args *argp = commands;
1383         int second;
1384         if (word == 0 || *word == 0)
1385                 return (0);
1386         second = word[1];
1387         while (*argp->name) {
1388                 if (second == argp->name[1] && strcmp(word, argp->name) == 0)
1389                         return (argp);
1390                 argp++;
1391         }
1392         return (0);
1393 }
```

*unchanged_portion_omitted_*