```
**********************************************************
    1920 Sat May 11 15:20:42 2013
new/usr/src/cmd/awk/Makefile
3731 Update nawk to version 20121220
**********************************************************
```

```
#
# CDDL HEADER START
#
# The contents of this file are subject to the terms of the
# Common Development and Distribution License, Version 1.0 only
# (the "License").  You may not use this file except in compliance
# with the License.
#
# You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
# or http://www.opensolaris.org/os/licensing.
# See the License for the specific language governing permissions
# and limitations under the License.
#
# When distributing Covered Code, include this CDDL HEADER in each
# file and include the License file at usr/src/OPENSOLARIS.LICENSE.
# If applicable, add the following below this CDDL HEADER, with the
# fields enclosed by brackets "[]" replaced with your own identifying
# information: Portions Copyright [yyyy] [name of copyright owner]
#
# CDDL HEADER END
#
#
#
# Copyright 2005 Sun Microsystems, Inc.  All rights reserved.
# Use is subject to license terms.
#

# NOTE: awk is oawk.

PROG= nawk

OBJ1= b.o lex.o lib.o main.o parse.o proctab.o run.o tran.o
OBJ2= awk.g.o
OBJ1= b.o lib.o main.o parse.o proctab.o run.o tran.o
OBJ2= awk.g.o awk.lx.o
OBJS= $(OBJ2) $(OBJ1)
SRCS= $(OBJ1:%.o=%.c)

include ../Makefile.cmd

CERRWARN += -_gcc=-Wno-implicit-function-declaration
CERRWARN += -_gcc=-Wno-unused-label
CERRWARN += -_gcc=-Wno-parentheses
CERRWARN += -_gcc=-Wno-unused-variable
CERRWARN += -_gcc=-Wno-uninitialized

#
# Message catalog
#
POFILES= $(OBJS:%.o=%.po)
POFILE= awk.po
XGETFLAGS += -a -x awk.xcl
#

CPPFLAGS += -D_FILE_OFFSET_BITS=64
YFLAGS += -d
LDLIBS += -lm
LINTFLAGS += -u
CLEANFILES= maketab proctab.c awk.g.c awk.lx.c y.tab.h

.KEEP_STATE:
```



```
all: $(PROG)

$(PROG): $(OBJS)
        $(LINK.c) $(OBJS) -o $@ $(LDLIBS)
        $(POST_PROCESS)

#
# message catalog
#

$(POFILE):  y.tab.h $(POFILES)
        $(RM) $@
        cat $(POFILES) > $@
#

proctab.c: maketab
        rm -f $@; ./maketab > $@

maketab: maketab.c
        $(NATIVECC) -O maketab.c -o $@ $(LDLIBS)

install: all $(ROOTPROG) $(ROOTLINK)

clean:
        $(RM) $(OBJS) $(CLEANFILES)

lint: awk.g.c lint_SRCS
        $(LINT.c) awk.g.c $(LDLIBS)
        $(LINT.c) maketab.c $(LDLIBS)
#endif /* ! codereview */

awk.g.c + y.tab.h: awk.g.y

awk.g.o: awk.g.c

awk.lx.c: awk.lx.l

proctab.o: proctab.c
        $(COMPILE.c) proctab.c
        $(POST_PROCESS_O)

include ../Makefile.targ
```

```
**********************************************************
   14927 Sat May 11 15:20:42 2013
new/usr/src/cmd/awk/awk.g.y
3731 Update nawk to version 20121220
**********************************************************
```

```
%{
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License, Version 1.0 only
 * (the "License").  You may not use this file except in compliance
 * with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
%}
/*
 * Copyright 2005 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * Copyright (C) Lucent Technologies 1997
 * All Rights Reserved
 *
 * Permission to use, copy, modify, and distribute this software and
 * its documentation for any purpose and without fee is hereby
 * granted, provided that the above copyright notice appear in all
 * copies and that both that the copyright notice and this
 * permission notice and warranty disclaimer appear in supporting
 * documentation, and that the name Lucent Technologies or any of
 * its entities not be used in advertising or publicity pertaining
 * to distribution of the software without specific, written prior
 * permission.
 *
 * LUCENT DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
 * INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS.
 * IN NO EVENT SHALL LUCENT OR ANY OF ITS ENTITIES BE LIABLE FOR ANY
 * SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
 * WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER
 * IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION,
 * ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF
 * THIS SOFTWARE.
 */
/*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
/*        All Rights Reserved   */

%{
#ident  "%Z%%M% %I%     %E% SMI"        /* SVr4.0 2.10  */
%}

%{
#include "awk.h"
int yywrap(void) { return(1); }
```



```
Node    *beginloc = 0;
Node    *endloc = 0;
#ifndef DEBUG
#       define  PUTS(x)
#endif
Node    *beginloc = 0, *endloc = 0;
int     infunc  = 0;    /* = 1 if in arglist or body of func */
int     inloop = 0;     /* = 1 if in while, for, do */
uchar   *curfname = 0;  /* current function name */
uchar   *curfname = 0;
Node    *arglist = 0;   /* list of args for current function */
static void     setfname(Cell *);
static int      constnode(Node *);
static uchar    *strnode(Node *);
static Node     *notnull();
%}

%union {
        Node    *p;
        Cell    *cp;
        int     i;
        uchar   *s;
}

%token  <i>     FIRSTTOKEN      /* must be first */
%token  <p>     PROGRAM PASTAT PASTAT2 XBEGIN XEND
%token  <i>     NL ',' '{' '(' '|' ';' '/' ')' '}' '[' ']'
%token  <i>     ARRAY
%token  <i>     MATCH NOTMATCH MATCHOP
%token  <i>     FINAL DOT ALL CCL NCCL CHAR OR STAR QUEST PLUS EMPTYRE
%token  <i>     FINAL DOT ALL CCL NCCL CHAR OR STAR QUEST PLUS
%token  <i>     AND BOR APPEND EQ GE GT LE LT NE IN
%token  <i>     ARG BLTIN BREAK CLOSE CONTINUE DELETE DO EXIT FOR FUNC
%token  <i>     SUB GSUB IF INDEX LSUBSTR MATCHFCN NEXT NEXTFILE
%token  <i>     SUB GSUB IF INDEX LSUBSTR MATCHFCN NEXT
%token  <i>     ADD MINUS MULT DIVIDE MOD
%token  <i>     ASSIGN ASGNOP ADDEQ SUBEQ MULTEQ DIVEQ MODEQ POWEQ
%token  <i>     PRINT PRINTF SPRINTF
%token  <p>     ELSE INTEST CONDEXPR
%token  <i>     POSTINCR PREINCR POSTDECR PREDECR
%token  <cp>    VAR IVAR VARNF CALL NUMBER STRING
%token  <cp>    VAR IVAR VARNF CALL NUMBER STRING FIELD
%token  <s>     REGEXPR

%type   <p>     pas pattern ppattern plist pplist patlist prarg term re
%type   <p>     pas pattern ppattern plist pplist patlist prarg term
%type   <p>     pa_pat pa_stat pa_stats
%type   <s>     reg_expr
%type   <p>     simple_stmt opt_simple_stmt stmt stmtlist
%type   <p>     var varname funcname varlist
%type   <p>     for if else while
%type   <i>     do st
%type   <i>     pst opt_pst lbrace rbrace rparen comma nl opt_nl and bor
%type   <p>     for if while
%type   <i>     pst opt_pst lbrace rparen comma nl opt_nl and bor
%type   <i>     subop print

%right  ASGNOP
%right  '?'
%right  ':'
%left   BOR
%left   AND
%left   GETLINE
%nonassoc APPEND EQ GE GT LE LT NE MATCHOP IN '|'
%left   ARG BLTIN BREAK CALL CLOSE CONTINUE DELETE DO EXIT FOR FUNC
```

```
 92 %left   ARG BLTIN BREAK CALL CLOSE CONTINUE DELETE DO EXIT FOR FIELD FUNC
114 %left   GSUB IF INDEX LSUBSTR MATCHFCN NEXT NUMBER
115 %left   PRINT PRINTF RETURN SPLIT SPRINTF STRING SUB SUBSTR
116 %left   REGEXPR VAR VARNF IVAR WHILE '('
117 %left   CAT
118 %left   '+' '-'
119 %left   '*' '/' '%'
120 %left   NOT UMINUS
121 %right  POWER
122 %right  DECR INCR
123 %left   INDIRECT
124 %token  LASTTOKEN       /* must be last */

126 %%

128 program:
129           pas   { if (errorflag==0)
130                     winner = (Node *)stat3(PROGRAM, beginloc, $1, endloc); }
131         | error { yyclearin; bracecheck(); SYNTAX("bailing out"); }
110         | error { yyclearin; bracecheck(); ERROR "bailing out" SYNTAX; }
132         ;

134 and:
135           AND | and NL
136         ;

138 bor:
139           BOR | bor NL
140         ;

142 comma:
143           ',' | comma NL
144         ;

146 do:
147           DO | do NL
148         ;

150 else:
151           ELSE | else NL
152         ;

154 for:
155           FOR '(' opt_simple_stmt ';' opt_nl pattern ';' opt_nl opt_simple_stmt
156                 { --inloop; $$ = stat4(FOR, $3, notnull($6), $9, $12); }
157         | FOR '(' opt_simple_stmt ';'  ';' opt_nl opt_simple_stmt rparen {inloop
158                 { --inloop; $$ = stat4(FOR, $3, NIL, $7, $10); }
159         | FOR '(' varname IN varname rparen {inloop++;} stmt
160                 { --inloop; $$ = stat3(IN, $3, makearr($5), $8); }
134         | FOR '(' opt_simple_stmt ';' pattern ';' opt_simple_stmt rparen stmt
135                 { $$ = stat4(FOR, $3, notnull($5), $7, $9); }
136         | FOR '(' opt_simple_stmt ';'  ';' opt_simple_stmt rparen stmt
137                 { $$ = stat4(FOR, $3, NIL, $6, $8); }
138         | FOR '(' varname IN varname rparen stmt
139                 { $$ = stat3(IN, $3, makearr($5), $7); }
161         ;

163 funcname:
164           VAR   { setfname($1); }
165         | CALL  { setfname($1); }
166         ;

168 if:
169           IF '(' pattern rparen         { $$ = notnull($3); }
170         ;
```



```
172 lbrace:
173           '{' | lbrace NL
174         ;

176 nl:
177           NL | nl NL
178         ;

180 opt_nl:
181           /* empty */   { $$ = 0; }
182         | nl
183         ;

185 opt_pst:
186           /* empty */   { $$ = 0; }
187         | pst
188         ;


191 opt_simple_stmt:
192           /* empty */                   { $$ = 0; }
193         | simple_stmt
194         ;

196 pas:
197           opt_pst                       { $$ = 0; }
198         | opt_pst pa_stats opt_pst      { $$ = $2; }
199         ;

201 pa_pat:
202           pattern       { $$ = notnull($1); }
203         ;

205 pa_stat:
206           pa_pat                        { $$ = stat2(PASTAT, $1, stat2(PRINT, re
207         | pa_pat lbrace stmtlist '}'    { $$ = stat2(PASTAT, $1, $3); }
208         | pa_pat ',' opt_nl pa_pat              { $$ = pa2stat($1, $4, stat2(PRI
209         | pa_pat ',' opt_nl pa_pat lbrace stmtlist '}'  { $$ = pa2stat($1, $4, $
187         | pa_pat ',' pa_pat             { $$ = pa2stat($1, $3, stat2(PRINT, rect
188         | pa_pat ',' pa_pat lbrace stmtlist '}' { $$ = pa2stat($1, $3, $5); }
210         | lbrace stmtlist '}'           { $$ = stat2(PASTAT, NIL, $2); }
211         | XBEGIN lbrace stmtlist '}'
212                 { beginloc = linkum(beginloc, $3); $$ = 0; }
213         | XEND lbrace stmtlist '}'
214                 { endloc = linkum(endloc, $3); $$ = 0; }
215         | FUNC funcname '(' varlist rparen {infunc++;} lbrace stmtlist '}'
216                 { infunc--; curfname=0; defn((Cell *)$2, $4, $8); $$ = 0; }
217         ;

219 pa_stats:
220           pa_stat
221         | pa_stats opt_pst pa_stat      { $$ = linkum($1, $3); }
222         ;

224 patlist:
225           pattern
226         | patlist comma pattern         { $$ = linkum($1, $3); }
227         ;

229 ppattern:
230           var ASGNOP ppattern           { $$ = op2($2, $1, $3); }
231         | ppattern '?' ppattern ':' ppattern %prec '?'
232                 { $$ = op3(CONDEXPR, notnull($1), $3, $5); }
233         | ppattern bor ppattern %prec BOR
234                 { $$ = op2(BOR, notnull($1), notnull($3)); }
235         | ppattern and ppattern %prec AND
```

```
236                 { $$ = op2(AND, notnull($1), notnull($3)); }
216         | NOT ppattern
217                 { $$ = op1(NOT, notnull($2)); }
237         | ppattern MATCHOP reg_expr     { $$ = op3($2, NIL, $1, (Node*)makedfa($
238         | ppattern MATCHOP ppattern
239                 { if (constnode($3))
240                         $$ = op3($2, NIL, $1, (Node*)makedfa(strnode($3), 0));
241                   else
242                         $$ = op3($2, (Node *)1, $1, $3); }
243         | ppattern IN varname           { $$ = op2(INTEST, $1, makearr($3)); }
244         | '(' plist ')' IN varname      { $$ = op2(INTEST, $2, makearr($5)); }
245         | ppattern term %prec CAT       { $$ = op2(CAT, $1, $2); }
246         | re
227         | reg_expr
228                 { $$ = op3(MATCH, NIL, rectonode(), (Node*)makedfa($1, 0)); }
247         | term
248         ;

250 pattern:
251           var ASGNOP pattern            { $$ = op2($2, $1, $3); }
252         | pattern '?' pattern ':' pattern %prec '?'
253                 { $$ = op3(CONDEXPR, notnull($1), $3, $5); }
254         | pattern bor pattern %prec BOR
255                 { $$ = op2(BOR, notnull($1), notnull($3)); }
256         | pattern and pattern %prec AND
257                 { $$ = op2(AND, notnull($1), notnull($3)); }
240         | NOT pattern
241                 { $$ = op1(NOT, op2(NE,$2,valtonode(lookup((uchar *)"$zero&null"
258         | pattern EQ pattern            { $$ = op2($2, $1, $3); }
259         | pattern GE pattern            { $$ = op2($2, $1, $3); }
260         | pattern GT pattern            { $$ = op2($2, $1, $3); }
261         | pattern LE pattern            { $$ = op2($2, $1, $3); }
262         | pattern LT pattern            { $$ = op2($2, $1, $3); }
263         | pattern NE pattern            { $$ = op2($2, $1, $3); }
264         | pattern MATCHOP reg_expr      { $$ = op3($2, NIL, $1, (Node*)makedfa($
265         | pattern MATCHOP pattern
266                 { if (constnode($3))
267                         $$ = op3($2, NIL, $1, (Node*)makedfa(strnode($3), 0));
268                   else
269                         $$ = op3($2, (Node *)1, $1, $3); }
270         | pattern IN varname            { $$ = op2(INTEST, $1, makearr($3)); }
271         | '(' plist ')' IN varname      { $$ = op2(INTEST, $2, makearr($5)); }
272         | pattern '|' GETLINE var       {
273                         if (safe) SYNTAX("cmd | getline is unsafe");
274                         else $$ = op3(GETLINE, $4, itonp($2), $1); }
275         | pattern '|' GETLINE           {
276                         if (safe) SYNTAX("cmd | getline is unsafe");
277                         else $$ = op3(GETLINE, (Node*)0, itonp($2), $1); }
256         | pattern '|' GETLINE var       { $$ = op3(GETLINE, $4, (Node*)$2, $1);
257         | pattern '|' GETLINE           { $$ = op3(GETLINE, (Node*)0, (Node*)$2,
278         | pattern term %prec CAT        { $$ = op2(CAT, $1, $2); }
279         | re
259         | reg_expr
260                 { $$ = op3(MATCH, NIL, rectonode(), (Node*)makedfa($1, 0)); }
280         | term
281         ;

283 plist:
284           pattern comma pattern         { $$ = linkum($1, $3); }
285         | plist comma pattern           { $$ = linkum($1, $3); }
286         ;

288 pplist:
289           ppattern
290         | pplist comma ppattern         { $$ = linkum($1, $3); }
```



```
292 prarg:
293           /* empty */                   { $$ = rectonode(); }
294         | pplist
295         | '(' plist ')'                 { $$ = $2; }
296         ;

298 print:
299           PRINT | PRINTF
300         ;

302 pst:
303           NL | ';' | pst NL | pst ';'
304         ;

306 rbrace:
307           '}' | rbrace NL
308         ;

310 re:
311            reg_expr
312                 { $$ = op3(MATCH, NIL, rectonode(), (Node*)makedfa($1, 0)); }
313         | NOT re        { $$ = op1(NOT, notnull($2)); }
314         ;

316 #endif /* ! codereview */
317 reg_expr:
318           '/' {startreg();} REGEXPR '/'         { $$ = $3; }
319         ;

321 rparen:
322           ')' | rparen NL
323         ;

325 simple_stmt:
326           print prarg '|' term          {
327                         if (safe) SYNTAX("print | is unsafe");
328                         else $$ = stat3($1, $2, itonp($3), $4); }
329         | print prarg APPEND term       {
330                         if (safe) SYNTAX("print >> is unsafe");
331                         else $$ = stat3($1, $2, itonp($3), $4); }
332         | print prarg GT term           {
333                         if (safe) SYNTAX("print > is unsafe");
334                         else $$ = stat3($1, $2, itonp($3), $4); }
291         | print prarg '|' term          { $$ = stat3($1, $2, (Node *) $3, $4); }
292         | print prarg APPEND term       { $$ = stat3($1, $2, (Node *) $3, $4); }
293         | print prarg GT term           { $$ = stat3($1, $2, (Node *) $3, $4); }
335         | print prarg                   { $$ = stat3($1, $2, NIL, NIL); }
336         | DELETE varname '[' patlist ']' { $$ = stat2(DELETE, makearr($2), $4);
337         | DELETE varname                { $$ = stat2(DELETE, makearr($2), 0); }
296         | DELETE varname                { yyclearin; ERROR "you can only delete
338         | pattern                       { $$ = exptostat($1); }
339         | error                         { yyclearin; SYNTAX("illegal statement")
298         | error                         { yyclearin; ERROR "illegal statement" S
340         ;

342 st:
343           nl
344         | ';' opt_nl
302         | nl | ';' opt_nl
345         ;

347 stmt:
348           BREAK st              {
349                         if (!inloop) SYNTAX("break illegal outside of loops");
350                         $$ = stat1(BREAK, NIL); }
351         | CONTINUE st           {
```

```
352                         if (!inloop) SYNTAX("continue illegal outside of loops")
353                          $$ = stat1(CONTINUE, NIL); }
354         | do {inloop++;} stmt {--inloop;} WHILE '(' pattern ')' st
355                 { $$ = stat2(DO, $3, notnull($7)); }
306           BREAK st              { $$ = stat1(BREAK, NIL); }
307         | CLOSE pattern st      { $$ = stat1(CLOSE, $2); }
308         | CONTINUE st           { $$ = stat1(CONTINUE, NIL); }
309         | do stmt WHILE '(' pattern ')' st
310                 { $$ = stat2(DO, $2, notnull($5)); }
356         | EXIT pattern st       { $$ = stat1(EXIT, $2); }
357         | EXIT st               { $$ = stat1(EXIT, NIL); }
358         | for
359         | if stmt else stmt     { $$ = stat3(IF, $1, $2, $4); }
360         | if stmt               { $$ = stat3(IF, $1, $2, NIL); }
361         | lbrace stmtlist rbrace { $$ = $2; }
362         | NEXT st       { if (infunc)
363                                 SYNTAX("next is illegal inside a function");
318                                 ERROR "next is illegal inside a function" SYNTAX
364                           $$ = stat1(NEXT, NIL); }
365         | NEXTFILE st   { if (infunc)
366                                 SYNTAX("nextfile is illegal inside a function");
367                           $$ = stat1(NEXTFILE, NIL); }
368 #endif /* ! codereview */
369         | RETURN pattern st     { $$ = stat1(RETURN, $2); }
370         | RETURN st             { $$ = stat1(RETURN, NIL); }
371         | simple_stmt st
372         | while {inloop++;} stmt                { --inloop; $$ = stat2(WHILE, $1
320         | while stmt            { $$ = stat2(WHILE, $1, $2); }
373         | ';' opt_nl            { $$ = 0; }
374         ;

376 stmtlist:
377           stmt
378         | stmtlist stmt         { $$ = linkum($1, $2); }
379         ;

381 subop:
382           SUB | GSUB
383         ;

385 term:
386           term '/' ASGNOP term          { $$ = op2(DIVEQ, $1, $4); }
387         | term '+' term                 { $$ = op2(ADD, $1, $3); }
334           term '+' term                 { $$ = op2(ADD, $1, $3); }
388         | term '-' term                 { $$ = op2(MINUS, $1, $3); }
389         | term '*' term                 { $$ = op2(MULT, $1, $3); }
390         | term '/' term                 { $$ = op2(DIVIDE, $1, $3); }
391         | term '%' term                 { $$ = op2(MOD, $1, $3); }
392         | term POWER term               { $$ = op2(POWER, $1, $3); }
393         | '-' term %prec UMINUS         { $$ = op1(UMINUS, $2); }
394         | '+' term %prec UMINUS         { $$ = $2; }
395         | NOT term %prec UMINUS         { $$ = op1(NOT, notnull($2)); }
396         | BLTIN '(' ')'                 { $$ = op2(BLTIN, itonp($1), rectonode()
397         | BLTIN '(' patlist ')'         { $$ = op2(BLTIN, itonp($1), $3); }
398         | BLTIN                         { $$ = op2(BLTIN, itonp($1), rectonode()
399         | CALL '(' ')'                  { $$ = op2(CALL, celltonode($1,CVAR), NI
400         | CALL '(' patlist ')'          { $$ = op2(CALL, celltonode($1,CVAR), $3
401         | CLOSE term                    { $$ = op1(CLOSE, $2); }
342         | BLTIN '(' ')'                 { $$ = op2(BLTIN, (Node *) $1, rectonode
343         | BLTIN '(' patlist ')'         { $$ = op2(BLTIN, (Node *) $1, $3); }
344         | BLTIN                         { $$ = op2(BLTIN, (Node *) $1, rectonode
345         | CALL '(' ')'                  { $$ = op2(CALL, valtonode($1,CVAR), NIL
346         | CALL '(' patlist ')'          { $$ = op2(CALL, valtonode($1,CVAR), $3)
402         | DECR var                      { $$ = op1(PREDECR, $2); }
403         | INCR var                      { $$ = op1(PREINCR, $2); }
404         | var DECR                      { $$ = op1(POSTDECR, $1); }
```



```
405         | var INCR                      { $$ = op1(POSTINCR, $1); }
406         | GETLINE var LT term           { $$ = op3(GETLINE, $2, itonp($3), $4);
407         | GETLINE LT term               { $$ = op3(GETLINE, NIL, itonp($2), $3);
351         | GETLINE var LT term           { $$ = op3(GETLINE, $2, (Node *)$3, $4);
352         | GETLINE LT term               { $$ = op3(GETLINE, NIL, (Node *)$2, $3)
408         | GETLINE var                   { $$ = op3(GETLINE, $2, NIL, NIL); }
409         | GETLINE                       { $$ = op3(GETLINE, NIL, NIL, NIL); }
410         | INDEX '(' pattern comma pattern ')'
411                 { $$ = op2(INDEX, $3, $5); }
412         | INDEX '(' pattern comma reg_expr ')'
413                 { SYNTAX("index() doesn't permit regular expressions");
414                   /* LINTED align */
358                 { ERROR "index() doesn't permit regular expressions" SYNTAX;
415                   $$ = op2(INDEX, $3, (Node*)$5); }
416         | '(' pattern ')'               { $$ = $2; }
417         | MATCHFCN '(' pattern comma reg_expr ')'
418                 { $$ = op3(MATCHFCN, NIL, $3, (Node*)makedfa($5, 1)); }
419         | MATCHFCN '(' pattern comma pattern ')'
420                 { if (constnode($5))
421                         $$ = op3(MATCHFCN, NIL, $3, (Node*)makedfa(strnode($5),
422                   else
423                         $$ = op3(MATCHFCN, (Node *)1, $3, $5); }
424         | NUMBER                        { $$ = celltonode($1, CCON); }
368         | NUMBER                        { $$ = valtonode($1, CCON); }
425         | SPLIT '(' pattern comma varname comma pattern ')'     /* string */
426                 { $$ = op4(SPLIT, $3, makearr($5), $7, (Node*)STRING); }
427         | SPLIT '(' pattern comma varname comma reg_expr ')'    /* const /regexp
428                 { $$ = op4(SPLIT, $3, makearr($5), (Node*)makedfa($7, 1), (Node
429         | SPLIT '(' pattern comma varname ')'
430                 { $$ = op4(SPLIT, $3, makearr($5), NIL, (Node*)STRING); }  /* de
431         | SPRINTF '(' patlist ')'       { $$ = op1($1, $3); }
432         | STRING                        { $$ = celltonode($1, CCON); }
376         | STRING                        { $$ = valtonode($1, CCON); }
433         | subop '(' reg_expr comma pattern ')'
434                 { $$ = op4($1, NIL, (Node*)makedfa($3, 1), $5, rectonode()); }
435         | subop '(' pattern comma pattern ')'
436                 { if (constnode($3))
437                         $$ = op4($1, NIL, (Node*)makedfa(strnode($3), 1), $5, re
438                   else
439                         $$ = op4($1, (Node *)1, $3, $5, rectonode()); }
440         | subop '(' reg_expr comma pattern comma var ')'
441                 { $$ = op4($1, NIL, (Node*)makedfa($3, 1), $5, $7); }
442         | subop '(' pattern comma pattern comma var ')'
443                 { if (constnode($3))
444                         $$ = op4($1, NIL, (Node*)makedfa(strnode($3), 1), $5, $7
445                   else
446                         $$ = op4($1, (Node *)1, $3, $5, $7); }
447         | SUBSTR '(' pattern comma pattern comma pattern ')'
448                 { $$ = op3(SUBSTR, $3, $5, $7); }
449         | SUBSTR '(' pattern comma pattern ')'
450                 { $$ = op3(SUBSTR, $3, $5, NIL); }
451         | var
452         ;

454 var:
455           varname
456         | varname '[' patlist ']'       { $$ = op2(ARRAY, makearr($1), $3); }
457         | IVAR                          { $$ = op1(INDIRECT, celltonode($1, CVAR
401         | FIELD                         { $$ = valtonode($1, CFLD); }
402         | IVAR                          { $$ = op1(INDIRECT, valtonode($1, CVAR)
458         | INDIRECT term                 { $$ = op1(INDIRECT, $2); }
459         ;

461 varlist:
462           /* nothing */         { arglist = $$ = 0; }
463         | VAR                   { arglist = $$ = celltonode($1,CVAR); }
```

```
464         | varlist comma VAR     {
465                                   checkdup($1, $3);
466                                   arglist = $$ = linkum($1,celltonode($3,CVAR)); }
408         | VAR                   { arglist = $$ = valtonode($1,CVAR); }
409         | varlist comma VAR     { arglist = $$ = linkum($1,valtonode($3,CVAR));
467         ;

469 varname:
470           VAR                   { $$ = celltonode($1, CVAR); }
471         | ARG                   { $$ = op1(ARG, itonp($1)); }
413           VAR                   { $$ = valtonode($1, CVAR); }
414         | ARG                   { $$ = op1(ARG, (Node *) $1); }
472         | VARNF                 { $$ = op1(VARNF, (Node *) $1); }
473         ;


476 while:
477           WHILE '(' pattern rparen      { $$ = notnull($3); }
478         ;

480 %%

482 static void
483 setfname(Cell *p)
484 {
485         if (isarr(p))
486                 SYNTAX("%s is an array, not a function", p->nval);
487         else if (isfcn(p))
488                 SYNTAX("you can't define function %s more than once", p->nval);
429                 ERROR "%s is an array, not a function", p->nval SYNTAX;
430         else if (isfunc(p))
431                 ERROR "you can't define function %s more than once", p->nval SYN
489         curfname = p->nval;
490 }


493 static int
494 constnode(Node *p)
495 {
496         return (isvalue(p) && ((Cell *) (p->narg[0]))->csub == CCON);
439         return p->ntype == NVALUE && ((Cell *) (p->narg[0]))->csub == CCON;
497 }

499 static uchar *
500 strnode(Node *p)
501 {
502         return (((Cell *)(p->narg[0]))->sval);
445         return ((Cell *)(p->narg[0]))->sval;
503 }

505 static Node *
506 notnull(Node *n)
507 {
508         switch (n->nobj) {
509         case LE: case LT: case EQ: case NE: case GT: case GE:
510         case BOR: case AND: case NOT:
511                 return (n);
454                 return n;
512         default:
513                 return (op2(NE, n, nullnode));
514         }
515 }

517 static void
518 checkdup(Node *vl, Cell *cp)    /* check if name already in list */
519 {
```



```
520         uchar *s = cp->nval;

522         for (; vl; vl = vl->nnext) {
523                 if (strcmp((char *)s, (char *)((Cell *)(vl->narg[0]))->nval) ==
524                         SYNTAX("duplicate argument %s", s);
525                         break;
526                 }
456                 return op2(NE, n, nullnode);
527         }
528 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   11430 Sat May 11 15:20:42 2013
new/usr/src/cmd/awk/awk.h
3731 Update nawk to version 20121220
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright (c) 1989, 2010, Oracle and/or its affiliates. All rights reserved.
 */

/*
 * Copyright (C) Lucent Technologies 1997
 * All Rights Reserved
 *
 * Permission to use, copy, modify, and distribute this software and
 * its documentation for any purpose and without fee is hereby
 * granted, provided that the above copyright notice appear in all
 * copies and that both that the copyright notice and this
 * permission notice and warranty disclaimer appear in supporting
 * documentation, and that the name Lucent Technologies or any of
 * its entities not be used in advertising or publicity pertaining
 * to distribution of the software without specific, written prior
 * permission.
 *
 * LUCENT DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
 * INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS.
 * IN NO EVENT SHALL LUCENT OR ANY OF ITS ENTITIES BE LIABLE FOR ANY
 * SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
 * WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER
 * IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION,
 * ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF
 * THIS SOFTWARE.
 */
/*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
/*        All Rights Reserved   */

#ifndef AWK_H
#define AWK_H

#include <sys/types.h>
#include <assert.h>
#endif /* ! codereview */
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```



```
#include <libintl.h>
#include <limits.h>

typedef double  Awkfloat;
typedef unsigned char uchar;

#define xfree(a)        { if ((a) != NULL) { free((void *)(a)); (a) = NULL; } }
#define xfree(a)        { if ((a) != NULL) { free(a); a = NULL; } }

#define DEBUG
#ifdef  DEBUG
                        /* uses have to be doubly parenthesized */
#define dprintf(x)      if (dbg) (void) printf x
#else
#define dprintf(x)
#endif

extern  char    errbuf[200];
extern  void    error(int, char *);
#define ERROR   (void) snprintf(errbuf, sizeof (errbuf),
/*CSTYLED*/
#define FATAL   ), error(1, errbuf)
/*CSTYLED*/
#define WARNING ), error(0, errbuf)
/*CSTYLED*/
#define SYNTAX  ), yyerror(errbuf)
/*CSTYLED*/
#define CONT    )

extern int      compile_time;   /* 1 if compiling, 0 if running */
extern int      safe;           /* 0 => unsafe, 1 => safe */
#endif /* ! codereview */

#define RECSIZE (8 * 1024)      /* sets limit on records, fields, etc., etc. */
#define FLD_INCR        64
#define LINE_INCR       256

/* ensure that there is extra 1 byte in the buffer */
#define expand_buf(p, n, r)     \
        if (*(n) == 0 || (r) >= (*(n) - 1)) r_expand_buf(p, n, r)

extern uchar    **FS;
extern uchar    **RS;
extern uchar    **ORS;
extern uchar    **OFS;
extern uchar    **OFMT;
extern Awkfloat *NR;
extern Awkfloat *FNR;
extern Awkfloat *NF;
extern uchar    **FILENAME;
extern uchar    **SUBSEP;
extern Awkfloat *RSTART;
extern Awkfloat *RLENGTH;

extern uchar    *record;
extern size_t   record_size;
extern int      errorflag;
extern int      donefld;        /* 1 if record broken into fields */
extern int      donerec;        /* 1 if record is valid (no fld has changed */

extern  uchar   *patbeg;        /* beginning of pattern matched */
extern  int     patlen;         /* length.  set in b.c */

/* Cell:  all information about a variable or constant */

typedef struct Cell {
```

```
        uchar   ctype;          /* OCELL, OBOOL, OJUMP, etc. */
        uchar   csub;           /* CCON, CTEMP, CFLD, etc. */
        uchar   *nval;          /* name, for variables only */
        uchar   *sval;          /* string value */
        Awkfloat fval;          /* value as number */
        int     tval;
        unsigned tval;
                /* type info: STR|NUM|ARR|FCN|FLD|CON|DONTFREE */
        struct Cell *cnext;     /* ptr to next if chained */
} Cell;

typedef struct Array {          /* symbol table array */
typedef struct {                /* symbol table array */
        int     nelem;          /* elements in table right now */
        int     size;           /* size of tab */
        Cell    **tab;          /* hash table pointers */
} Array;

#define NSYMTAB 50      /* initial size of a symbol table */
extern Array    *symtab, *makesymtab(int);
extern Cell     *setsymtab(const uchar *, const uchar *, Awkfloat,
    unsigned int, Array *);
extern Cell     *lookup(const uchar *, Array *);
extern Cell     *setsymtab(uchar *, uchar *, Awkfloat, unsigned int, Array *);
extern Cell     *lookup(uchar *, Array *);

extern Cell     *recloc;        /* location of input record */
extern Cell     *nrloc;         /* NR */
extern Cell     *fnrloc;        /* FNR */
extern Cell     *nfloc;         /* NF */
extern Cell     *rstartloc;     /* RSTART */
extern Cell     *rlengthloc;    /* RLENGTH */

/* Cell.tval values: */
#define NUM     01      /* number value is valid */
#define STR     02      /* string value is valid */
#define DONTFREE 04     /* string space is not freeable */
#define CON     010     /* this is a constant */
#define ARR     020     /* this is an array */
#define FCN     040     /* this is a function name */
#define FLD     0100    /* this is a field $1, $2, ... */
#define REC     0200    /* this is $0 */

#define freeable(p)     (!((p)->tval & DONTFREE))

extern Awkfloat setfval(Cell *, Awkfloat), getfval(Cell *), r_getfval(Cell *);
extern  uchar   *setsval(Cell *, const uchar *);
extern  uchar   *getsval(Cell *);
extern  uchar   *tostring(const uchar *);
extern  uchar   *tokname(int);
extern  uchar   *qstring(const uchar *, int);
extern uchar    *setsval(Cell *, uchar *), *getsval(Cell *), *r_getsval(Cell *);
extern uchar    *tostring(uchar *), *tokname(int), *qstring(uchar *, int);

#define getfval(p)      \
        (((p)->tval & (ARR|FLD|REC|NUM)) == NUM ? (p)->fval : r_getfval(p))
#define getsval(p)      \
        (((p)->tval & (ARR|FLD|REC|STR)) == STR ? (p)->sval : r_getsval(p))

/* function types */
#define FLENGTH 1
#define FSQRT   2
#define FEXP    3
#define FLOG    4
#define FINT    5
#define FSYSTEM 6
```



```
#define FRAND   7
#define FSRAND  8
#define FSIN    9
#define FCOS    10
#define FATAN   11
#define FTOUPPER 12
#define FTOLOWER 13
#define FFLUSH  14
#endif /* ! codereview */

/* Node:  parse tree is made of nodes, with Cell's at bottom */

typedef struct Node {
        int     ntype;
        struct  Node *nnext;
        off_t lineno;
        int     nobj;
        struct Node *narg[1];
                /* variable: actual size set by calling malloc */
} Node;

#define NIL     ((Node *)0)

extern Node     *winner;
extern Node     *nullstat;
extern Node     *nullnode;

/* ctypes */
#define OCELL   1
#define OBOOL   2
#define OJUMP   3

/* Cell subtypes: csub */
#define CFREE   7
#define CCOPY   6
#define CCON    5
#define CTEMP   4
#define CNAME   3
#define CVAR    2
#define CFLD    1
#define CUNK    0
#endif /* ! codereview */

/* bool subtypes */
#define BTRUE   11
#define BFALSE  12

/* jump subtypes */
#define JEXIT   21
#define JNEXT   22
#define JBREAK  23
#define JCONT   24
#define JRET    25
#define JNEXTFILE       26
#endif /* ! codereview */

/* node types */
#define NVALUE  1
#define NSTAT   2
#define NEXPR   3
#define NFIELD  4

extern  Cell    *(*proctab[])(Node **, int);
extern  Cell    *nullproc(Node **, int);
extern  int     pairstack[], paircnt;
```

```
227 extern  int     pgetc(void);
228 extern  Node    *stat1(int, Node *);
229 extern  Node    *stat2(int, Node *, Node *);
151 extern  Node    *stat1(int, Node *), *stat2(int, Node *, Node *);
230 extern  Node    *stat3(int, Node *, Node *, Node *);
231 extern  Node    *stat4(int, Node *, Node *, Node *, Node *);
232 extern  Node    *pa2stat(Node *, Node *, Node *);
233 extern  Node    *op1(int, Node *);
234 extern  Node    *op2(int, Node *, Node *);
155 extern  Node    *op1(int, Node *), *op2(int, Node *, Node *);
235 extern  Node    *op3(int, Node *, Node *, Node *);
236 extern  Node    *op4(int, Node *, Node *, Node *, Node *);
237 extern  Node    *linkum(Node *, Node *);
238 extern  Node    *celltonode(Cell *, int);
158 extern  Node    *linkum(Node *, Node *), *valtonode(Cell *, int);
239 extern  Node    *rectonode(void), *exptostat(Node *);
240 extern  Node    *makearr(Node *);
241 extern  Node    *itonp(int);
242 #endif /* ! codereview */

244 #define notlegal(n)     \
245         (n <= FIRSTTOKEN || n >= LASTTOKEN || proctab[n-FIRSTTOKEN] == nullproc)
246 #define isvalue(n)      ((n)->ntype == NVALUE)
247 #define isexpr(n)       ((n)->ntype == NEXPR)
248 #define isjump(n)       ((n)->ctype == OJUMP)
249 #define isexit(n)       ((n)->csub == JEXIT)
250 #define isbreak(n)      ((n)->csub == JBREAK)
251 #define iscont(n)       ((n)->csub == JCONT)
252 #define isnext(n)       ((n)->csub == JNEXT || (n)->csub == JNEXTFILE)
161 #define isnext(n)       ((n)->csub == JNEXT)
253 #define isret(n)        ((n)->csub == JRET)
254 #define isrec(n)        ((n)->tval & REC)
255 #define isfld(n)        ((n)->tval & FLD)
256 #endif /* ! codereview */
257 #define isstr(n)        ((n)->tval & STR)
258 #define isnum(n)        ((n)->tval & NUM)
259 #define isarr(n)        ((n)->tval & ARR)
260 #define isfcn(n)        ((n)->tval & FCN)
163 #define isfunc(n)       ((n)->tval & FCN)
261 #define istrue(n)       ((n)->csub == BTRUE)
262 #define istemp(n)       ((n)->csub == CTEMP)
263 #define isargument(n)   ((n)->nobj == ARG)
264 /* #define      freeable(p)     (!((p)->tval & DONTFREE)) */
265 #define freeable(p)     (((p)->tval & (STR|DONTFREE)) == STR)
266 #endif /* ! codereview */

268 #define NCHARS  (256+3)         /* 256 handles 8-bit chars; 128 does 7-bit */
269                                 /* watch out in match(), etc. */
166 #define NCHARS  (256+1)
270 #define NSTATES 32

272 typedef struct rrow {
273         long    ltype;  /* long avoids pointer warnings on 64-bit */
274         union {
275                 int i;
276                 Node *np;
277                 uchar *up;
278         } lval;         /* because Al stores a pointer in it! */
170         int     ltype;
171         int     lval;
279         int     *lfollow;
280 } rrow;

282 typedef struct fa {
283         uchar   gototab[NSTATES][NCHARS];
284         uchar   out[NSTATES];
```



```
285 #endif /* ! codereview */
286         uchar   *restr;
287         int     *posns[NSTATES];
288 #endif /* ! codereview */
289         int     anchor;
290         int     use;
176         uchar   gototab[NSTATES][NCHARS];
177         int     *posns[NSTATES];
178         uchar   out[NSTATES];
291         int     initstat;
292         int     curstat;
293         int     accept;
294         int     reset;
295         struct  rrow re[1]; /* variable: actual size set by calling malloc */
183         struct  rrow re[1];
296 } fa;

298 /* b.c */
299 extern  fa      *makedfa(const uchar *, int);
300 extern  int     nematch(fa *, const uchar *);
301 extern  int     match(fa *, const uchar *);
302 extern  int     pmatch(fa *, const uchar *);

304 /* lex.c */
305 extern  void    unput(int);
306 extern  void    unputstr(const char *);
307 extern  void    startreg(void);
187 extern  fa      *makedfa(uchar *, int);
188 extern  int     nematch(fa *, uchar *);
189 extern  int     match(fa *, uchar *);
190 extern  int     pmatch(fa *, uchar *);

309 /* lib.c */
310 extern  void    recinit(unsigned int);
311 extern  void    growfldtab(int);
312 extern  int     isclvar(const uchar *);
313 extern  int     is_number(const uchar *);
193 extern  int     isclvar(uchar *);
194 extern  int     is_number(uchar *);
314 extern  void    setclvar(uchar *);
315 extern  int     readrec(uchar **, size_t *, FILE *);
316 extern  void    bracecheck(void);
317 extern  void    syminit(void);
318 extern  void    yyerror(char *);
319 extern  void    fldbld(void);
320 extern  void    recbld(void);
321 extern  int     getrec(uchar **, size_t *, int);
202 extern  int     getrec(uchar **, size_t *);
322 extern  Cell    *fieldadr(int);
323 extern  void    newfld(int);
324 extern  Cell    *getfld(int);
325 extern  int     fldidx(Cell *);
326 extern  void    SYNTAX(const char *, ...);
327 extern  void    FATAL(const char *, ...) __attribute__((__noreturn__));
328 extern  void    WARNING(const char *, ...);
329 extern  void    error(void);
330 extern  double  errcheck(double, const char *);
207 extern  double  errcheck(double, char *);
331 extern  void    fpecatch(int);
332 extern  void    nextfile(void);
209 extern  void    init_buf(uchar **, size_t *, size_t);
210 extern  void    adjust_buf(uchar **, size_t);
211 extern  void    r_expand_buf(uchar **, size_t *, size_t);

334 extern  int     donefld;
335 extern  int     donerec;
```

```
336 extern  uchar   *record;
337 extern  size_t  record_size;

339 /* main.c */
340 extern  int     dbg;
341 extern  uchar   *cmdname;
342 extern  uchar   *lexprog;
343 extern  int     compile_time;
344 extern  char    radixpoint;
345 extern  uchar   *cursource(void);
346 #endif /* ! codereview */

348 /* tran.c */
349 extern  void    syminit(void);
350 extern  void    arginit(int, uchar **);
351 extern  void    envinit(uchar **);
352 extern  void    freesymtab(Cell *);
353 extern  void    freeelem(Cell *, const uchar *);
224 extern  void    freeelem(Cell *, uchar *);
354 extern  void    funnyvar(Cell *, char *);
355 extern  int     hash(const uchar *, int);
226 extern  int     hash(uchar *, int);
356 extern  Awkfloat *ARGC;

358 /* run.c */
359 extern  int     adjbuf(uchar **, size_t *, int, int, uchar **, const char *);
360 #endif /* ! codereview */
361 extern  void    run(Node *);

363 extern  int     paircnt;
364 extern  Node    *winner;

366 #ifndef input
367 extern  int     input(void);
368 #endif
369 extern  int     yyparse(void);
370 extern  FILE    *yyin;
371 extern  off_t   lineno;

373 /* parse.c */
374 extern  int     ptoi(void *);
375 extern  int     isarg(const uchar *);
376 extern  void    defn(Cell *, Node *, Node *);

378 #endif /* ! codereview */
379 /* proc */
380 extern Cell *nullproc(Node **, int);
381 extern Cell *program(Node **, int);
382 extern Cell *boolop(Node **, int);
383 extern Cell *relop(Node **, int);
384 extern Cell *array(Node **, int);
385 extern Cell *indirect(Node **, int);
386 extern Cell *substr(Node **, int);
387 extern Cell *sub(Node **, int);
388 extern Cell *gsub(Node **, int);
389 extern Cell *sindex(Node **, int);
390 extern Cell *awksprintf(Node **, int);
230 extern Cell *a_sprintf(Node **, int);
391 extern Cell *arith(Node **, int);
392 extern Cell *incrdecr(Node **, int);
393 extern Cell *cat(Node **, int);
394 extern Cell *pastat(Node **, int);
395 extern Cell *dopa2(Node **, int);
396 extern Cell *matchop(Node **, int);
397 extern Cell *intest(Node **, int);
398 extern Cell *awkprintf(Node **, int);
```



```
399 extern Cell *printstat(Node **, int);
238 extern Cell *aprintf(Node **, int);
239 extern Cell *print(Node **, int);
400 extern Cell *closefile(Node **, int);
401 extern Cell *awkdelete(Node **, int);
241 extern Cell *delete(Node **, int);
402 extern Cell *split(Node **, int);
403 extern Cell *assign(Node **, int);
404 extern Cell *condexpr(Node **, int);
405 extern Cell *ifstat(Node **, int);
406 extern Cell *whilestat(Node **, int);
407 extern Cell *forstat(Node **, int);
408 extern Cell *dostat(Node **, int);
409 extern Cell *instat(Node **, int);
410 extern Cell *jump(Node **, int);
411 extern Cell *bltin(Node **, int);
412 extern Cell *call(Node **, int);
413 extern Cell *arg(Node **, int);
414 extern Cell *getnf(Node **, int);
415 extern Cell *awkgetline(Node **, int);
255 extern Cell *getaline(Node **, int);

417 #endif /* AWK_H */
```

```
**********************************************************
   24521 Sat May 11 15:20:43 2013
new/usr/src/cmd/awk/b.c
3731 Update nawk to version 20121220
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License, Version 1.0 only
 * (the "License").  You may not use this file except in compliance
 * with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright 2005 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * Copyright (C) Lucent Technologies 1997
 * All Rights Reserved
 *
 * Permission to use, copy, modify, and distribute this software and
 * its documentation for any purpose and without fee is hereby
 * granted, provided that the above copyright notice appear in all
 * copies and that both that the copyright notice and this
 * permission notice and warranty disclaimer appear in supporting
 * documentation, and that the name Lucent Technologies or any of
 * its entities not be used in advertising or publicity pertaining
 * to distribution of the software without specific, written prior
 * permission.
 *
 * LUCENT DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
 * INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS.
 * IN NO EVENT SHALL LUCENT OR ANY OF ITS ENTITIES BE LIABLE FOR ANY
 * SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
 * WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER
 * IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION,
 * ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF
 * THIS SOFTWARE.
 */
/*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
/*        All Rights Reserved   */

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

#define DEBUG

#include "awk.h"
#include "y.tab.h"

#define HAT     (NCHARS+2)      /* matches ^ in regular expr */
```



```
#define HAT     (NCHARS-1)      /* matches ^ in regular expr */
                                /* NCHARS is 2**n */
#define MAXLIN  22
#define MAXLIN (3 * LINE_MAX)

#define type(v)         (v)->nobj       /* badly overloaded here */
#define info(v)         (v)->ntype      /* badly overloaded here */
#define type(v)         (v)->nobj
#define left(v)         (v)->narg[0]
#define right(v)        (v)->narg[1]
#define parent(v)       (v)->nnext

#define LEAF    case CCL: case NCCL: case CHAR: case DOT: case FINAL: case ALL:
#define ELEAF   case EMPTYRE:           /* empty string in regexp */
#endif /* ! codereview */
#define UNARY   case STAR: case PLUS: case QUEST:

/*
 * encoding in tree Nodes:
 *      leaf (CCL, NCCL, CHAR, DOT, FINAL, ALL):
 *              left is index, right contains value or pointer to value
 *      unary (STAR, PLUS, QUEST): left is child, right is null
 *      binary (CAT, OR): left and right are children
 *      parent contains pointer to parent
 */

int     *setvec;
int     *tmpset;
int     maxsetvec = 0;
int     setvec[MAXLIN];
int     tmpset[MAXLIN];
Node    *point[MAXLIN];

int     rtok;           /* next token in current re */
int     rlxval;
static  uchar   *rlxstr;
static  uchar   *prestr;        /* current position in current re */
static  uchar   *lastre;        /* origin of last re */
uchar   *rlxstr;
uchar   *prestr;        /* current position in current re */
uchar   *lastre;        /* origin of last re */

static  int setcnt;
static  int poscnt;

uchar   *patbeg;
int     patlen;

#define NFA     20      /* cache this many dynamic fa's */
fa      *fatab[NFA];
int     nfatab  = 0;    /* entries in fatab */

static fa       *mkdfa(const uchar *, int);
static fa       *mkdfa(uchar *, int);
static int      makeinit(fa *, int);
static void     penter(Node *);
static void     freetr(Node *);
static void     overflo(const char *);
static void     overflo(char *);
static void     cfoll(fa *, Node *);
static void     follow(Node *);
static Node     *reparse(const uchar *);
static Node     *reparse(uchar *);
static int      relex(void);
static void     freefa(fa *);
static int      cgoto(fa *, int, int);
```

```
112 static Node     *regexp(void);
113 static Node     *primary(void);
114 static Node     *concat(Node *);
115 static Node     *alt(Node *);
116 static Node     *unary(Node *);
117 #endif /* ! codereview */

119 fa *
120 makedfa(const uchar *s, int anchor)     /* returns dfa for reg expr s */
 79 makedfa(uchar *s, int anchor)   /* returns dfa for reg expr s */
121 {
122         int i, use, nuse;
123         fa *pfa;
124         static int now = 1;

126         if (setvec == 0) {      /* first time through any RE */
127                 maxsetvec = MAXLIN;
128                 setvec = (int *)malloc(maxsetvec * sizeof (int));
129                 tmpset = (int *)malloc(maxsetvec * sizeof (int));
130                 if (setvec == 0 || tmpset == 0)
131                         overflo("out of space initializing makedfa");
132         }
133 #endif /* ! codereview */

135         if (compile_time)       /* a constant for sure */
136                 return (mkdfa(s, anchor));
137         for (i = 0; i < nfatab; i++) {  /* is it there already? */
138                 if (fatab[i]->anchor == anchor &&
139                     strcmp((const char *)fatab[i]->restr, (char *)s) == 0) {
140                         fatab[i]->use = now++;
 83                     strcmp((char *)fatab[i]->restr, (char *)s) == 0) {
 84                         fatab[i]->use++;
141                         return (fatab[i]);
142                 }
143         }
144         pfa = mkdfa(s, anchor);
145         if (nfatab < NFA) {     /* room for another */
146                 fatab[nfatab] = pfa;
147                 fatab[nfatab]->use = now++;
 91                 fatab[nfatab]->use = 1;
148                 nfatab++;
149                 return (pfa);
150         }
151         use = fatab[0]->use;    /* replace least-recently used */
152         nuse = 0;
153         for (i = 1; i < nfatab; i++) {
 97         for (i = 1; i < nfatab; i++)
154                 if (fatab[i]->use < use) {
155                         use = fatab[i]->use;
156                         nuse = i;
157                 }
158         }
159 #endif /* ! codereview */
160         freefa(fatab[nuse]);
161         fatab[nuse] = pfa;
162         pfa->use = now++;
102         pfa->use = 1;
163         return (pfa);
164 }

166 fa *
167 mkdfa(const uchar *s, int anchor)
168 {                               /* does the real work of making a dfa */
107 mkdfa(uchar *s, int anchor)     /* does the real work of making a dfa */
169                                 /* anchor = 1 for anchored matches, else 0 */
109 {
```



```
170         Node *p, *p1;
171         fa *f;

173         p = reparse(s);
174         p1 = op2(CAT, op2(STAR, op2(ALL, NIL, NIL), NIL), p);
175                 /* put ALL STAR in front of reg.  exp. */
176         p1 = op2(CAT, p1, op2(FINAL, NIL, NIL));
177                 /* put FINAL after reg.  exp. */

179         poscnt = 0;
180         penter(p1);     /* enter parent pointers and leaf indices */
181         if ((f = (fa *)calloc(1, sizeof (fa) + poscnt * sizeof (rrow))) == NULL)
182                 overflo("out of space for fa");
122                 overflo("no room for fa");
183         /* penter has computed number of positions in re */
184         f->accept = poscnt-1;
185         cfoll(f, p1);   /* set up follow sets */
186         freetr(p1);
187         if ((f->posns[0] =
188             (int *)calloc(1, *(f->re[0].lfollow) * sizeof (int))) == NULL) {
189                         overflo("out of space in makedfa");
190         }
191         if ((f->posns[1] = (int *)calloc(1, sizeof (int))) == NULL)
192                 overflo("out of space in makedfa");
193         *f->posns[1] = 0;
194         f->initstat = makeinit(f, anchor);
195         f->anchor = anchor;
196         f->restr = tostring(s);
197         return (f);
198 }

200 static int
201 makeinit(fa *f, int anchor)
202 {
203         int i, k;
143         register int i, k;

205         f->curstat = 2;
206         f->out[2] = 0;
207         f->reset = 0;
208         k = *(f->re[0].lfollow);
209         xfree(f->posns[2]);
210         if ((f->posns[2] = (int *)calloc(1, (k+1) * sizeof (int))) == NULL)
211                 overflo("out of space in makeinit");
212         for (i = 0; i <= k; i++) {
213                 (f->posns[2])[i] = (f->re[0].lfollow)[i];
214         }
215         if ((f->posns[2])[1] == f->accept)
216                 f->out[2] = 1;
217         for (i = 0; i < NCHARS; i++)
218                 f->gototab[2][i] = 0;
219         f->curstat = cgoto(f, 2, HAT);
220         if (anchor) {
221                 *f->posns[2] = k-1;     /* leave out position 0 */
222                 for (i = 0; i < k; i++) {
223                         (f->posns[0])[i] = (f->posns[2])[i];
224                 }

226                 f->out[0] = f->out[2];
227                 if (f->curstat != 2)
228                         --(*f->posns[f->curstat]);
229         }
230         return (f->curstat);
231 }

233 void
```

```
234 penter(Node *p) /* set up parent pointers and leaf indices */
235 {
236         switch (type(p)) {
237         ELEAF
238 #endif /* ! codereview */
239         LEAF
240                 info(p) = poscnt;
241                 poscnt++;
177                 left(p) = (Node *) poscnt;
178                 point[poscnt++] = p;
242                 break;
243         UNARY
244                 penter(left(p));
245                 parent(left(p)) = p;
246                 break;
247         case CAT:
248         case OR:
249                 penter(left(p));
250                 penter(right(p));
251                 parent(left(p)) = p;
252                 parent(right(p)) = p;
253                 break;
254         default:        /* can't happen */
255                 FATAL("can't happen: unknown type %d in penter", type(p));
191         default:
192                 ERROR "unknown type %d in penter", type(p) FATAL;
256                 break;
257         }
258 }

260 static void
261 freetr(Node *p) /* free parse tree */
262 {
263         switch (type(p)) {
264         ELEAF
265 #endif /* ! codereview */
266         LEAF
267                 xfree(p);
268                 break;
269         UNARY
270                 freetr(left(p));
271                 xfree(p);
272                 break;
273         case CAT:
274         case OR:
275                 freetr(left(p));
276                 freetr(right(p));
277                 xfree(p);
278                 break;
279         default:        /* can't happen */
280                 FATAL("can't happen: unknown type %d in freetr", type(p));
201         default:
202                 ERROR "unknown type %d in freetr", type(p) FATAL;
281                 break;
282         }
283 }

285 /*
286  * in the parsing of regular expressions, metacharacters like . have
287  * to be seen literally;  \056 is not a metacharacter.
288  */
289 int
290 hexstr(uchar **pp)      /* find and eval hex string at pp, return new p */
291 {                       /* only pick up one 8-bit byte (2 chars) */
292         uchar *p;
293         int n = 0;
```



```
294         int i;

296         for (i = 0, p = (uchar *)*pp; i < 2 && isxdigit(*p); i++, p++) {
297                 if (isdigit(*p))
298                         n = 16 * n + *p - '0';
299                 else if (*p >= 'a' && *p <= 'f')
300                         n = 16 * n + *p - 'a' + 10;
301                 else if (*p >= 'A' && *p <= 'F')
302                         n = 16 * n + *p - 'A' + 10;
303         }
304         *pp = (uchar *)p;
305         return (n);
306 }

308 /* multiple use of arg */
309 #define isoctdigit(c)   ((c) >= '0' && (c) <= '7')

311 int
312 quoted(uchar **pp)      /* pick up next thing after a \\ */
313 {                       /* and increment *pp */

315         uchar *p = *pp;
316         int c;
207 uchar *
208 cclenter(uchar *p)
209 {
210         register int i, c;
211         uchar *op, *chars, *ret;
212         size_t  bsize;

214         init_buf(&chars, &bsize, LINE_INCR);
215         op = p;
216         i = 0;
217         while ((c = *p++) != 0) {
218                 if (c == '\\') {
318         if ((c = *p++) == 't')
319                 c = '\t';
320         else if (c == 'n')
321                 c = '\n';
322         else if (c == 'f')
323                 c = '\f';
324         else if (c == 'r')
325                 c = '\r';
326         else if (c == 'b')
327                 c = '\b';
328         else if (c == '\\')
329                 c = '\\';
330         else if (c == 'x') {    /* hexadecimal goo follows */
331                 c = hexstr(&p); /* this adds a null if number is invalid */
332         } else if (isoctdigit(c)) {     /* \d \dd \ddd */
231                         else if (isdigit(c)) {
333                 int n = c - '0';
334                 if (isoctdigit(*p)) {
233                                 if (isdigit(*p)) {
335                         n = 8 * n + *p++ - '0';
336                         if (isoctdigit(*p))
235                                         if (isdigit(*p))
337                                 n = 8 * n + *p++ - '0';
338                 }
339                 c = n;
340         } /* else */
341                 /* c = c; */
342         *pp = p;
343         return (c);
344 }
```

```
uchar *
cclenter(const uchar *argp)             /* add a character class */
{
        int i, c, c2;
        uchar  *p = (uchar *)argp;
        uchar *op, *bp;
        static uchar *buf = NULL;
        size_t  bufsz = 100;

        op = p;
        if (buf == 0 && (buf = (uchar *)malloc(bufsz)) == NULL)
                FATAL("out of space for character class [%.10s...] 1", p);
        bp = buf;
        for (i = 0; (c = *p++) != 0; ) {
                if (c == '\\') {
                        c = quoted(&p);
                } else if (c == '-' && i > 0 && bp[-1] != 0) {
                } else if (c == '-' && i > 0 && chars[i-1] != 0) {
                        if (*p != 0) {
                                c = bp[-1];
                                c2 = *p++;
                                if (c2 == '\\')
                                        c2 = quoted(&p);
                                if (c > c2) {   /* empty; ignore */
                                        bp--;
                                        i--;
                                        continue;
                                }
                                while (c < c2) {
                                        if (adjbuf(&buf, &bufsz,
                                            bp - buf + 2, 100, &bp,
                                            "cclenter1") == 0)
                                                FATAL(
                        "out of space for character class [%.10s...] 2", p);
                                        *bp++ = ++c;
                                        i++;
                                c = chars[i-1];
                                while ((uchar)c < *p) { /* fails if *p is \\ */
                                        expand_buf(&chars, &bsize, i);
                                        chars[i++] = ++c;
                                }
                                p++;
                                continue;
                        }
                }
                if (!adjbuf(&buf, &bufsz, bp-buf+2, 100, &bp, "cclenter2"))
                        FATAL(
                "out of space for character class [%.10s...] 3", p);
                *bp++ = c;
                i++;
                expand_buf(&chars, &bsize, i);
                chars[i++] = c;
        }
        *bp = 0;
        dprintf(("cclenter: in = |%s|, out = |%s|\n", op, buf));
        chars[i++] = '\0';
        dprintf(("cclenter: in = |%s|, out = |%s|\n", op, chars));
        xfree(op);
        return (tostring(buf));
        ret = tostring(chars);
        free(chars);
        return (ret);
}

static void
overflo(const char *s)
```



```
overflo(char *s)
{
        FATAL("regular expression too big: %.30s...", gettext((char *)s));
        ERROR "regular expression too big: %s", gettext((char *)s) FATAL;
}

/* enter follow set of each leaf of vertex v into lfollow[leaf] */
static void
cfoll(fa *f, Node *v)
{
        int i;
        int *p;
        register int i;
        register int *p;

        switch (type(v)) {
        ELEAF
#endif /* ! codereview */
        LEAF
                f->re[info(v)].ltype = type(v);
                f->re[info(v)].lval.np = right(v);
                while (f->accept >= maxsetvec) {        /* guessing here! */
                        maxsetvec *= 4;
                        setvec = (int *)realloc(setvec, maxsetvec *
                            sizeof (int));
                        tmpset = (int *)realloc(tmpset, maxsetvec *
                            sizeof (int));
                        if (setvec == 0 || tmpset == 0)
                                overflo("out of space in cfoll()");
                }
                f->re[(int)left(v)].ltype = type(v);
                f->re[(int)left(v)].lval = (int)right(v);
                for (i = 0; i <= f->accept; i++)
                        setvec[i] = 0;
                setcnt = 0;
                follow(v);      /* computes setvec and setcnt */
                if ((p = (int *)calloc(1, (setcnt+1) * sizeof (int))) == NULL)
                        overflo("out of space building follow set");
                f->re[info(v)].lfollow = p;
                        overflo("follow set overflow");
                f->re[(int)left(v)].lfollow = p;
                *p = setcnt;
                for (i = f->accept; i >= 0; i--) {
                        if (setvec[i] == 1)
                                *++p = i;
                }
                break;
        UNARY
                cfoll(f, left(v));
                break;
        case CAT:
        case OR:
                cfoll(f, left(v));
                cfoll(f, right(v));
                break;
        default:        /* can't happen */
                FATAL("can't happen: unknown type %d in cfoll", type(v));
        default:
                ERROR "unknown type %d in cfoll", type(v) FATAL;
        }
}

/*
 * collects initially active leaves of p into setvec
 * returns 0 or 1 depending on whether p matches empty string
 */
```

```
455 static int
456 first(Node *p)
457 {
458         int b, lp;
312         register int b;

460         switch (type(p)) {
461         ELEAF
462 #endif /* ! codereview */
463         LEAF
464                 /* look for high-water mark of subscripts */
465                 lp = info(p);
466                 /* guessing here! */
467                 while (setcnt >= maxsetvec || lp >= maxsetvec) {
468                         maxsetvec *= 4;
469                         setvec = (int *)realloc(setvec, maxsetvec *
470                             sizeof (int));
471                         tmpset = (int *)realloc(tmpset, maxsetvec *
472                             sizeof (int));
473                         if (setvec == 0 || tmpset == 0)
474                                 overflo("out of space in first()");
475                 }
476                 if (type(p) == EMPTYRE) {
477                         setvec[lp] = 0;
478                         return (0);
479                 }
480                 if (setvec[lp] != 1) {
481                         setvec[lp] = 1;
315                 if (setvec[(int)left(p)] != 1) {
316                         setvec[(int)left(p)] = 1;
482                         setcnt++;
483                 }
484                 if (type(p) == CCL && (*(uchar *)right(p)) == '\0')
485                         return (0);             /* empty CCL */
486                 else
487                         return (1);
488         case PLUS:
489                 if (first(left(p)) == 0)
490                         return (0);
491                 return (1);
492         case STAR:
493         case QUEST:
494                 (void) first(left(p));
495                 return (0);
496         case CAT:
497                 if (first(left(p)) == 0 && first(right(p)) == 0)
498                         return (0);
499                 return (1);
500         case OR:
501                 b = first(right(p));
502                 if (first(left(p)) == 0 || b == 0)
503                         return (0);
504                 return (1);
505         }
506         /* can't happen */
507         FATAL("can't happen: unknown type %d in first", type(p));
508         /*NOTREACHED*/
341         ERROR "unknown type %d in first", type(p) FATAL;
509         return (-1);
510 }

512 /* collects leaves that can follow v into setvec */
513 static void
514 follow(Node *v)
515 {
516         Node *p;
```



```
518         if (type(v) == FINAL)
519                 return;
520         p = parent(v);
521         switch (type(p)) {
522         case STAR:
523         case PLUS:
524                 (void) first(v);
525                 follow(p);
526                 return;

528         case OR:
529         case QUEST:
530                 follow(p);
531                 return;

533         case CAT:
534                 if (v == left(p)) {     /* v is left child of p */
535                         if (first(right(p)) == 0) {
536                                 follow(p);
537                                 return;
538                         }
539                 } else          /* v is right child */
540                         follow(p);
541                 return;
375         default:
376                 ERROR "unknown type %d in follow", type(p) FATAL;
377                 break;
542         }
543 }

545 static int
546 member(int c, const uchar *sarg)        /* is c in s? */
382 member(uchar c, uchar *s)       /* is c in s? */
547 {
548         uchar *s = (uchar *)sarg;

550 #endif /* ! codereview */
551         while (*s)
552                 if (c == *s++)
553                         return (1);
554         return (0);
555 }


558 int
559 match(fa *f, const uchar *p0)   /* shortest match ? */
384 match(fa *f, uchar *p)
560 {
561         int s, ns;
562         uchar *p = (uchar *)p0;
386         register int s, ns;

564         s = f->reset ? makeinit(f, 0) : f->initstat;
565         if (f->out[s])
566                 return (1);
567         do {
568                 /* assert(*p < NCHARS); */
569 #endif /* ! codereview */
570                 if ((ns = f->gototab[s][*p]) != 0)
571                         s = ns;
572                 else
573                         s = cgoto(f, s, *p);
574                 if (f->out[s])
575                         return (1);
576         } while (*p++ != 0);
```

```
577         return (0);
578 }

580 int
581 pmatch(fa *f, const uchar *p0)          /* longest match, for sub */
392 pmatch(fa *f, uchar *p)
582 {
583         int s, ns;
584         uchar *p = (uchar *)p0;
585         uchar *q;
394         register int s, ns;
395         register uchar *q;
586         int i, k;

588         /* s = f->reset ? makeinit(f,1) : f->initstat; */
589 #endif /* ! codereview */
590         if (f->reset) {
591                 f->initstat = s = makeinit(f, 1);
592         } else {
593                 s = f->initstat;
594         }
595         patbeg = p;
596         patlen = -1;
597         do {
598                 q = p;
599                 do {
600                         if (f->out[s])          /* final state */
601                                 patlen = q-p;
602                         /* assert(*q < NCHARS); */
603 #endif /* ! codereview */
604                         if ((ns = f->gototab[s][*q]) != 0)
605                                 s = ns;
606                         else
607                                 s = cgoto(f, s, *q);
608                         if (s == 1) {   /* no transition */
609                                 if (patlen >= 0) {
610                                         patbeg = p;
611                                         return (1);
612                                 } else
613                                         goto nextin;    /* no match */
614                         }
615                 } while (*q++ != 0);
616                 if (f->out[s])
617                         patlen = q - p - 1;     /* don't count $ */
618                 if (patlen >= 0) {
619                         patbeg = p;
620                         return (1);
621                 }
622 nextin:
398         nextin:
623                 s = 2;
624                 if (f->reset) {
625                         for (i = 2; i <= f->curstat; i++)
626                                 xfree(f->posns[i]);
627                         k = *f->posns[0];
628                         if ((f->posns[2] =
629                             (int *)calloc(1, (k + 1) * sizeof (int))) == NULL) {
630                                 overflo("out of space in pmatch");
631                         }
632                         for (i = 0; i <= k; i++)
633                                 (f->posns[2])[i] = (f->posns[0])[i];
634                         f->initstat = f->curstat = 2;
635                         f->out[2] = f->out[0];
636                         for (i = 0; i < NCHARS; i++)
637                                 f->gototab[2][i] = 0;
638                 }
```



```
639         } while (*p++ != 0);
640         return (0);
641 }

643 int
644 nematch(fa *f, const uchar *p0)         /* non-empty match, for sub */
420 nematch(fa *f, uchar *p)
645 {
646         int s, ns;
647         uchar *p = (uchar *)p0;
648         uchar *q;
422         register int s, ns;
423         register uchar *q;
649         int i, k;

651         /* s = f->reset ? makeinit(f,1) : f->initstat; */
652 #endif /* ! codereview */
653         if (f->reset) {
654                 f->initstat = s = makeinit(f, 1);
655         } else {
656                 s = f->initstat;
657         }
658         patlen = -1;
659         while (*p) {
660                 q = p;
661                 do {
662                         if (f->out[s])          /* final state */
663                                 patlen = q - p;
664                         /* assert(*q < NCHARS); */
426                                 patlen = q-p;
665                         if ((ns = f->gototab[s][*q]) != 0)
666                                 s = ns;
667                         else
668                                 s = cgoto(f, s, *q);
669                         if (s == 1) {   /* no transition */
670                                 if (patlen > 0) {
671                                         patbeg = p;
672                                         return (1);
673                                 } else
674                                         goto nnextin;   /* no nonempty match */
675                         }
676                 } while (*q++ != 0);
677                 if (f->out[s])
678                         patlen = q - p - 1;     /* don't count $ */
679                 if (patlen > 0) {
680                         patbeg = p;
681                         return (1);
682                 }
683 nnextin:
445         nnextin:
684                 s = 2;
685                 if (f->reset) {
686                         for (i = 2; i <= f->curstat; i++)
687                                 xfree(f->posns[i]);
688                         k = *f->posns[0];
689                         if ((f->posns[2] =
690                             (int *)calloc(1, (k + 1) * sizeof (int))) == NULL) {
691                                 overflo("out of state space");
692                         }
693                         for (i = 0; i <= k; i++)
694                                 (f->posns[2])[i] = (f->posns[0])[i];
695                         f->initstat = f->curstat = 2;
696                         f->out[2] = f->out[0];
697                         for (i = 0; i < NCHARS; i++)
698                                 f->gototab[2][i] = 0;
699                 }
```

```
700                 p++;
701         }
702         return (0);
703 }

467 static Node *regexp(void), *primary(void), *concat(Node *);
468 static Node *alt(Node *), *unary(Node *);

705 static Node *
706 reparse(const uchar *p)
471 reparse(uchar *p)
707 {
708         /* parses regular expression pointed to by p */
709         /* uses relex() to scan regular expression */
710         Node *np;

712         dprintf(("reparse <%s>\n", p));
713         /* prestr points to string to be parsed */
714         lastre = prestr = (uchar *)p;
478         lastre = prestr = p;    /* prestr points to string to be parsed */
715         rtok = relex();
716         /* GNU compatibility: an empty regexp matches anything */
717         if (rtok == '\0') {
718                 /* FATAL("empty regular expression"); previous */
719                 return (op2(EMPTYRE, NIL, NIL));
720         }
480         if (rtok == '\0')
481                 ERROR "empty regular expression" FATAL;
721         np = regexp();
722         if (rtok != '\0') {
723                 FATAL("syntax error in regular expression %s at %s",
724                     lastre, prestr);
725         }
483         if (rtok == '\0') {
726         return (np);
485         } else {
486                 ERROR "syntax error in regular expression %s at %s",
487                     lastre, prestr FATAL;
488         }
489         /*NOTREACHED*/
490         return (NULL);
727 }

729 static Node *
730 regexp(void)    /* top-level parse of reg expr */
494 regexp(void)
731 {
732         return (alt(concat(primary())));
733 }

735 static Node *
736 primary(void)
737 {
738         Node *np;

740         switch (rtok) {
741         case CHAR:
742                 np = op2(CHAR, NIL, itonp(rlxval));
506                 np = op2(CHAR, NIL, (Node *)rlxval);
743                 rtok = relex();
744                 return (unary(np));
745         case ALL:
746                 rtok = relex();
747                 return (unary(op2(ALL, NIL, NIL)));
748         case EMPTYRE:
749                 rtok = relex();
```



```
750                 return (unary(op2(ALL, NIL, NIL)));
751 #endif /* ! codereview */
752         case DOT:
753                 rtok = relex();
754                 return (unary(op2(DOT, NIL, NIL)));
755         case CCL:
756                 /*LINTED align*/
757                 np = op2(CCL, NIL, (Node *)cclenter(rlxstr));
758                 rtok = relex();
759                 return (unary(np));
760         case NCCL:
761                 /*LINTED align*/
762                 np = op2(NCCL, NIL, (Node *)cclenter(rlxstr));
763                 rtok = relex();
764                 return (unary(np));
765         case '^':
766                 rtok = relex();
767                 return (unary(op2(CHAR, NIL, itonp(HAT))));
512                 return (unary(op2(CHAR, NIL, (Node *)HAT)));
768         case '$':
769                 rtok = relex();
770                 return (unary(op2(CHAR, NIL, NIL)));
771         case '(':
772                 rtok = relex();
773                 if (rtok == ')') {      /* special pleading for () */
774                         rtok = relex();
775                         return (unary(op2(CCL, NIL,
776                             /*LINTED align*/
777                             (Node *)tostring((uchar *)""))));
778                 }
779                 np = regexp();
780                 if (rtok == ')') {
781                         rtok = relex();
782                         return (unary(np));
783                 } else {
784                         FATAL("syntax error in regular expression %s at %s",
785                             lastre, prestr);
529                         ERROR "syntax error in regular expression %s at %s",
530                             lastre, prestr FATAL;
786                 }
787         default:
788                 FATAL("illegal primary in regular expression %s at %s",
789                     lastre, prestr);
533                 ERROR "illegal primary in regular expression %s at %s",
534                     lastre, prestr FATAL;
790         }
791         /*NOTREACHED*/
792         return (NULL);
793 }

795 static Node *
796 concat(Node *np)
797 {
798         switch (rtok) {
799         case CHAR: case DOT: case ALL: case EMPTYRE: case CCL: case NCCL:
800         case '$': case '(':
544         case CHAR: case DOT: case ALL: case CCL: case NCCL: case '$': case '(':
801                 return (concat(op2(CAT, np, primary())));
802         default:
803                 return (np);
804         }
805 }
______unchanged_portion_omitted_

835 /*
836  * Character class definitions conformant to the POSIX locale as
```

```
 * defined in IEEE P1003.1 draft 7 of June 2001, assuming the source
 * and operating character sets are both ASCII (ISO646) or supersets
 * thereof.
 *
 * Note that to avoid overflowing the temporary buffer used in
 * relex(), the expanded character class (prior to range expansion)
 * must be less than twice the size of their full name.
 */

/*
 * Because isblank doesn't show up in any of the header files on any
 * system i use, it's defined here.  if some other locale has a richer
 * definition of "blank", define HAS_ISBLANK and provide your own
 * version.
 * the parentheses here are an attempt to find a path through the maze
 * of macro definition and/or function and/or version provided.  thanks
 * to nelson beebe for the suggestion; let's see if it works everywhere.
 */

/* #define HAS_ISBLANK */
#ifndef HAS_ISBLANK

int
(xisblank)(int c)
{
        return (c == ' ' || c == '\t');
}

#endif

struct charclass {
        const char *cc_name;
        int cc_namelen;
        int (*cc_func)(int);
} charclasses[] = {
        { "alnum",      5,      isalnum },
        { "alpha",      5,      isalpha },
#ifndef HAS_ISBLANK
        { "blank",      5,      isspace }, /* was isblank */
#else
        { "blank",      5,      isblank },
#endif
        { "cntrl",      5,      iscntrl },
        { "digit",      5,      isdigit },
        { "graph",      5,      isgraph },
        { "lower",      5,      islower },
        { "print",      5,      isprint },
        { "punct",      5,      ispunct },
        { "space",      5,      isspace },
        { "upper",      5,      isupper },
        { "xdigit",     6,      isxdigit },
        { NULL,         0,      NULL },
};

#endif /* ! codereview */
static int
relex(void)             /* lexical analyzer for reparse */
{
        int c, n;
        int cflag;
        uchar *buf = 0;
        size_t bufsz = 100;
        uchar *bp;
        struct charclass *cc;
        int i;
        register int c;
```



```
        uchar *cbuf;
        int clen, cflag;

        switch (c = *prestr++) {
        case '|': return OR;
        case '*': return STAR;
        case '+': return PLUS;
        case '?': return QUEST;
        case '.': return DOT;
        case '\0': prestr--; return '\0';
        case '^':
        case '$':
        case '(':
        case ')':
                return (c);
        case '\\':
                rlxval = quoted(&prestr);
                if ((c = *prestr++) == 't')
                        c = '\t';
                else if (c == 'n')
                        c = '\n';
                else if (c == 'f')
                        c = '\f';
                else if (c == 'r')
                        c = '\r';
                else if (c == 'b')
                        c = '\b';
                else if (c == '\\')
                        c = '\\';
                else if (isdigit(c)) {
                        int n = c - '0';
                        if (isdigit(*prestr)) {
                                n = 8 * n + *prestr++ - '0';
                                if (isdigit(*prestr))
                                        n = 8 * n + *prestr++ - '0';
                        }
                        c = n;
                } /* else it's now in c */
                rlxval = c;
                return (CHAR);
        default:
                rlxval = c;
                return (CHAR);
        case '[':
                if (buf == 0 && (buf = (uchar *)malloc(bufsz)) == NULL)
                        FATAL("out of space in reg expr %.10s..", lastre);
                bp = buf;
                clen = 0;
                if (*prestr == '^') {
                        cflag = 1;
                        prestr++;
                } else
                        cflag = 0;
                n = 2 * strlen((const char *) prestr)+1;
                if (!adjbuf(&buf, &bufsz, n, n, &bp, "relex1"))
                        FATAL("out of space for reg expr %.10s...", lastre);
                init_buf(&cbuf, NULL, strlen((char *)prestr) * 2 + 1);
                for (;;) {
                        if ((c = *prestr++) == '\\') {
                                *bp++ = '\\';
                                cbuf[clen++] = '\\';
                                if ((c = *prestr++) == '\0') {
                                        FATAL(
                                "nonterminated character class %s", lastre);
                                        ERROR
                                "nonterminated character class %s", lastre FATAL;
```

```
939                                     }
940                                     *bp++ = c;
941                             } else if (c == '[' && *prestr == ':') {
942                                     /*
943                                      * POSIX char class names, Dag-Erling
944                                      * Smorgrav, des@ofug.org
945                                      */
946                                     for (cc = charclasses; cc->cc_name; cc++)
947                                             if (strncmp((const char *) prestr + 1,
948                                                 (const char *) cc->cc_name,
949                                                 cc->cc_namelen) == 0)
950                                                     break;
951                                     if (cc->cc_name != NULL &&
952                                         prestr[1 + cc->cc_namelen] == ':' &&
953                                         prestr[2 + cc->cc_namelen] == ']') {
954                                             prestr += cc->cc_namelen + 3;
955                                             for (i = 0; i < NCHARS; i++) {
956                                                     if (!adjbuf(&buf, &bufsz,
957                                                         bp - buf + 1, 100, &bp,
958                                                         "relex2"))
959                                                             FATAL(
960                                     "out of space for reg expr %.10s...", lastre);
961                                                     if (cc->cc_func(i)) {
962                                                             *bp++ = i;
963                                                             n++;
964                                                     }
965                                             }
966                                     } else
967                                             *bp++ = c;
968                             } else if (c == '\0') {
969                                     FATAL("nonterminated character class %.20s",
970                                         lastre);
971                             } else if (bp == buf) { /* 1st char is special */
972                                     *bp++ = c;
637                                     cbuf[clen++] = c;
973                             } else if (c == ']') {
974                                     *bp++ = 0;
975                                     rlxstr = tostring(buf);
639                                     cbuf[clen] = 0;
640                                     rlxstr = tostring(cbuf);
641                                     free(cbuf);
976                                     if (cflag == 0)
977                                             return (CCL);
978                                     else
979                                             return (NCCL);
646                             } else if (c == '\n') {
647                                     ERROR "newline in character class %s...",
648                                         lastre FATAL;
649                             } else if (c == '\0') {
650                                     ERROR "nonterminated character class %s",
651                                         lastre FATAL;
980                             } else
981                                     *bp++ = c;
653                                     cbuf[clen++] = c;
982                     }
983                     /*NOTREACHED*/
984             }
985             return (0);
986 }

988 static int
989 cgoto(fa *f, int s, int c)
990 {
991             int i, j, k;
992             int *p, *q;
663             register int i, j, k;
```



```
664             register int *p, *q;

994             assert(c == HAT || c < NCHARS);
995             while (f->accept >= maxsetvec) {        /* guessing here! */
996                     maxsetvec *= 4;
997                     setvec = (int *)realloc(setvec, maxsetvec * sizeof (int));
998                     tmpset = (int *)realloc(tmpset, maxsetvec * sizeof (int));
999                     if (setvec == 0 || tmpset == 0)
1000                            overflo("out of space in cgoto()");
1001            }
1002 #endif /* ! codereview */
1003            for (i = 0; i <= f->accept; i++)
1004                    setvec[i] = 0;
1005            setcnt = 0;
1006            /* compute positions of gototab[s,c] into setvec */
1007            p = f->posns[s];
1008            for (i = 1; i <= *p; i++) {
1009                    if ((k = f->re[p[i]].ltype) != FINAL) {
1010                            if ((k == CHAR && c == ptoi(f->re[p[i]].lval.np)) ||
1011                                (k == DOT && c != 0 && c != HAT) ||
1012                                (k == ALL && c != 0) ||
1013                                (k == EMPTYRE && c != 0) ||
1014                                (k == CCL &&
1015                                member(c, (uchar *)f->re[p[i]].lval.up)) ||
1016                                ((k == NCCL &&
1017                                !member(c, (uchar *)f->re[p[i]].lval.up)) &&
1018                                c != 0 && c != HAT)) {
666                             if (k == CHAR && c == f->re[p[i]].lval ||
667                                 k == DOT && c != 0 && c != HAT ||
668                                 k == ALL && c != 0 ||
669                                 k == CCL &&
670                                 member(c, (uchar *)f->re[p[i]].lval) ||
671                                 k == NCCL &&
672                                 !member(c, (uchar *)f->re[p[i]].lval) &&
673                                 c != 0 && c != HAT) {
1019                                    q = f->re[p[i]].lfollow;
1020                                    for (j = 1; j <= *q; j++) {
1021                                            if (q[j] >= maxsetvec) {
1022                                                    maxsetvec *= 4;
1023                                                    setvec = (int *)realloc(setvec,
1024                                                        maxsetvec * sizeof (int));
1025                                                    tmpset = (int *)realloc(tmpset,
1026                                                        maxsetvec * sizeof (int));
1027                                                    if (setvec == 0 ||
1028                                                        tmpset == 0)
1029                                                            overflo(
1030                                                    "cgoto overflow");
1031                                            }
1032 #endif /* ! codereview */
1033                                            if (setvec[q[j]] == 0) {
1034                                                    setcnt++;
1035                                                    setvec[q[j]] = 1;
1036                                            }
1037                                    }
1038                            }
1039                    }
1040            }
1041            /* determine if setvec is a previous state */
1042            tmpset[0] = setcnt;
1043            j = 1;
1044            for (i = f->accept; i >= 0; i--)
1045                    if (setvec[i]) {
1046                            tmpset[j++] = i;
1047                    }
1048            /* tmpset == previous state? */
1049            for (i = 1; i <= f->curstat; i++) {
```

```
1050                 p = f->posns[i];
1051                 if ((k = tmpset[0]) != p[0])
1052                         goto different;
1053                 for (j = 1; j <= k; j++)
1054                         if (tmpset[j] != p[j])
1055                                 goto different;
1056                 /* setvec is state i */
1057                 f->gototab[s][c] = i;
1058                 return (i);
1059         different:;
1060         }

1062         /* add tmpset to current set of states */
1063         if (f->curstat >= NSTATES - 1) {
 676         if (f->curstat >= NSTATES-1) {
1064                 f->curstat = 2;
1065                 f->reset = 1;
1066                 for (i = 2; i < NSTATES; i++)
1067                         xfree(f->posns[i]);
1068         } else
1069                 ++(f->curstat);
1070         for (i = 0; i < NCHARS; i++)
1071                 f->gototab[f->curstat][i] = 0;
1072         xfree(f->posns[f->curstat]);
1073         if ((p = (int *)calloc(1, (setcnt + 1) * sizeof (int))) == NULL)
1074                 overflo("out of space in cgoto");

1076         f->posns[f->curstat] = p;
1077         f->gototab[s][c] = f->curstat;
1078         for (i = 0; i <= setcnt; i++)
1079                 p[i] = tmpset[i];
1080         if (setvec[f->accept])
1081                 f->out[f->curstat] = 1;
1082         else
1083                 f->out[f->curstat] = 0;
1084         return (f->curstat);
1085 }

1087 static void
1088 freefa(fa *f)   /* free a finite automaton */
 701 freefa(fa *f)
1089 {
1090         int i;

 704         register int i;

1092         if (f == NULL)
1093                 return;
1094         for (i = 0; i <= f->curstat; i++)
1095                 xfree(f->posns[i]);
1096         for (i = 0; i <= f->accept; i++) {
 710         for (i = 0; i <= f->accept; i++)
1097                 xfree(f->re[i].lfollow);
1098                 if (f->re[i].ltype == CCL || f->re[i].ltype == NCCL)
1099                         xfree((f->re[i].lval.np));
1100         }
1101 #endif /* ! codereview */
1102         xfree(f->restr);
1103         xfree(f);
1104 }
```

```
**********************************************************
   13454 Sat May 11 15:20:44 2013
new/usr/src/cmd/awk/lex.c
3731 Update nawk to version 20121220
**********************************************************
```

```
/*
 * Copyright (C) Lucent Technologies 1997
 * All Rights Reserved
 *
 * Permission to use, copy, modify, and distribute this software and
 * its documentation for any purpose and without fee is hereby
 * granted, provided that the above copyright notice appear in all
 * copies and that both that the copyright notice and this
 * permission notice and warranty disclaimer appear in supporting
 * documentation, and that the name Lucent Technologies or any of
 * its entities not be used in advertising or publicity pertaining
 * to distribution of the software without specific, written prior
 * permission.
 *
 * LUCENT DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
 * INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS.
 * IN NO EVENT SHALL LUCENT OR ANY OF ITS ENTITIES BE LIABLE FOR ANY
 * SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
 * WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER
 * IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION,
 * ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF
 * THIS SOFTWARE.
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "awk.h"
#include "y.tab.h"

extern YYSTYPE  yylval;
extern int      infunc;

off_t   lineno  = 1;
int     bracecnt = 0;
int     brackcnt  = 0;
int     parencnt = 0;

typedef struct Keyword {
        const char *word;
        int     sub;
        int     type;
} Keyword;

Keyword keywords[] = {  /* keep sorted: binary searched */
        { "BEGIN",      XBEGIN,         XBEGIN },
        { "END",        XEND,           XEND },
        { "NF",         VARNF,          VARNF },
        { "atan2",      FATAN,          BLTIN },
        { "break",      BREAK,          BREAK },
        { "close",      CLOSE,          CLOSE },
        { "continue",   CONTINUE,       CONTINUE },
        { "cos",        FCOS,           BLTIN },
        { "delete",     DELETE,         DELETE },
        { "do",         DO,             DO },
        { "else",       ELSE,           ELSE },
        { "exit",       EXIT,           EXIT },
        { "exp",        FEXP,           BLTIN },
        { "fflush",     FFLUSH,         BLTIN },
        { "for",        FOR,            FOR },
```



```
        { "func",       FUNC,           FUNC },
        { "function",   FUNC,           FUNC },
        { "getline",    GETLINE,        GETLINE },
        { "gsub",       GSUB,           GSUB },
        { "if",         IF,             IF },
        { "in",         IN,             IN },
        { "index",      INDEX,          INDEX },
        { "int",        FINT,           BLTIN },
        { "length",     FLENGTH,        BLTIN },
        { "log",        FLOG,           BLTIN },
        { "match",      MATCHFCN,       MATCHFCN },
        { "next",       NEXT,           NEXT },
        { "nextfile",   NEXTFILE,       NEXTFILE },
        { "print",      PRINT,          PRINT },
        { "printf",     PRINTF,         PRINTF },
        { "rand",       FRAND,          BLTIN },
        { "return",     RETURN,         RETURN },
        { "sin",        FSIN,           BLTIN },
        { "split",      SPLIT,          SPLIT },
        { "sprintf",    SPRINTF,        SPRINTF },
        { "sqrt",       FSQRT,          BLTIN },
        { "srand",      FSRAND,         BLTIN },
        { "sub",        SUB,            SUB },
        { "substr",     SUBSTR,         SUBSTR },
        { "system",     FSYSTEM,        BLTIN },
        { "tolower",    FTOLOWER,       BLTIN },
        { "toupper",    FTOUPPER,       BLTIN },
        { "while",      WHILE,          WHILE },
};

#define RET(x)  { if (dbg) (void) printf("lex %s\n", tokname(x)); return (x); }

static int
peek(void)
{
        int c = input();
        unput(c);
        return (c);
}

static int
gettok(uchar **pbuf, int *psz)  /* get next input token */
{
        int c, retc;
        uchar *buf = *pbuf;
        size_t sz = *psz;
        uchar *bp = buf;

        c = input();
        if (c == 0)
                return (0);
        buf[0] = c;
        buf[1] = 0;
        if (!isalnum(c) && c != '.' && c != '_')
                return (c);

        *bp++ = c;
        if (isalpha(c) || c == '_') {   /* it's a varname */
                for (; (c = input()) != 0; ) {
                        if (bp-buf >= sz)
                                if (!adjbuf(&buf, &sz, bp - buf + 2, 100,
                                    &bp, "gettok"))
                                        FATAL(
                                "out of space for name %.10s...", buf);
                        if (isalnum(c) || c == '_')
                                *bp++ = c;
```

```
128                         else {
129                                 *bp = 0;
130                                 unput(c);
131                                 break;
132                         }
133                 }
134                 *bp = 0;
135                 retc = 'a';     /* alphanumeric */
136         } else {        /* maybe it's a number, but could be . */
137                 char *rem;
138                 /* read input until can't be a number */
139                 for (; (c = input()) != 0; ) {
140                         if (bp-buf >= sz)
141                                 if (!adjbuf(&buf, &sz, bp - buf + 2, 100,
142                                     &bp, "gettok"))
143                                         FATAL(
144                                 "out of space for number %.10s...", buf);
145                         if (isdigit(c) || c == 'e' || c == 'E' ||
146                             c == '.' || c == '+' || c == '-')
147                                 *bp++ = c;
148                         else {
149                                 unput(c);
150                                 break;
151                         }
152                 }
153                 *bp = 0;
154                 (void) strtod((char *)buf, &rem);       /* parse the number */
155                 /* it wasn't a valid number at all */
156                 if (rem == (char *)buf) {
157                         buf[1] = 0;     /* return one character as token */
158                         retc = buf[0];  /* character is its own type */
159                         unputstr(rem+1); /* put rest back for later */
160                 } else {        /* some prefix was a number */
161                         unputstr(rem);  /* put rest back for later */
162                         rem[0] = 0;     /* truncate buf after number part */
163                         retc = '0';     /* type is number */
164                 }
165         }
166         *pbuf = buf;
167         *psz = sz;
168         return (retc);
169 }

171 int     word(char *);
172 int     string(void);
173 int     regexpr(void);
174 int     sc      = 0;    /* 1 => return a } right now */
175 int     reg     = 0;    /* 1 => return a REGEXPR now */

177 int
178 yylex(void)
179 {
180         int c;
181         static uchar *buf = 0;
182         static int bufsize = 5; /* BUG: setting this small causes core dump! */

184         if (buf == 0 && (buf = (uchar *)malloc(bufsize)) == NULL)
185                 FATAL("out of space in yylex");
186         if (sc) {
187                 sc = 0;
188                 RET('}');
189         }
190         if (reg) {
191                 reg = 0;
192                 return (regexpr());
193         }
```



```
194         for (;;) {
195                 c = gettok(&buf, &bufsize);
196                 if (c == 0)
197                         return (0);
198                 if (isalpha(c) || c == '_')
199                         return (word((char *)buf));
200                 if (isdigit(c)) {
201                         yylval.cp = setsymtab(buf, tostring(buf),
202                             atof((char *)buf), CON|NUM, symtab);
203                         /* should this also have STR set? */
204                         RET(NUMBER);
205                 }

207                 yylval.i = c;
208                 switch (c) {
209                 case '\n':      /* {EOL} */
210                         RET(NL);
211                 case '\r':      /* assume \n is coming */
212                 case ' ':       /* {WS}+ */
213                 case '\t':
214                         break;
215                 case '#':       /* #.* strip comments */
216                         while ((c = input()) != '\n' && c != 0)
217                                 ;
218                         unput(c);
219                         break;
220                 case ';':
221                         RET(';');
222                 case '\\':
223                         if (peek() == '\n') {
224                                 (void) input();
225                         } else if (peek() == '\r') {
226                                 (void) input();
227                                 (void) input(); /* \n */
228                                 lineno++;
229                         } else {
230                                 RET(c);
231                         }
232                         break;
233                 case '&':
234                         if (peek() == '&') {
235                                 (void) input(); RET(AND);
236                         } else
237                                 RET('&');
238                 case '|':
239                         if (peek() == '|') {
240                                 (void) input(); RET(BOR);
241                         } else
242                                 RET('|');
243                 case '!':
244                         if (peek() == '=') {
245                                 (void) input(); yylval.i = NE; RET(NE);
246                         } else if (peek() == '~') {
247                                 (void) input(); yylval.i = NOTMATCH;
248                                 RET(MATCHOP);
249                         } else
250                                 RET(NOT);
251                 case '~':
252                         yylval.i = MATCH;
253                         RET(MATCHOP);
254                 case '<':
255                         if (peek() == '=') {
256                                 (void) input(); yylval.i = LE; RET(LE);
257                         } else {
258                                 yylval.i = LT; RET(LT);
259                         }
```

```
260                 case '=':
261                         if (peek() == '=') {
262                                 (void) input(); yylval.i = EQ; RET(EQ);
263                         } else {
264                                 yylval.i = ASSIGN; RET(ASGNOP);
265                         }
266                 case '>':
267                         if (peek() == '=') {
268                                 (void) input(); yylval.i = GE; RET(GE);
269                         } else if (peek() == '>') {
270                                 (void) input(); yylval.i = APPEND; RET(APPEND);
271                         } else {
272                                 yylval.i = GT; RET(GT);
273                         }
274                 case '+':
275                         if (peek() == '+') {
276                                 (void) input(); yylval.i = INCR; RET(INCR);
277                         } else if (peek() == '=') {
278                                 (void) input(); yylval.i = ADDEQ; RET(ASGNOP);
279                         } else
280                                 RET('+');
281                 case '-':
282                         if (peek() == '-') {
283                                 (void) input(); yylval.i = DECR; RET(DECR);
284                         } else if (peek() == '=') {
285                                 (void) input(); yylval.i = SUBEQ; RET(ASGNOP);
286                         } else
287                                 RET('-');
288                 case '*':
289                         if (peek() == '=') {    /* *= */
290                                 (void) input(); yylval.i = MULTEQ; RET(ASGNOP);
291                         } else if (peek() == '*') {     /* ** or **= */
292                                 (void) input(); /* eat 2nd * */
293                                 if (peek() == '=') {
294                                         (void) input(); yylval.i = POWEQ;
295                                         RET(ASGNOP);
296                                 } else {
297                                         RET(POWER);
298                                 }
299                         } else
300                                 RET('*');
301                 case '/':
302                         RET('/');
303                 case '%':
304                         if (peek() == '=') {
305                                 (void) input(); yylval.i = MODEQ; RET(ASGNOP);
306                         } else
307                                 RET('%');
308                 case '^':
309                         if (peek() == '=') {
310                                 (void) input(); yylval.i = POWEQ; RET(ASGNOP);
311                         } else
312                                 RET(POWER);

314                 case '$':
315                         /* BUG: awkward, if not wrong */
316                         c = gettok(&buf, &bufsize);
317                         if (isalpha(c)) {
318                                 /* very special */
319                                 if (strcmp((char *)buf, "NF") == 0) {
320                                         unputstr("(NF)");
321                                         RET(INDIRECT);
322                                 }
323                                 c = peek();
324                                 if (c == '(' || c == '[' ||
325                                     (infunc && isarg(buf) >= 0)) {
```



```
326                                         unputstr((char *)buf);
327                                         RET(INDIRECT);
328                                 }
329                                 yylval.cp = setsymtab(buf, (uchar *)"", 0.0,
330                                     STR | NUM, symtab);
331                                 RET(IVAR);
332                         } else if (c == 0) {    /*  */
333                                 SYNTAX("unexpected end of input after $");
334                                 RET(';');
335                         } else {
336                                 unputstr((char *)buf);
337                                 RET(INDIRECT);
338                         }

340                 case '}':
341                         if (--bracecnt < 0)
342                                 SYNTAX("extra }");
343                         sc = 1;
344                         RET(';');
345                 case ']':
346                         if (--brackcnt < 0)
347                                 SYNTAX("extra ]");
348                         RET(']');
349                 case ')':
350                         if (--parencnt < 0)
351                                 SYNTAX("extra )");
352                         RET(')');
353                 case '{':
354                         bracecnt++;
355                         RET('{');
356                 case '[':
357                         brackcnt++;
358                         RET('[');
359                 case '(':
360                         parencnt++;
361                         RET('(');

363                 case '"':
364                         /* BUG: should be like tran.c ? */
365                         return (string());

367                 default:
368                         RET(c);
369                 }
370         }
371 }

373 int
374 string(void)
375 {
376         int c, n;
377         uchar *s, *bp;
378         static uchar *buf = NULL;
379         static size_t bufsz = 500;

381         if (buf == 0 && (buf = (uchar *)malloc(bufsz)) == NULL)
382                 FATAL("out of space for strings");
383         for (bp = buf; (c = input()) != '"'; ) {
384                 if (!adjbuf(&buf, &bufsz, bp-buf+2, 500, &bp, "string"))
385                         FATAL("out of space for string %.10s...", buf);
386                 switch (c) {
387                 case '\n':
388                 case '\r':
389                 case 0:
390                         SYNTAX("non-terminated string %.10s...", buf);
391                         lineno++;
```

```
392                         if (c == 0)     /* hopeless */
393                                 FATAL("giving up");
394                         break;
395                 case '\\':
396                         c = input();
397                         switch (c) {
398                         case '"': *bp++ = '"'; break;
399                         case 'n': *bp++ = '\n'; break;
400                         case 't': *bp++ = '\t'; break;
401                         case 'f': *bp++ = '\f'; break;
402                         case 'r': *bp++ = '\r'; break;
403                         case 'b': *bp++ = '\b'; break;
404                         case 'v': *bp++ = '\v'; break;
405                         case 'a': *bp++ = '\007'; break;
406                         case '\\': *bp++ = '\\'; break;
407
408                         case '0': case '1': case '2': /* octal: \d \dd \ddd */
409                         case '3': case '4': case '5': case '6': case '7':
410                                 n = c - '0';
411                                 if ((c = peek()) >= '0' && c < '8') {
412                                         n = 8 * n + input() - '0';
413                                         if ((c = peek()) >= '0' && c < '8')
414                                                 n = 8 * n + input() - '0';
415                                 }
416                                 *bp++ = n;
417                                 break;
418
419                         case 'x': {     /* hex  \x0-9a-fA-F + */
420                                 char xbuf[100], *px;
421                 for (px = xbuf; (c = input()) != 0 && px - xbuf < 100 - 2; ) {
422                                         if (isdigit(c) ||
423                                             (c >= 'a' && c <= 'f') ||
424                                             (c >= 'A' && c <= 'F'))
425                                                 *px++ = c;
426                                         else
427                                                 break;
428                                 }
429                                 *px = 0;
430                                 unput(c);
431                                 (void) sscanf(xbuf, "%x", (unsigned int *)&n);
432                                 *bp++ = n;
433                                 break;
434                         }
435
436                         default:
437                                 *bp++ = c;
438                                 break;
439                         }
440                         break;
441                 default:
442                         *bp++ = c;
443                         break;
444                 }
445         }
446         *bp = 0;
447         s = tostring(buf);
448         *bp++ = ' '; *bp++ = 0;
449         yylval.cp = setsymtab(buf, s, 0.0, CON|STR|DONTFREE, symtab);
450         RET(STRING);
451 }


454 int
455 binsearch(char *w, Keyword *kp, int n)
456 {
457         int cond, low, mid, high;
```



```
459         low = 0;
460         high = n - 1;
461         while (low <= high) {
462                 mid = (low + high) / 2;
463                 if ((cond = strcmp(w, kp[mid].word)) < 0)
464                         high = mid - 1;
465                 else if (cond > 0)
466                         low = mid + 1;
467                 else
468                         return (mid);
469         }
470         return (-1);
471 }

473 int
474 word(char *w)
475 {
476         Keyword *kp;
477         int c, n;

479         n = binsearch(w, keywords, sizeof (keywords) / sizeof (keywords[0]));
480         /*
481          * BUG: this ought to be inside the if;
482          * in theory could fault (daniel barrett)
483          */
484         kp = keywords + n;
485         if (n != -1) {  /* found in table */
486                 yylval.i = kp->sub;
487                 switch (kp->type) {     /* special handling */
488                 case BLTIN:
489                         if (kp->sub == FSYSTEM && safe)
490                                 SYNTAX("system is unsafe");
491                         RET(kp->type);
492                 case FUNC:
493                         if (infunc)
494                                 SYNTAX("illegal nested function");
495                         RET(kp->type);
496                 case RETURN:
497                         if (!infunc)
498                                 SYNTAX("return not in function");
499                         RET(kp->type);
500                 case VARNF:
501                         yylval.cp = setsymtab((uchar *)"NF", (uchar *)"", 0.0,
502                                 NUM, symtab);
503                         RET(VARNF);
504                 default:
505                         RET(kp->type);
506                 }
507         }
508         c = peek();     /* look for '(' */
509         if (c != '(' && infunc && (n = isarg((uchar *)w)) >= 0) {
510                 yylval.i = n;
511                 RET(ARG);
512         } else {
513                 yylval.cp = setsymtab((uchar *)w, (uchar *)"", 0.0,
514                   STR | NUM | DONTFREE, symtab);
515                 if (c == '(') {
516                         RET(CALL);
517                 } else {
518                         RET(VAR);
519                 }
520         }
521 }

523 void
```

```
startreg(void)  /* next call to yylex will return a regular expression */
{
        reg = 1;
}

int
regexpr(void)
{
        int c;
        static uchar *buf = NULL;
        static size_t bufsz = 500;
        uchar *bp;

        if (buf == 0 && (buf = (uchar *)malloc(bufsz)) == NULL)
                FATAL("out of space for rex expr");
        bp = buf;
        for (; (c = input()) != '/' && c != 0; ) {
                if (!adjbuf(&buf, &bufsz, bp-buf+3, 500, &bp, "regexpr"))
                        FATAL("out of space for reg expr %.10s...", buf);
                if (c == '\n') {
                        SYNTAX("newline in regular expression %.10s...", buf);
                        unput('\n');
                        break;
                } else if (c == '\\') {
                        *bp++ = '\\';
                        *bp++ = input();
                } else {
                        *bp++ = c;
                }
        }
        *bp = 0;
        if (c == 0)
                SYNTAX("non-terminated regular expression %.10s...", buf);
        yylval.s = tostring(buf);
        unput('/');
        RET(REGEXPR);
}

/* low-level lexical stuff, sort of inherited from lex */

char    ebuf[300];
char    *ep = ebuf;
char    yysbuf[100];    /* pushback buffer */
char    *yysptr = yysbuf;
FILE    *yyin = 0;

int
input(void)     /* get next lexical input character */
{
        int c;
        extern uchar *lexprog;

        if (yysptr > yysbuf)
                c = (uchar)*--yysptr;
        else if (lexprog != NULL) {     /* awk '...' */
                if ((c = (uchar)*lexprog) != 0)
                        lexprog++;
        } else                          /* awk -f ... */
                c = pgetc();
        if (c == '\n')
                lineno++;
        else if (c == EOF)
                c = 0;
        if (ep >= ebuf + sizeof (ebuf))
                ep = ebuf;
        return (*ep++ = c);
```



```
}

void
unput(int c)    /* put lexical character back on input */
{
        if (c == '\n')
                lineno--;
        if (yysptr >= yysbuf + sizeof (yysbuf))
                FATAL("pushed back too much: %.20s...", yysbuf);
        *yysptr++ = c;
        if (--ep < ebuf)
                ep = ebuf + sizeof (ebuf) - 1;
}

void
unputstr(const char *s) /* put a string back on input */
{
        int i;

        for (i = strlen(s)-1; i >= 0; i--)
                unput(s[i]);
}
#endif /* ! codereview */
```

```
**********************************************************
   19425 Sat May 11 15:20:44 2013
new/usr/src/cmd/awk/lib.c
3731 Update nawk to version 20121220
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright 2006 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * Copyright (C) Lucent Technologies 1997
 * All Rights Reserved
 *
 * Permission to use, copy, modify, and distribute this software and
 * its documentation for any purpose and without fee is hereby
 * granted, provided that the above copyright notice appear in all
 * copies and that both that the copyright notice and this
 * permission notice and warranty disclaimer appear in supporting
 * documentation, and that the name Lucent Technologies or any of
 * its entities not be used in advertising or publicity pertaining
 * to distribution of the software without specific, written prior
 * permission.
 *
 * LUCENT DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
 * INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS.
 * IN NO EVENT SHALL LUCENT OR ANY OF ITS ENTITIES BE LIABLE FOR ANY
 * SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
 * WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER
 * IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION,
 * ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF
 * THIS SOFTWARE.
 */
/*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
/*        All Rights Reserved   */

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

#include <errno.h>
#include <stdarg.h>
#include <math.h>
#endif /* ! codereview */
#include "awk.h"
#include "y.tab.h"
```



```
uchar   *record;
size_t  record_size = RECSIZE;

Cell    **fldtab;       /* pointers to Cells */
char    inputFS[100] = " ";

#define MAXFLD  2
int     nfields = MAXFLD;       /* last allocated slot for $i */
size_t  record_size;

int     donefld;        /* 1 = implies rec broken into fields */
int     donerec;        /* 1 = record is valid (no flds have changed) */

int     lastfld = 0;    /* last used field */
static struct fldtab_chunk {
        struct fldtab_chunk     *next;
        Cell                    fields[FLD_INCR];
} *fldtab_head, *fldtab_tail;

static  size_t  fldtab_maxidx;

static FILE     *infile = NULL;
static uchar    *file   = (uchar*) "";
static uchar    *fields;
static size_t   fields_size = RECSIZE;
static size_t   fields_size = LINE_INCR;

static int      maxfld  = 0;    /* last used field */
static int      argno   = 1;    /* current input argument number */

static  uchar   *getargv(int);
static  void    cleanfld(int, int);
static  int     refldbld(uchar *, uchar *);
static  void    bcheck2(int, int, int);
static  void    eprint(void);
static  void    bclass(int);
static  void    makefields(int, int);


static Cell dollar0 = { OCELL, CFLD, NULL, (uchar *)"", 0.0, REC|STR|DONTFREE };
static Cell dollar1 = { OCELL, CFLD, NULL, (uchar *)"", 0.0, FLD|STR|DONTFREE };

void
recinit(unsigned int n)
{
        if ((record = (uchar *)malloc(n)) == NULL ||
            (fields = (uchar *)malloc(n+1)) == NULL ||
            (fldtab = (Cell **)malloc((nfields + 1) *
            sizeof (Cell *))) == NULL ||
            (fldtab[0] = (Cell *)malloc(sizeof (Cell))) == NULL)
                FATAL("out of space for $0 and fields");
        *fldtab[0] = dollar0;
        fldtab[0]->sval = record;
        fldtab[0]->nval = tostring((uchar *)"0");
        makefields(1, nfields);
}

static void
makefields(int n1, int n2)      /* create $n1..$n2 inclusive */
{
        char temp[50];
        int i;

        for (i = n1; i <= n2; i++) {
                fldtab[i] = (Cell *)malloc(sizeof (struct Cell));
                if (fldtab[i] == NULL)
```

```
115                         FATAL("out of space in makefields %d", i);
116                 *fldtab[i] = dollar1;
117                 (void) sprintf(temp, "%d", i);
118                 fldtab[i]->nval = tostring((uchar *)temp);
119         }
120 }
121 #endif /* ! codereview */

123 static void
124 initgetrec(void)
125 {
126         int i;
127         uchar *p;

129         for (i = 1; i < *ARGC; i++) {
130                 p = getargv(i); /* find 1st real filename */
131                 if (p == NULL || *p == '\0') {  /* deleted or zapped */
132                         argno++;
133                         continue;
134                 }
135                 if (!isclvar(p)) {
136                         (void) setsval(lookup((uchar *)"FILENAME", symtab), p);
 59                 if (!isclvar(p = getargv(i)))   /* find 1st real filename */
137                         return;
138                 }
139 #endif /* ! codereview */
140                 setclvar(p);    /* a commandline assignment before filename */
141                 argno++;
142         }
143         infile = stdin;         /* no filenames, so use stdin */
 61         /* *FILENAME = file = (uchar*) "-"; */
144 }

146 static int firsttime = 1;

148 #endif /* ! codereview */
149 int
150 getrec(uchar **pbuf, size_t *pbufsize, int isrecord)
151 {                               /* get next input record */
152                                 /* note: cares whether buf == record */
 64 getrec(uchar **bufp, size_t *bufsizep)
 65 {
153         int c;
154         uchar_t *buf = *pbuf;
155         uchar saveb0;
156         size_t bufsize = *pbufsize, savebufsize = bufsize;
 67         static int firsttime = 1;
 68         uchar_t *buf, *nbuf;
 69         size_t  len;

158         if (firsttime) {
159                 firsttime = 0;
160                 initgetrec();
161         }
162         dprintf(("RS=<%s>, FS=<%s>, ARGC=%f, FILENAME=%s\n",
163             *RS, *FS, *ARGC, *FILENAME));
164         if (isrecord) {
165 #endif /* ! codereview */
166                 donefld = 0;
167                 donerec = 1;
168         }
169         saveb0 = buf[0];
170         buf[0] = 0;
171 #endif /* ! codereview */
172         while (argno < *ARGC || infile == stdin) {
173                 dprintf(("argno=%d, file=|%s|\n", argno, file));
```



```
174                 if (infile == NULL) {   /* have to open a new file */
175                         file = getargv(argno);
176                         /* deleted or zapped */
177                         if (file == NULL || *file == '\0') {
 77                         if (*file == '\0') {    /* it's been zapped */
178                                 argno++;
179                                 continue;
180                         }
181                         if (isclvar(file)) {    /* a var=value arg */
182                                 setclvar(file);
183                                 argno++;
184                                 continue;
185                         }
186                         *FILENAME = file;
187                         dprintf(("opening file %s\n", file));
188                         if (*file == '-' && *(file+1) == '\0')
189                                 infile = stdin;
190                         else if ((infile = fopen((char *)file, "r")) == NULL)
191                                 FATAL("can't open file %s", file);
 91                                 ERROR "can't open file %s", file FATAL;
192                         (void) setfval(fnrloc, 0.0);
193                 }
194                 c = readrec(&buf, &bufsize, infile);
 94                 c = readrec(&nbuf, &len, infile);
 95                 expand_buf(bufp, bufsizep, len);
 96                 buf = *bufp;
 97                 (void) memcpy(buf, nbuf, len);
 98                 buf[len] = '\0';
 99                 free(nbuf);

195                 if (c != 0 || buf[0] != '\0') { /* normal record */
196                         if (isrecord) {
197                                 if (freeable(fldtab[0]))
198                                         xfree(fldtab[0]->sval);
199                                 fldtab[0]->sval = buf;  /* buf == record */
200                                 fldtab[0]->tval = REC | STR | DONTFREE;
201                                 if (is_number(fldtab[0]->sval)) {
202                                         fldtab[0]->fval =
203                                             atof((const char *)fldtab[0]->sval);
204                                         fldtab[0]->tval |= NUM;
102                         if (bufp == &record) {
103                                 if (!(recloc->tval & DONTFREE))
104                                         xfree(recloc->sval);
105                                 recloc->sval = record;
106                                 recloc->tval = REC | STR | DONTFREE;
107                                 if (is_number(recloc->sval)) {
108                                         recloc->fval =
109                                             atof((const char *)recloc->sval);
110                                         recloc->tval |= NUM;
205                                 }
206                         }
207                         (void) setfval(nrloc, nrloc->fval+1);
208                         (void) setfval(fnrloc, fnrloc->fval+1);
209                         *pbuf = buf;
210                         *pbufsize = bufsize;
211 #endif /* ! codereview */
212                         return (1);
213                 }
214                 /* EOF arrived on this file; set up next */
215                 if (infile != stdin)
216                         (void) fclose(infile);
217                 infile = NULL;
218                 argno++;
219         }
220         buf[0] = saveb0;
221         *pbuf = buf;
```

```
222         *pbufsize = savebufsize;
223 #endif /* ! codereview */
224         return (0);     /* true end of file */
225 }
226
227 void
228 nextfile(void)
229 {
230         if (infile != NULL && infile != stdin)
231                 (void) fclose(infile);
232         infile = NULL;
233         argno++;
234 }
235
236 /*
237  * read one record into buf
238  */
239 #endif /* ! codereview */
240 int
241 readrec(uchar **pbuf, size_t *pbufsize, FILE *inf)
115 readrec(uchar **bufp, size_t *sizep, FILE *inf) /* read one record into buf */
242 {
243         int sep, c;
244         uchar *rr, *buf = *pbuf;
245         size_t bufsize = *pbufsize;
246
247         if (strlen((char *)*FS) >= sizeof (inputFS))
248                 FATAL("field separator %.10s... is too long", *FS);
249         /*
250          * fflush(stdout); avoids some buffering problem
251          * but makes it 25% slower
252          */
118         uchar   *buf;
119         int     count;
120         size_t  bufsize;
121
254         /* for subsequent field splitting */
255         (void) strcpy(inputFS, (char *)*FS);
122         init_buf(&buf, &bufsize, LINE_INCR);
256         if ((sep = **RS) == 0) {
257                 sep = '\n';
258                 /* skip leading \n's */
259                 while ((c = getc(inf)) == '\n' && c != EOF)
260                         ;
261                 if (c != EOF)
262                         (void) ungetc(c, inf);
263         }
264         for (rr = buf; ; ) {
265                 for (; (c = getc(inf)) != sep && c != EOF; ) {
266                         if (rr-buf+1 > bufsize)
267                                 if (!adjbuf(&buf, &bufsize, 1+rr-buf,
268                                     record_size, &rr, "readrec 1"))
269                                         FATAL(
270                                 "input record '%.30s...' too long", buf);
271                         *rr++ = c;
131         count = 0;
132         for (;;) {
133                 while ((c = getc(inf)) != sep && c != EOF) {
134                         expand_buf(&buf, &bufsize, count);
135                         buf[count++] = c;
272                 }
273                 if (**RS == sep || c == EOF)
274                         break;
275                 if ((c = getc(inf)) == '\n' || c == EOF) /* 2 in a row */
276                         break;
277                 if (!adjbuf(&buf, &bufsize, 2+rr-buf, record_size, &rr,
```



```
278                     "readrec 2"))
279                         FATAL("input record '%.30s...' too long", buf);
280                 *rr++ = '\n';
281                 *rr++ = c;
282         }
283         if (!adjbuf(&buf, &bufsize, 1+rr-buf, record_size, &rr, "readrec 3"))
284                 FATAL("input record '%.30s...' too long", buf);
285         *rr = 0;
141                 expand_buf(&buf, &bufsize, count + 1);
142                 buf[count++] = '\n';
143                 buf[count++] = c;
144         }
145         buf[count] = '\0';
286         dprintf(("readrec saw <%s>, returns %d\n",
287             buf, c == EOF && rr == buf ? 0 : 1));
288         *pbuf = buf;
289         *pbufsize = bufsize;
290         return (c == EOF && rr == buf ? 0 : 1);
147             buf, c == EOF && count == 0 ? 0 : 1));
148         *bufp = buf;
149         *sizep = count;
150         return (c == EOF && count == 0 ? 0 : 1);
291 }
292
153 /* get ARGV[n] */
293 static uchar *
294 getargv(int n)          /* get ARGV[n] */
155 getargv(int n)
295 {
296         Cell *x;
297         uchar *s, temp[50];
158         uchar *s, temp[11];
298         extern Array *ARGVtab;
299
300         (void) sprintf((char *)temp, "%d", n);
301         if (lookup(temp, ARGVtab) == NULL)
302                 return (NULL);
303 #endif /* ! codereview */
304         x = setsymtab(temp, (uchar *)"", 0.0, STR, ARGVtab);
305         s = getsval(x);
306         dprintf(("getargv(%d) returns |%s|\n", n, s));
307         return (s);
308 }
309
310 void
311 setclvar(uchar *s)      /* set var=value from s */
312 {
313         uchar *p;
314         Cell *q;
315
316         for (p = s; *p != '='; p++)
317                 ;
318         *p++ = 0;
319         p = qstring(p, '\0');
320         q = setsymtab(s, p, 0.0, STR, symtab);
321         (void) setsval(q, p);
322         if (is_number(q->sval)) {
323                 q->fval = atof((const char *)q->sval);
324                 q->tval |= NUM;
325         }
326         dprintf(("command line set %s to |%s|\n", s, p));
162         free(p);
327 }
328
329 void
330 fldbld(void)    /* create fields from current record */
```

```
166 fldbld(void)
331 {
332         /* this relies on having fields[] the same length as $0 */
333         /* the fields are all stored in this one array with \0's */
334         /* possibly with a final trailing \0 not associated with any field */
335 #endif /* ! codereview */
336         uchar *r, *fr, sep;
337         Cell *p;
338         int i, j;
339         size_t n;
168         int i;
169         size_t  len;

341         if (donefld)
342                 return;
343         if (!isstr(fldtab[0]))
344                 (void) getsval(fldtab[0]);
345         r = fldtab[0]->sval;
346         n = strlen((char *)r);
347         if (n > fields_size) {
348                 xfree(fields);
349                 /* possibly 2 final \0s */
350                 if ((fields = (uchar *)malloc(n + 2)) == NULL)
351                         FATAL("out of space for fields in fldbld %d", n);
352                 fields_size = n;
353         }
173         if (!(recloc->tval & STR))
174                 (void) getsval(recloc);
175         r = recloc->sval;       /* was record! */

177         /* make sure fields is always allocated */
178         adjust_buf(&fields, fields_size);

180         /*
181          * make sure fields has enough size. We don't expand the buffer
182          * in the middle of the loop, since p->sval has already pointed
183          * the address in the fields.
184          */
185         len = strlen((char *)r) + 1;
186         expand_buf(&fields, &fields_size, len);
354         fr = fields;

355         i = 0;  /* number of fields accumulated here */
356         (void) strcpy(inputFS, (char *)*FS);
357         if (strlen(inputFS) > 1) {      /* it's a regular expression */
358                 i = refldbld(r, (uchar *)inputFS);
359         } else if ((sep = *inputFS) == ' ') {   /* default whitespace */
190         if (strlen((char *)*FS) > 1) {  /* it's a regular expression */
191                 i = refldbld(r, *FS);
192         } else if ((sep = **FS) == ' ') {
360                 for (i = 0; ; ) {
361                         while (*r == ' ' || *r == '\t' || *r == '\n')
362                                 r++;
363                         if (*r == 0)
364                                 break;
365                         i++;
366                         if (i > nfields)
367                                 growfldtab(i);
368                         if (freeable(fldtab[i]))
369                                 xfree(fldtab[i]->sval);
370                         fldtab[i]->sval = fr;
371                         fldtab[i]->tval = FLD | STR | DONTFREE;
199                         p = getfld(i);
200                         if (!(p->tval & DONTFREE))
201                                 xfree(p->sval);
202                         p->sval = fr;
```



```
203                         p->tval = FLD | STR | DONTFREE;
372                         do
373                                 *fr++ = *r++;
374                         while (*r != ' ' && *r != '\t' && *r != '\n' &&
375                             *r != '\0')
376                                 ;
377                         *fr++ = 0;
378                 }
379                 *fr = 0;
380         } else if ((sep = *inputFS) == 0) {     /* new: FS="" => 1 char/field */
381                 for (i = 0; *r != 0; r++) {
382                         uchar buf[2];
383                         i++;
384                         if (i > nfields)
385                                 growfldtab(i);
386                         if (freeable(fldtab[i]))
387                                 xfree(fldtab[i]->sval);
388                         buf[0] = *r;
389                         buf[1] = 0;
390                         fldtab[i]->sval = tostring(buf);
391                         fldtab[i]->tval = FLD | STR;
392                 }
393                 *fr = 0;
394 #endif /* ! codereview */
395         } else if (*r != 0) {   /* if 0, it's a null field */
396                 /*
397                  * subtlecase : if length(FS) == 1 && length(RS > 0)
398                  * \n is NOT a field separator (cf awk book 61,84).
399                  * this variable is tested in the inner while loop.
400                  */
401                 int rtest = '\n';  /* normal case */
402                 if (strlen((char *)*RS) > 0)
403                         rtest = '\0';
404 #endif /* ! codereview */
405                 for (;;) {
406                         i++;
407                         if (i > nfields)
408                                 growfldtab(i);
409                         if (freeable(fldtab[i]))
410                                 xfree(fldtab[i]->sval);
411                         fldtab[i]->sval = fr;
412                         fldtab[i]->tval = FLD | STR | DONTFREE;
413                         /* \n is always a separator */
414                         while (*r != sep && *r != rtest && *r != '\0')
212                         p = getfld(i);
213                         if (!(p->tval & DONTFREE))
214                                 xfree(p->sval);
215                         p->sval = fr;
216                         p->tval = FLD | STR | DONTFREE;
217                         /* \n always a separator */
218                         while (*r != sep && *r != '\n' && *r != '\0')
415                                 *fr++ = *r++;
416                         *fr++ = 0;
417                         if (*r++ == 0)
418                                 break;
419                 }
420                 *fr = 0;
421         }
422         if (i > nfields)
423                 FATAL("record '%.30s...' has too many fields; can't happen", r);
424 #endif /* ! codereview */
425         /* clean out junk from previous record */
426         cleanfld(i + 1, lastfld);
427         lastfld = i;
226         cleanfld(i, maxfld);
227         maxfld = i;
```

```
428         donefld = 1;
429         for (j = 1; j <= lastfld; j++) {
430                 p = fldtab[j];
229         for (i = 1; i <= maxfld; i++) {
230                 p = getfld(i);
431                 if (is_number(p->sval)) {
432                         p->fval = atof((const char *)p->sval);
433                         p->tval |= NUM;
434                 }
435         }
436         (void) setfval(nfloc, (Awkfloat)lastfld);

237         (void) setfval(nfloc, (Awkfloat) maxfld);
437         if (dbg) {
438                 for (j = 0; j <= lastfld; j++) {
439                         p = fldtab[j];
440                         (void) printf("field %d (%s): |%s|\n", j, p->nval,
441                             p->sval);
239                 for (i = 0; i <= maxfld; i++) {
240                         p = getfld(i);
241                         (void) printf("field %d: |%s|\n", i, p->sval);
442                 }
443         }
444 }

446 static void
447 cleanfld(int n1, int n2)        /* clean out fields n1..n2 inclusive */
448 {                               /* nvals remain intact */
449         static uchar *nullstat = (uchar *)"";
248 {
249         static uchar *nullstat = (uchar *) "";
450         Cell *p;
451         int i;

453         for (i = n1; i <= n2; i++) {
454                 p = fldtab[i];
455                 if (freeable(p))
253         for (i = n2; i > n1; i--) {
254                 p = getfld(i);
255                 if (!(p->tval & DONTFREE))
456                         xfree(p->sval);
457                 p->sval = nullstat;
458 #endif /* ! codereview */
459                 p->tval = FLD | STR | DONTFREE;
257                 p->sval = nullstat;
460         }
461 }

463 void
464 newfld(int n)   /* add field n after end of existing lastfld */
262 newfld(int n)   /* add field n (after end) */
263 {
264         if (n < 0)
265                 ERROR "accessing invalid field", record FATAL;
266         (void) getfld(n);
267         cleanfld(maxfld, n);
268         maxfld = n;
269         (void) setfval(nfloc, (Awkfloat) n);
270 }

272 /*
273  * allocate field table. We don't reallocate the table since there
274  * might be somewhere recording the address of the table.
275  */
276 static void
277 morefld(void)
```



```
465 {
466         if (n > nfields)
467                 growfldtab(n);
468         cleanfld(lastfld + 1, n);
469         lastfld = n;
470         (void) setfval(nfloc, (Awkfloat)n);
279         int     i;
280         struct fldtab_chunk *fldcp;
281         Cell    *newfld;

283         if ((fldcp = calloc(sizeof (struct fldtab_chunk), 1)) == NULL)
284                 ERROR "out of space in morefld" FATAL;

286         newfld = &fldcp->fields[0];
287         for (i = 0; i < FLD_INCR; i++) {
288                 newfld[i].ctype = OCELL;
289                 newfld[i].csub = CFLD;
290                 newfld[i].nval = NULL;
291                 newfld[i].sval = (uchar *)"";
292                 newfld[i].fval = 0.0;
293                 newfld[i].tval = FLD|STR|DONTFREE;
294                 newfld[i].cnext = NULL;
295         }
296         /*
297          * link this field chunk
298          */
299         if (fldtab_head == NULL)
300                 fldtab_head = fldcp;
301         else
302                 fldtab_tail->next = fldcp;
303         fldtab_tail = fldcp;
304         fldcp->next = NULL;

306         fldtab_maxidx += FLD_INCR;
471 }

473 Cell *
474 fieldadr(int n)         /* get nth field */
310 getfld(int idx)
475 {
476         if (n < 0)
477                 FATAL("trying to access out of range field %d", n);
478         if (n > nfields)        /* fields after NF are empty */
479                 growfldtab(n);  /* but does not increase NF */
480         return (fldtab[n]);
312         struct fldtab_chunk *fldcp;
313         int     cbase;

315         if (idx < 0)
316                 ERROR "trying to access field %d", idx FATAL;
317         while (idx >= fldtab_maxidx)
318                 morefld();
319         cbase = 0;
320         for (fldcp = fldtab_head; fldcp != NULL; fldcp = fldcp->next) {
321                 if (idx < (cbase + FLD_INCR))
322                         return (&fldcp->fields[idx - cbase]);
323                 cbase += FLD_INCR;
324         }
325         /* should never happen */
326         ERROR "trying to access invalid field %d", idx FATAL;
327         return (NULL);
481 }

483 void
484 growfldtab(int n)       /* make new fields up to at least $n */
485 {
```

```
        int nf = 2 * nfields;
        size_t s;

        if (n > nf)
                nf = n;
        /* freebsd: how much do we need? */
        s = (nf + 1) * (sizeof (struct Cell *));
        if (s / sizeof (struct Cell *) - 1 == nf)       /* didn't overflow */
                fldtab = (Cell **) realloc(fldtab, s);
        else    /* overflow sizeof int */
                xfree(fldtab);  /* make it null */
        if (fldtab == NULL)
                FATAL("out of space creating %d fields", nf);
        makefields(nfields + 1, nf);
        nfields = nf;
int
fldidx(Cell *vp)
{
        struct fldtab_chunk *fldcp;
        Cell    *tbl;
        int     cbase;

        cbase = 0;
        for (fldcp = fldtab_head; fldcp != NULL; fldcp = fldcp->next) {
                tbl = &fldcp->fields[0];
                if (vp >= tbl && vp < (tbl + FLD_INCR))
                        return (cbase + (vp - tbl));
                cbase += FLD_INCR;
        }
        /* should never happen */
        ERROR "trying to access unknown field" FATAL;
        return (0);
}

static int
refldbld(uchar *rec, uchar *fs) /* build fields from reg expr in FS */
{
        /* this relies on having fields[] the same length as $0 */
        /* the fields are all stored in this one array with \0's */
#endif /* ! codereview */
        uchar *fr;
        int i, tempstat;
        fa *pfa;
        size_t n;
        Cell    *p;
        size_t  len;

        n = strlen((char *)rec);
        if (n > fields_size) {
                xfree(fields);
                if ((fields = (uchar *)malloc(n + 1)) == NULL)
                        FATAL("out of space for fields in refldbld %d", n);
                fields_size = n;
        }
        /* make sure fields is allocated */
        adjust_buf(&fields, fields_size);
        fr = fields;
        *fr = '\0';
        if (*rec == '\0')
                return (0);

        len = strlen((char *)rec) + 1;
        expand_buf(&fields, &fields_size, len);
        fr = fields;

        pfa = makedfa(fs, 1);
```



```
        dprintf(("into refldbld, rec = <%s>, pat = <%s>\n", rec, fs));
        tempstat = pfa->initstat;
        for (i = 1; ; i++) {
                if (i > nfields)
                        growfldtab(i);
                if (freeable(fldtab[i]))
                        xfree(fldtab[i]->sval);
                fldtab[i]->tval = FLD | STR | DONTFREE;
                fldtab[i]->sval = fr;
                p = getfld(i);
                if (!(p->tval & DONTFREE))
                        xfree(p->sval);
                p->tval = FLD | STR | DONTFREE;
                p->sval = fr;
                dprintf(("refldbld: i=%d\n", i));
                if (nematch(pfa, rec)) {
                        pfa->initstat = 2;      /* horrible coupling to b.c */
                        pfa->initstat = 2;
                        dprintf(("match %s (%d chars)\n", patbeg, patlen));
                        (void) strncpy((char *)fr, (char *)rec, patbeg-rec);
                        fr += patbeg - rec + 1;
                        *(fr-1) = '\0';
                        rec = patbeg + patlen;
                } else {
                        dprintf(("no match %s\n", rec));
                        (void) strcpy((char *)fr, (char *)rec);
                        pfa->initstat = tempstat;
                        break;
                }
        }
        return (i);
}

void
recbld(void)            /* create $0 from $1..$NF if necessary */
recbld(void)
{
        int i;
        uchar *r, *p;
        uchar *p;
        size_t cnt, len, olen;

        if (donerec == 1)
                return;
        r = record;
        cnt = 0;
        olen = strlen((char *)*OFS);
        for (i = 1; i <= *NF; i++) {
                p = getsval(fldtab[i]);
                if (!adjbuf(&record, &record_size, 1 + strlen((char *)p) + r -
                    record, record_size, &r, "recbld 1"))
                        FATAL("created $0 '%.30s...' too long", record);
                while ((*r = *p++) != 0)
                        r++;
                p = getsval(getfld(i));
                len = strlen((char *)p);
                expand_buf(&record, &record_size, cnt + len + olen);
                (void) memcpy(&record[cnt], p, len);
                cnt += len;
                if (i < *NF) {
                        if (!adjbuf(&record, &record_size, 2 +
                            strlen((char *)*OFS) + r - record, record_size,
                            &r, "recbld 2"))
                                FATAL("created $0 '%.30s...' too long", record);
                        for (p = *OFS; (*r = *p++) != 0; )
                                r++;
```

```
411                         (void) memcpy(&record[cnt], *OFS, olen);
412                         cnt += olen;
576                 }
577         }
578         if (!adjbuf(&record, &record_size, 2 + r - record, record_size, &r,
579             "recbld 3"))
580                 FATAL("built giant record `%.30s...'", record);
581         *r = '\0';
582         dprintf(("in recbld inputFS=%s, fldtab[0]=%p\n", inputFS,
583             (void *)fldtab[0]));

585         if (freeable(fldtab[0]))
586                 xfree(fldtab[0]->sval);
587         fldtab[0]->tval = REC | STR | DONTFREE;
588         fldtab[0]->sval = record;

590         dprintf(("in recbld inputFS=%s, fldtab[0]=%p\n", inputFS,
591             (void *)fldtab[0]));
415         record[cnt] = '\0';
416         dprintf(("in recbld FS=%o, recloc=%p\n", **FS, (void *)recloc));
417         if (!(recloc->tval & DONTFREE))
418                 xfree(recloc->sval);
419         recloc->tval = REC | STR | DONTFREE;
420         recloc->sval = record;
421         dprintf(("in recbld FS=%o, recloc=%p\n", **FS, (void *)recloc));
592         dprintf(("recbld = |%s|\n", record));
593         donerec = 1;
594 }

596 int     errorflag       = 0;

598 void
599 yyerror(char *s)
426 Cell *
427 fieldadr(int n)
600 {
601         SYNTAX("%s", s);
429         if (n < 0)
430                 ERROR "trying to access field %d", n FATAL;
431         return (getfld(n));
602 }

434 int     errorflag       = 0;
435 char    errbuf[200];

604 void
605 SYNTAX(const char *fmt, ...)
438 yyerror(char *s)
606 {
607         extern uchar *cmdname, *curfname;
608         static int been_here = 0;
609         va_list varg;
610 #endif /* ! codereview */

612         if (been_here++ > 2)
613                 return;
614         (void) fprintf(stderr, "%s: ", cmdname);
615         va_start(varg, fmt);
616         (void) vfprintf(stderr, fmt, varg);
617         va_end(varg);
442         (void) fprintf(stderr, "%s: %s", cmdname, s);
618         (void) fprintf(stderr, gettext(" at source line %lld"), lineno);
619         if (curfname != NULL)
620                 (void) fprintf(stderr, gettext(" in function %s"), curfname);
621         if (compile_time == 1 && cursource() != NULL)
622                 (void) fprintf(stderr, gettext(" source file %s"), cursource());
```



```
623 #endif /* ! codereview */
624         (void) fprintf(stderr, "\n");
625         errorflag = 2;
626         eprint();
627 }

446 /*ARGSUSED*/
629 void
630 fpecatch(int n)
448 fpecatch(int sig)
631 {
632         FATAL("floating point exception %d", n);
450         ERROR "floating point exception" FATAL;
633 }
_____unchanged_portion_omitted_

666 void
667 FATAL(const char *fmt, ...)
668 {
669         extern uchar *cmdname;
670         va_list varg;

672         (void) fflush(stdout);
673         (void) fprintf(stderr, "%s: ", cmdname);
674         va_start(varg, fmt);
675         (void) vfprintf(stderr, fmt, varg);
676         va_end(varg);
677         error();
678         if (dbg > 1)    /* core dump if serious debugging on */
679                 abort();
680         exit(2);
681 }

683 void
684 WARNING(const char *fmt, ...)
485 error(int f, char *s)
685 {
487         extern Node *curnode;
686         extern uchar *cmdname;
687         va_list varg;
688 #endif /* ! codereview */

690         (void) fflush(stdout);
691         (void) fprintf(stderr, "%s: ", cmdname);
692         va_start(varg, fmt);
693         (void) vfprintf(stderr, fmt, varg);
694         va_end(varg);
695         error();
696 }

698 void
699 error(void)
700 {
701         extern Node *curnode;

489         (void) fprintf(stderr, "%s", s);
703         (void) fprintf(stderr, "\n");
704         if (compile_time != 2 && NR && *NR > 0) {
705                 (void) fprintf(stderr,
706                     gettext(" input record number %g"), (int) (*FNR));
493                 gettext(" input record number %g"), *FNR);
707                 if (strcmp((char *)*FILENAME, "-") != 0)
708                         (void) fprintf(stderr, gettext(", file %s"), *FILENAME);
709                 (void) fprintf(stderr, "\n");
710         }
711         if (compile_time != 2 && curnode)
```

```
712                 (void) fprintf(stderr, gettext(" source line number %d"),
499                 (void) fprintf(stderr, gettext(" source line number %lld\n"),
713                     curnode->lineno);
714         else if (compile_time != 2 && lineno) {
715                 (void) fprintf(stderr,
716                     gettext(" source line number %d"), lineno);
503                     gettext(" source line number %lld\n"), lineno);
717         }
718         if (compile_time == 1 && cursource() != NULL)
719                 (void) fprintf(stderr,
720                     gettext(" source file %s"), cursource());
721         (void) fprintf(stderr, "\n");
722 #endif /* ! codereview */
723         eprint();
505         if (f) {
506                 if (dbg)
507                         abort();
508                 exit(2);
509         }
724 }

726 static void
727 eprint(void)    /* try to print context around error */
728 {
729         uchar *p, *q;
730         int c;
731         static int been_here = 0;
732         extern uchar ebuf[], *ep;
518         extern uchar ebuf[300], *ep;

734         if (compile_time == 2 || compile_time == 0 || been_here++ > 0)
735                 return;
736         p = ep - 1;
737         if (p > ebuf && *p == '\n')
738                 p--;
739         for (; p > ebuf && *p != '\n' && *p != '\0'; p--)
740                 ;
741         while (*p == '\n')
742                 p++;
743         (void) fprintf(stderr, gettext(" context is\n\t"));
744         for (q = ep-1; q >= p && *q != ' ' && *q != '\t' && *q != '\n'; q--)
745                 ;
746         for (; p < q; p++)
747                 if (*p)
748                         (void) putc(*p, stderr);
749         (void) fprintf(stderr, " >>> ");
750         for (; p < ep; p++)
751                 if (*p)
752                         (void) putc(*p, stderr);
753         (void) fprintf(stderr, " <<< ");
754         if (*ep)
755                 while ((c = input()) != '\n' && c != '\0' && c != EOF) {
756                         (void) putc(c, stderr);
757                         bclass(c);
758                 }
759         (void) putc('\n', stderr);
760         ep = ebuf;
761 }
______unchanged_portion_omitted_

776 double
777 errcheck(double x, const char *s)
563 errcheck(double x, char *s)
778 {
565         extern int errno;
```



```
779         if (errno == EDOM) {
780                 errno = 0;
781                 WARNING("%s argument out of domain", s);
569                 ERROR "%s argument out of domain", s WARNING;
782                 x = 1;
783         } else if (errno == ERANGE) {
784                 errno = 0;
785                 WARNING("%s result out of range", s);
573                 ERROR "%s result out of range", s WARNING;
786                 x = 1;
787         }
788         return (x);
789 }

579 void
580 PUTS(uchar *s)
581 {
582         dprintf(("%s\n", s));
583 }

791 int
792 isclvar(const uchar *s) /* is s of form var=something? */
586 isclvar(uchar *s)       /* is s of form var=something? */
793 {
794         const uchar *os = s;
588         if (s != NULL) {

796         if (!isalpha(*s) && *s != '_')
797                 return (0);
798         for (; *s; s++) {
799                 if (!(isalnum(*s) || *s == '_'))
590                 /* Must begin with an underscore or alphabetic character */
591                 if (isalpha(*s) || (*s == '_')) {

593                         for (s++; *s; s++) {
594                                 /*
595                                  * followed by a sequence of underscores,
596                                  * digits, and alphabetics
597                                  */
598                                 if (!(isalnum(*s) || *s == '_')) {
800                         break;
801         }
601                         }
602                         return (*s == '=' && *(s + 1) != '=');
603                 }
604         }

803         return (*s == '=' && s > os && *(s+1) != '=');
606         return (0);
804 }

806 /* strtod is supposed to be a proper test of what's a valid number */
807 /* appears to be broken in gcc on linux: thinks 0x123 is a valid FP number */
808 /* wrong: violates 4.10.1.4 of ansi C standard */
609 #define MAXEXPON        38      /* maximum exponent for fp number */

810 int
811 is_number(const uchar *s)
612 is_number(uchar *s)
812 {
813         double r;
814         char *ep;
815         errno = 0;
816         r = strtod((const char *)s, &ep);
817         if (ep == (char *)s || r == HUGE_VAL || errno == ERANGE)
614         int d1, d2;
```

```
615         int point;
616         uchar *es;
617         extern char     radixpoint;

619         d1 = d2 = point = 0;
620         while (*s == ' ' || *s == '\t' || *s == '\n')
621                 s++;
622         if (*s == '\0')
623                 return (0);     /* empty stuff isn't number */
624         if (*s == '+' || *s == '-')
625                 s++;
626         if (!isdigit(*s) && *s != radixpoint)
627                 return (0);
628         if (isdigit(*s)) {
629                 do {
630                         d1++;
631                         s++;
632                 } while (isdigit(*s));
633         }
634         if (d1 >= MAXEXPON)
635                 return (0);     /* too many digits to convert */
636         if (*s == radixpoint) {
637                 point++;
638                 s++;
639         }
640         if (isdigit(*s)) {
641                 d2++;
642                 do {
643                         s++;
644                 } while (isdigit(*s));
645         }
646         if (!(d1 || point && d2))
647                 return (0);
648         if (*s == 'e' || *s == 'E') {
649                 s++;
650                 if (*s == '+' || *s == '-')
651                         s++;
652                 if (!isdigit(*s))
818                 return (0);
819         while (*ep == ' ' || *ep == '\t' || *ep == '\n')
820                 ep++;
821         if (*ep == '\0')
654                 es = s;
655                 do {
656                         s++;
657                 } while (isdigit(*s));
658                 if (s - es > 2) {
659                         return (0);
660                 } else if (s - es == 2 &&
661                     (int)(10 * (*es-'0') + *(es+1)-'0') >= MAXEXPON) {
662                         return (0);
663                 }
664         }
665         while (*s == ' ' || *s == '\t' || *s == '\n')
666                 s++;
667         if (*s == '\0')
822                 return (1);
823         else
824                 return (0);
825 }
```

```
673 void
674 init_buf(uchar **optr, size_t *sizep, size_t amt)
675 {
676         uchar   *nptr = NULL;
```



```
678         if ((nptr = malloc(amt)) == NULL)
679                 ERROR "out of space in init_buf" FATAL;
680         /* initial buffer should have NULL terminated */
681         *nptr = '\0';
682         if (sizep != NULL)
683                 *sizep = amt;
684         *optr = nptr;
685 }

687 void
688 r_expand_buf(uchar **optr, size_t *sizep, size_t req)
689 {
690         uchar   *nptr;
691         size_t  amt, size = *sizep;

693         if (size != 0 && req < (size - 1))
694                 return;
695         amt = req + 1 - size;
696         amt = (amt / LINE_INCR + 1) * LINE_INCR;

698         if ((nptr = realloc(*optr, size + amt)) == NULL)
699                 ERROR "out of space in expand_buf" FATAL;
700         /* initial buffer should have NULL terminated */
701         if (size == 0)
702                 *nptr = '\0';
703         *sizep += amt;
704         *optr = nptr;
705 }

707 void
708 adjust_buf(uchar **optr, size_t size)
709 {
710         uchar   *nptr;

712         if ((nptr = realloc(*optr, size)) == NULL)
713                 ERROR "out of space in adjust_buf" FATAL;
714         *optr = nptr;
715 }
```

```
**********************************************************
    7218 Sat May 11 15:20:44 2013
new/usr/src/cmd/awk/main.c
3731 Update nawk to version 20121220
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * Copyright (C) Lucent Technologies 1997
 * All Rights Reserved
 *
 * Permission to use, copy, modify, and distribute this software and
 * its documentation for any purpose and without fee is hereby
 * granted, provided that the above copyright notice appear in all
 * copies and that both that the copyright notice and this
 * permission notice and warranty disclaimer appear in supporting
 * documentation, and that the name Lucent Technologies or any of
 * its entities not be used in advertising or publicity pertaining
 * to distribution of the software without specific, written prior
 * permission.
 *
 * LUCENT DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
 * INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS.
 * IN NO EVENT SHALL LUCENT OR ANY OF ITS ENTITIES BE LIABLE FOR ANY
 * SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
 * WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER
 * IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION,
 * ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF
 * THIS SOFTWARE.
 */
/*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
/*        All Rights Reserved   */

#include <stdio.h>
#include <ctype.h>
#include <signal.h>
#include <locale.h>
#include <libintl.h>
#include <stdarg.h>
#include <errno.h>
#include <values.h>
#include <langinfo.h>
```



```
#include "awk.h"
#include "y.tab.h"

char    *version = "version 20121220";
char    *version = "version Oct 11, 1989";

int     dbg     = 0;
Awkfloat        srand_seed = 1;
#endif /* ! codereview */
uchar   *cmdname;       /* gets argv[0] for error messages */
uchar   *lexprog;       /* points to program argument if it exists */
int     compile_time = 2;       /* for error printing: */
                                /* 2 = cmdline, 1 = compile, 0 = running */
char    radixpoint = '.';

#define MAX_PFILE       20      /* max number of -f's */

static uchar    *pfile[MAX_PFILE];      /* program filenames from -f's */
static uchar    **pfile = NULL; /* program filenames from -f's */
static int      npfile = 0;     /* number of filenames */
static int      curpfile = 0;   /* current filename */

int     safe = 0;       /* 1 => "safe" mode */

#endif /* ! codereview */
int
main(int argc, char *argv[], char *envp[])
{
        const uchar *fs = NULL;
        uchar *fs = NULL;
        char    *nl_radix;
        /*
         * At this point, numbers are still scanned as in
         * the POSIX locale.
         * (POSIX.2, volume 2, P867, L4742-4757)
         */
        (void) setlocale(LC_ALL, "");
        (void) setlocale(LC_NUMERIC, "C");
#if !defined(TEXT_DOMAIN)       /* Should be defined by cc -D */
#define TEXT_DOMAIN     "SYS_TEST"      /* Use this only if it weren't */
#endif
        (void) textdomain(TEXT_DOMAIN);
        cmdname = (uchar *)argv[0];
        if (argc == 1) {
                (void) fprintf(stderr, gettext(
                    "Usage: %s [-f programfile | 'program'] [-Ffieldsep] "
                    "[-v var=value] [files]\n"), cmdname);
                exit(1);
        }
        (void) signal(SIGFPE, fpecatch);

        srand_seed = 1;
        srand((unsigned int)srand_seed);

#endif /* ! codereview */
        yyin = NULL;
        symtab = makesymtab(NSYMTAB/NSYMTAB);
        syminit();
        while (argc > 1 && argv[1][0] == '-' && argv[1][1] != '\0') {
                if (strcmp(argv[1], "-version") == 0 ||
                    strcmp(argv[1], "--version") == 0) {
                        (void) printf("awk %s\n", version);
                        exit(0);
                        break;
                }
                if (strncmp(argv[1], "--", 2) == 0) {
```

```
 72                 if (strcmp(argv[1], "--") == 0) {
119                         /* explicit end of args */
120                         argc--;
121                         argv++;
122                         break;
123                 }
124                 switch (argv[1][1]) {
125                 case 's':
126                         if (strcmp(argv[1], "-safe") == 0)
127                                 safe = 1;
128                         break;
129 #endif /* ! codereview */
130                 case 'f':       /* next argument is program filename */
131                         if (argv[1][2] != 0) {  /* arg is -fsomething */
132                                 if (npfile >= MAX_PFILE - 1)
133                                         FATAL("too many -f options");
134                                 pfile[npfile++] = (uchar *)&argv[1][2];
135                         } else {                /* arg is -f something */
136                                 argc--; argv++;
 79                         argc--;
 80                         argv++;
137                                 if (argc <= 1)
138                                         FATAL("no program filename");
139                                 if (npfile >= MAX_PFILE - 1)
140                                         FATAL("too many -f options");
 82                                 ERROR "no program filename" FATAL;
 83                         pfile = realloc(pfile, sizeof (uchar *) * (npfile + 1));
 84                         if (pfile == NULL)
 85                                 ERROR "out of space in main" FATAL;
141                                 pfile[npfile++] = (uchar *)argv[1];
142                         }
143 #endif /* ! codereview */
144                         break;
145                 case 'F':       /* set field separator */
146                         if (argv[1][2] != 0) {  /* arg is -Fsomething */
147                                 /* wart: t=>\t */
148                                 if (argv[1][2] == 't' && argv[1][3] == 0)
149                                         fs = (uchar *) "\t";
150                                 else if (argv[1][2] != 0)
151                                         fs = (uchar *)&argv[1][2];
152                         } else {                /* arg is -F something */
153                                 argc--; argv++;
 87                                 if (argc > 1) {
154                                 /* wart: t=>\t */
155                                 if (argc > 1 && argv[1][0] == 't' &&
 89                                         if (argv[1][0] == 't' &&
156                                     argv[1][1] == 0)
157                                         fs = (uchar *) "\t";
158                                 else if (argc > 1 && argv[1][0] != 0)
 92                                         else if (argv[1][0] != 0)
159                                         fs = (uchar *)&argv[1][0];
160                         }
 95                         }
161                         if (fs == NULL || *fs == '\0')
162                                 WARNING("field separator FS is empty");
 97                                 ERROR "field separator FS is empty" WARNING;
163                         break;
164                 case 'v':       /* -v a=1 to be done NOW.  one -v for each */
165                         if (argv[1][2] != 0) {  /* arg is -vsomething */
166                                 if (isclvar((uchar *)&argv[1][2]))
167                                         setclvar((uchar *)&argv[1][2]);
168                                 else
169                                         FATAL(
170                                 "invalid -v option argument: %s", &argv[1][2]);
171                         } else {        /* arg is -v something */
172                                 argc--; argv++;
```



```
173                                 if (argc <= 1)
174                                         FATAL("no variable name");
175                                 if (isclvar((uchar *)argv[1]))
100                         if (argv[1][2] == '\0' && --argc > 1 &&
101                             isclvar((uchar *)(++argv)[1]))
176                                         setclvar((uchar *)argv[1]);
177                                 else
178                                         FATAL(
179                                 "invalid -v option argument: %s", argv[1]);
180                         }
181 #endif /* ! codereview */
182                         break;
183                 case 'd':
184                         dbg = atoi(&argv[1][2]);
185                         if (dbg == 0)
186                                 dbg = 1;
187                         (void) printf("awk %s\n", version);
188                         break;
189                 default:
190                         WARNING("unknown option %s ignored", argv[1]);
103                         ERROR "unknown option %s ignored", argv[1] WARNING;
191                         break;
192                 }
193                 argc--;
194                 argv++;
195         }
196         /* argv[1] is now the first argument */
197         if (npfile == 0) {      /* no -f; first argument is program */
198                 if (argc <= 1) {
199                         if (dbg)
200                                 exit(0);
201                         FATAL("no program given");
114                         ERROR "no program given" FATAL;
202                 }
203                 dprintf(("program = |%s|\n", argv[1]));
204                 lexprog = (uchar *)argv[1];
205                 argc--;
206                 argv++;
207         }
208         recinit(record_size);
209         syminit();
210 #endif /* ! codereview */
211         compile_time = 1;
212         argv[0] = (char *)cmdname;      /* put prog name at front of arglist */
213         dprintf(("argc=%d, argv[0]=%s\n", argc, argv[0]));
214         arginit(argc, (uchar **)argv);
215         if (!safe)
216 #endif /* ! codereview */
217                 envinit((uchar **)envp);
218         (void) yyparse();
219         if (fs)
220                 *FS = qstring(fs, '\0');
221         dprintf(("errorflag=%d\n", errorflag));
222         /*
223          * done parsing, so now activate the LC_NUMERIC
224          */
225         (void) setlocale(LC_ALL, "");
226         nl_radix = nl_langinfo(RADIXCHAR);
227         if (nl_radix)
228                 radixpoint = *nl_radix;

230         if (errorflag == 0) {
231                 compile_time = 0;
232                 run(winner);
233         } else
234                 bracecheck();
```

```
235         return (errorflag);
236 }

238 int
239 pgetc(void)             /* get 1 character from awk program */
121 pgetc(void)             /* get program character */
240 {
241         int c;

243         for (;;) {
244                 if (yyin == NULL) {
245                         if (curpfile >= npfile)
246                                 return (EOF);
247                         if (strcmp((char *)pfile[curpfile], "-") == 0)
248                                 yyin = stdin;
249                         else if ((yyin = fopen(
250                             (char *)pfile[curpfile], "r")) == NULL)
251                                 FATAL("can't open file %s", pfile[curpfile]);
252                         lineno = 1;
129                         yyin = (strcmp((char *)pfile[curpfile], "-") == 0) ?
130                             stdin : fopen((char *)pfile[curpfile], "r");
131                         if (yyin == NULL) {
132                                 ERROR "can't open file %s",
133                                     pfile[curpfile] FATAL;
134                         }
253                 }
254                 if ((c = getc(yyin)) != EOF)
255                         return (c);
256                 if (yyin != stdin)
257 #endif /* ! codereview */
258                         (void) fclose(yyin);
259                 yyin = NULL;
260                 curpfile++;
261         }
262 }

264 uchar *
265 cursource(void)         /* current source file name */
266 {
267         if (npfile > 0)
268                 return (pfile[curpfile]);
269         else
270                 return (NULL);
271 }
272 #endif /* ! codereview */
```

```
**********************************************************
    5935 Sat May 11 15:20:44 2013
new/usr/src/cmd/awk/maketab.c
3731 Update nawk to version 20121220
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright (c) 1989, 2010, Oracle and/or its affiliates. All rights reserved.
 */

/*
 * Copyright (C) Lucent Technologies 1997
 * All Rights Reserved
 *
 * Permission to use, copy, modify, and distribute this software and
 * its documentation for any purpose and without fee is hereby
 * granted, provided that the above copyright notice appear in all
 * copies and that both that the copyright notice and this
 * permission notice and warranty disclaimer appear in supporting
 * documentation, and that the name Lucent Technologies or any of
 * its entities not be used in advertising or publicity pertaining
 * to distribution of the software without specific, written prior
 * permission.
 *
 * LUCENT DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
 * INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS.
 * IN NO EVENT SHALL LUCENT OR ANY OF ITS ENTITIES BE LIABLE FOR ANY
 * SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
 * WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER
 * IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION,
 * ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF
 * THIS SOFTWARE.
 */
/*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
/*        All Rights Reserved   */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <libintl.h>
#include "awk.h"
#include "y.tab.h"

struct xx {
        int token;
        const char *name;
        const char *pname;
        char *name;
        char *pname;
} proc[] = {
        { PROGRAM, "program", NULL },
        { BOR, "boolop", " || " },
        { AND, "boolop", " && " },
        { NOT, "boolop", " !" },
        { NE, "relop", " != " },
        { EQ, "relop", " == " },
        { LE, "relop", " <= " },
        { LT, "relop", " < " },
        { GE, "relop", " >= " },
        { GT, "relop", " > " },
        { ARRAY, "array", NULL },
        { INDIRECT, "indirect", "$(" },
        { SUBSTR, "substr", "substr" },
        { SUB, "sub", "sub" },
        { GSUB, "gsub", "gsub" },
        { INDEX, "sindex", "sindex" },
        { SPRINTF, "awksprintf", "sprintf " },
        { SPRINTF, "a_sprintf", "sprintf " },
        { ADD, "arith", " + " },
        { MINUS, "arith", " - " },
        { MULT, "arith", " * " },
        { DIVIDE, "arith", " / " },
        { MOD, "arith", " % " },
        { UMINUS, "arith", " -" },
        { POWER, "arith", " ** " },
        { PREINCR, "incrdecr", "++" },
        { POSTINCR, "incrdecr", "++" },
        { PREDECR, "incrdecr", "--" },
        { POSTDECR, "incrdecr", "--" },
        { CAT, "cat", " " },
        { PASTAT, "pastat", NULL },
        { PASTAT2, "dopa2", NULL },
        { MATCH, "matchop", " ~ " },
        { NOTMATCH, "matchop", " !~ " },
        { MATCHFCN, "matchop", "matchop" },
        { INTEST, "intest", "intest" },
        { PRINTF, "awkprintf", "printf" },
        { PRINT, "printstat", "print" },
        { PRINTF, "aprintf", "printf" },
        { PRINT, "print", "print" },
        { CLOSE, "closefile", "closefile" },
        { DELETE, "awkdelete", "awkdelete" },
        { DELETE, "delete", "delete" },
        { SPLIT, "split", "split" },
        { ASSIGN, "assign", " = " },
        { ADDEQ, "assign", " += " },
        { SUBEQ, "assign", " -= " },
        { MULTEQ, "assign", " *= " },
        { DIVEQ, "assign", " /= " },
        { MODEQ, "assign", " %= " },
        { POWEQ, "assign", " ^= " },
        { CONDEXPR, "condexpr", " ?: " },
        { IF, "ifstat", "if(" },
        { WHILE, "whilestat", "while(" },
        { FOR, "forstat", "for(" },
        { DO, "dostat", "do" },
        { IN, "instat", "instat" },
        { NEXT, "jump", "next" },
        { NEXTFILE, "jump", "nextfile" },
#endif /* ! codereview */
        { EXIT, "jump", "exit" },
        { BREAK, "jump", "break" },
```

```
        { CONTINUE, "jump", "continue" },
        { RETURN, "jump", "ret" },
        { BLTIN, "bltin", "bltin" },
        { CALL, "call", "call" },
        { ARG, "arg", "arg" },
        { VARNF, "getnf", "NF" },
        { GETLINE, "awkgetline", "getline" },
        { GETLINE, "getaline", "getline" },
        { 0, "", "" },
};

#define SIZE    (LASTTOKEN - FIRSTTOKEN + 1)
const char *table[SIZE];
#define SIZE    LASTTOKEN - FIRSTTOKEN + 1
char *table[SIZE];
char *names[SIZE];

/* ARGSUSED */
#endif /* ! codereview */
int
main(int argc, char *argv[])
main()
{
        const struct xx *p;
        struct xx *p;
        int i, n, tok;
        char c;
        FILE *fp;
        char buf[200], name[200], def[200];
        char buf[100], name[100], def[100];

        (void) printf("#include <stdio.h>\n");
        (void) printf("#include \"awk.h\"\n");
        (void) printf("#include \"y.tab.h\"\n\n");
        for (i = SIZE; --i >= 0; )
                names[i] = "";
        printf("#include \"awk.h\"\n");
        printf("#include \"y.tab.h\"\n\n");

        if ((fp = fopen("y.tab.h", "r")) == NULL) {
                (void) fprintf(stderr,
                    gettext("maketab can't open y.tab.h!\n"));
                fprintf(stderr, gettext("maketab can't open y.tab.h!\n"));
                exit(1);
        }
        (void) printf("static uchar *printname[%d] = {\n", SIZE);
        printf("static uchar *printname[%d] = {\n", SIZE);
        i = 0;
        while (fgets(buf, sizeof (buf), fp) != NULL) {
                /* LINTED E_SEC_SCANF_UNBOUNDED_COPY */
#endif /* ! codereview */
                n = sscanf(buf, "%1c %s %s %d", &c, def, name, &tok);
                /* not a valid #define? */
                if (c != '#' || (n != 4 && strcmp(def, "define") != 0))
                        continue;
                if (tok < FIRSTTOKEN || tok > LASTTOKEN)
                if (c != '#' || n != 4 && strcmp(def, "define") != 0)
                        continue;
                names[tok-FIRSTTOKEN] = (char *)malloc(strlen(name)+1);
                (void) strcpy(names[tok-FIRSTTOKEN], name);
                (void) printf("\t(uchar *) \"%s\",\t/* %d */\n", name, tok);
                if (tok < FIRSTTOKEN || tok > LASTTOKEN) {
                        fprintf(stderr, gettext("maketab funny token %d %s\n"),
                            tok, buf);
                        exit(1);
                }
```



```
                names[tok-FIRSTTOKEN] = malloc(strlen(name)+1);
                strcpy(names[tok-FIRSTTOKEN], name);
                printf("\t(uchar *) \"%s\",\t/* %d */\n", name, tok);
                i++;
        }
        (void) printf("};\n\n");
        printf("};\n\n");

        for (p = proc; p->token != 0; p++)
                table[p->token-FIRSTTOKEN] = p->name;
        (void) printf("\nCell *(*proctab[%d])(Node **, int) = {\n", SIZE);
        printf("\nCell *(*proctab[%d])() = {\n", SIZE);
        for (i = 0; i < SIZE; i++)
                if (table[i] == 0)
                        (void) printf("\tnullproc,\t/* %s */\n", names[i]);
                        printf("\tnullproc,\t/* %s */\n", names[i]);
                else
                        (void) printf("\t%s,\t/* %s */\n", table[i], names[i]);
        (void) printf("};\n\n");
                        printf("\t%s,\t/* %s */\n", table[i], names[i]);
        printf("};\n\n");

        /* print a tokname() function */
        (void) printf("uchar *\ntokname(int n)\n");
        (void) printf("{\n");
        (void) printf(" static char buf[100];\n\n");
        (void) printf(" if (n < FIRSTTOKEN || n > LASTTOKEN) {\n");
        (void) printf("         (void) sprintf(buf, \"token %%d\", n);\n");
        (void) printf("         return ((uchar *)buf);\n");
        (void) printf(" }\n");
        (void) printf(" return printname[n-FIRSTTOKEN];\n");
        (void) printf("}\n");
        return (0);
        printf("uchar *\ntokname(int n)\n");    /* print a tokname() function */
        printf("{\n");
        printf("        static char buf[100];\n\n");
        printf("        if (n < FIRSTTOKEN || n > LASTTOKEN) {\n");
        printf("                (void) sprintf(buf, \"token %%d\", n);\n");
        printf("                return ((uchar *)buf);\n");
        printf("        }\n");
        printf("        return printname[n-257];\n");
        printf("}\n");
        exit(0);
}
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    6201 Sat May 11 15:20:45 2013
new/usr/src/cmd/awk/parse.c
3731 Update nawk to version 20121220
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License, Version 1.0 only
 * (the "License").  You may not use this file except in compliance
 * with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright 2005 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * Copyright (C) Lucent Technologies 1997
 * All Rights Reserved
 *
 * Permission to use, copy, modify, and distribute this software and
 * its documentation for any purpose and without fee is hereby
 * granted, provided that the above copyright notice appear in all
 * copies and that both that the copyright notice and this
 * permission notice and warranty disclaimer appear in supporting
 * documentation, and that the name Lucent Technologies or any of
 * its entities not be used in advertising or publicity pertaining
 * to distribution of the software without specific, written prior
 * permission.
 *
 * LUCENT DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
 * INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS.
 * IN NO EVENT SHALL LUCENT OR ANY OF ITS ENTITIES BE LIABLE FOR ANY
 * SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
 * WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER
 * IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION,
 * ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF
 * THIS SOFTWARE.
 */
/*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
/*        All Rights Reserved   */

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

#define DEBUG
#include "awk.h"
#include "y.tab.h"

Node *
nodealloc(int n)
{
        Node *x;
        register Node *x;

        x = (Node *)malloc(sizeof (Node) + (n - 1) * sizeof (Node *));
        if (x == NULL)
                FATAL("out of space in nodealloc");
                ERROR "out of space in nodealloc" FATAL;
        x->nnext = NULL;
        x->lineno = lineno;
        return (x);
}
```



```
_____unchanged_portion_omitted_

Node *
node1(int a, Node *b)
{
        Node *x;
        register Node *x;

        x = nodealloc(1);
        x->nobj = a;
        x->narg[0] = b;
        return (x);
}

Node *
node2(int a, Node *b, Node *c)
{
        Node *x;
        register Node *x;

        x = nodealloc(2);
        x->nobj = a;
        x->narg[0] = b;
        x->narg[1] = c;
        return (x);
}

Node *
node3(int a, Node *b, Node *c, Node *d)
{
        Node *x;
        register Node *x;

        x = nodealloc(3);
        x->nobj = a;
        x->narg[0] = b;
        x->narg[1] = c;
        x->narg[2] = d;
        return (x);
}

Node *
node4(int a, Node *b, Node *c, Node *d, Node *e)
{
        Node *x;
        register Node *x;
        x = nodealloc(4);
        x->nobj = a;
        x->narg[0] = b;
        x->narg[1] = c;
        x->narg[2] = d;
        x->narg[3] = e;
        return (x);
}
```

```
125 Node *
126 stat1(int a, Node *b)
107 stat3(int a, Node *b, Node *c, Node *d)
127 {
128         Node *x;
109         register Node *x;

130         x = node1(a, b);
111         x = node3(a, b, c, d);
131         x->ntype = NSTAT;
132         return (x);
133 }

135 Node *
136 stat2(int a, Node *b, Node *c)
117 op2(int a, Node *b, Node *c)
137 {
138         Node *x;
119         register Node *x;

140         x = node2(a, b, c);
141         x->ntype = NSTAT;
122         x->ntype = NEXPR;
142         return (x);
143 }

145 Node *
146 stat3(int a, Node *b, Node *c, Node *d)
127 op1(int a, Node *b)
147 {
148         Node *x;
129         register Node *x;

150         x = node3(a, b, c, d);
151         x->ntype = NSTAT;
131         x = node1(a, b);
132         x->ntype = NEXPR;
152         return (x);
153 }

155 Node *
156 stat4(int a, Node *b, Node *c, Node *d, Node *e)
137 stat1(int a, Node *b)
157 {
158         Node *x;
139         register Node *x;

160         x = node4(a, b, c, d, e);
141         x = node1(a, b);
161         x->ntype = NSTAT;
162         return (x);
163 }

165 Node *
166 op1(int a, Node *b)
147 op3(int a, Node *b, Node *c, Node *d)
167 {
168         Node *x;
149         register Node *x;

170         x = node1(a, b);
151         x = node3(a, b, c, d);
171         x->ntype = NEXPR;
172         return (x);
173 }
```



```
175 Node *
176 op2(int a, Node *b, Node *c)
157 op4(int a, Node *b, Node *c, Node *d, Node *e)
177 {
178         Node *x;
159         register Node *x;

180         x = node2(a, b, c);
161         x = node4(a, b, c, d, e);
181         x->ntype = NEXPR;
182         return (x);
183 }

185 Node *
186 op3(int a, Node *b, Node *c, Node *d)
167 stat2(int a, Node *b, Node *c)
187 {
188         Node *x;
169         register Node *x;

190         x = node3(a, b, c, d);
191         x->ntype = NEXPR;
171         x = node2(a, b, c);
172         x->ntype = NSTAT;
192         return (x);
193 }

195 Node *
196 op4(int a, Node *b, Node *c, Node *d, Node *e)
177 stat4(int a, Node *b, Node *c, Node *d, Node *e)
197 {
198         Node *x;
179         register Node *x;

200         x = node4(a, b, c, d, e);
201         x->ntype = NEXPR;
182         x->ntype = NSTAT;
202         return (x);
203 }

205 Node
206 *celltonode(Cell *a, int b)
186 Node *
187 valtonode(Cell *a, int b)
207 {
208         Node *x;
189         register Node *x;

210         a->ctype = OCELL;
211         a->csub = b;
212         x = node1(0, (Node *)a);
213         x->ntype = NVALUE;
214         return (x);
215 }

217 Node
218 *rectonode(void)        /* make $0 into a Node */
198 Node *
199 rectonode(void)
219 {
220         extern Cell *literal0;
221         return (op1(INDIRECT, celltonode(literal0, CUNK)));
201         /* return valtonode(lookup("$0", symtab), CFLD); */
202         return (valtonode(recloc, CFLD));
222 }
```

```
Node *
makearr(Node *p)
{
        Cell *cp;

        if (isvalue(p)) {
                cp = (Cell *)(p->narg[0]);
                if (isfcn(cp))
                        SYNTAX("%s is a function, not an array", cp->nval);
                if (isfunc(cp))
                        ERROR "%s is a function, not an array", cp->nval SYNTAX;
                else if (!isarr(cp)) {
                        xfree(cp->sval);
                        cp->sval = (uchar *)makesymtab(NSYMTAB);
                        cp->tval = ARR;
                }
        }
        return (p);
}

/*
 * The One True Awk sets PA2NUM to 50, which is not enough for webrev(1).
 */
#define PA2NUM  250     /* max number of pat,pat patterns allowed */
int     paircnt;                /* number of them in use */
int     pairstack[PA2NUM];      /* state of each pat,pat */

#endif /* ! codereview */
Node *
pa2stat(Node *a, Node *b, Node *c)      /* pat, pat {...} */
pa2stat(Node *a, Node *b, Node *c)
{
        Node *x;
        register Node *x;

        x = node4(PASTAT2, a, b, c, itonp(paircnt));
        if (paircnt++ >= PA2NUM)
                SYNTAX("limited to %d pat,pat statements", PA2NUM);
        x = node4(PASTAT2, a, b, c, (Node *)paircnt);
        paircnt++;
        x->ntype = NSTAT;
        return (x);
}

Node *
linkum(Node *a, Node *b)
{
        Node *c;
        register Node *c;

        if (errorflag)  /* don't link things that are wrong */
                return (a);
        if (a == NULL)
                return (b);
        else if (b == NULL)
                return (a);
        for (c = a; c->nnext != NULL; c = c->nnext)
                ;
        c->nnext = b;
        return (a);
}

void
defn(Cell *v, Node *vl, Node *st)       /* turn on FCN bit in definition */
{                                       /*   body of function, arglist */
```



```
{
        Node *p;
        int n;

        if (isarr(v)) {
                SYNTAX("'%s' is an array name and a function name",
                    v->nval);
                return;
        }
        if (isarg(v->nval) != -1) {
                SYNTAX("'%s' is both function name and argument name",
                    v->nval);
                ERROR "'%s' is an array name and a function name",
                    v->nval SYNTAX;
                return;
        }

#endif /* ! codereview */
        v->tval = FCN;
        v->sval = (uchar *)st;
        n = 0;  /* count arguments */
        for (p = vl; p; p = p->nnext)
                n++;
        v->fval = n;
        dprintf(("defining func %s (%d args)\n", v->nval, n));
}

int
isarg(const uchar *s)   /* is s in argument list for current function? */
{                       /* return -1 if not, otherwise arg # */
isarg(uchar *s) /* is s in argument list for current function? */
{
        extern Node *arglist;
        Node *p = arglist;
        int n;

        for (n = 0; p != 0; p = p->nnext, n++) {
                if (strcmp((char *)((Cell *)(p->narg[0]))->nval,
                    (char *)s) == 0) {
                        return (n);
                }
        }
        return (-1);
}

int
ptoi(void *p)           /* convert pointer to integer */
{
        return ((int)(long)p);  /* swearing that p fits, of course */
}

Node
*itonp(int i)           /* and vice versa */
{
        return ((Node *)(long)i);
}
#endif /* ! codereview */
```

```
**********************************************************
   48876 Sat May 11 15:20:45 2013
new/usr/src/cmd/awk/run.c
3731 Update nawk to version 20121220
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright (c) 1989, 2010, Oracle and/or its affiliates. All rights reserved.
 */

/*
 * Copyright (C) Lucent Technologies 1997
 * All Rights Reserved
 *
 * Permission to use, copy, modify, and distribute this software and
 * its documentation for any purpose and without fee is hereby
 * granted, provided that the above copyright notice appear in all
 * copies and that both that the copyright notice and this
 * permission notice and warranty disclaimer appear in supporting
 * documentation, and that the name Lucent Technologies or any of
 * its entities not be used in advertising or publicity pertaining
 * to distribution of the software without specific, written prior
 * permission.
 *
 * LUCENT DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
 * INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS.
 * IN NO EVENT SHALL LUCENT OR ANY OF ITS ENTITIES BE LIABLE FOR ANY
 * SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
 * WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER
 * IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION,
 * ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF
 * THIS SOFTWARE.
 */
/*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
/*        All Rights Reserved   */

#define tempfree(x)     if (istemp(x)) tfree(x);
#define tempfree(x, s)  if (istemp(x)) tfree(x, s)

#define execute(p) r_execute(p)

#define DEBUG
#include <setjmp.h>
#include <math.h>
#include <time.h>
#endif /* ! codereview */
```



```
#include "awk.h"
#include        <math.h>
#include "y.tab.h"
#include        <stdio.h>
#include        <ctype.h>
#include        <setjmp.h>
#include        <time.h>

#ifndef FOPEN_MAX
#define FOPEN_MAX       15      /* max number of open files, from ANSI std. */
#endif

static  Cell    *execute(Node *);
static  Cell    *gettemp(void);
static  Cell    *copycell(Cell *);
static  FILE    *openfile(int, const uchar *);
static  FILE    *redirect(int, Node *);
static  const char *filename(FILE *);
static  void    flush_all(void);
#endif /* ! codereview */

static jmp_buf env;
extern  Awkfloat        srand_seed;
#endif /* ! codereview */

Node    *winner = NULL; /* root of parse tree */
static  Cell    *r_execute(Node *);
static  Cell    *gettemp(char *), *copycell(Cell *);
static  FILE    *openfile(int, uchar *), *redirect(int, Node *);

int     paircnt;
Node    *winner = NULL;

static Cell     *tmps;

static Cell     *tmps;  /* free temporary cells for execution */
#endif /* ! codereview */
static Cell     truecell        = { OBOOL, BTRUE, 0, 0, 1.0, NUM };
Cell    *True   = &truecell;
Cell    *true   = &truecell;
static Cell     falsecell       = { OBOOL, BFALSE, 0, 0, 0.0, NUM };
Cell    *False  = &falsecell;
Cell    *false  = &falsecell;
static Cell     breakcell       = { OJUMP, JBREAK, 0, 0, 0.0, NUM };
Cell    *jbreak = &breakcell;
static Cell     contcell        = { OJUMP, JCONT, 0, 0, 0.0, NUM };
Cell    *jcont  = &contcell;
static Cell     nextcell        = { OJUMP, JNEXT, 0, 0, 0.0, NUM };
Cell    *jnext  = &nextcell;
static Cell     nextfilecell    = { OJUMP, JNEXTFILE, 0, 0, 0.0, NUM };
Cell    *jnextfile      = &nextfilecell;
#endif /* ! codereview */
static Cell     exitcell        = { OJUMP, JEXIT, 0, 0, 0.0, NUM };
Cell    *jexit  = &exitcell;
static Cell     retcell         = { OJUMP, JRET, 0, 0, 0.0, NUM };
Cell    *jret   = &retcell;
static Cell     tempcell        = { OCELL, CTEMP, 0, (uchar *)"", 0.0, NUM|
    STR|DONTFREE};
static Cell     tempcell        = { OCELL, CTEMP, 0, 0, 0.0, NUM };

Node    *curnode = NULL;        /* the node being executed, for debugging */

static  void    tfree(Cell *);
static  void    tfree(Cell *, char *);
static  void    closeall(void);
static  double  ipow(double, int);
```

```
102 static  void    stdinit(void);

104 /*
105  * buffer memory management
106  *
107  * pbuf:    address of pointer to buffer being managed
108  * psiz:    address of buffer size variable
109  * minlen:  minimum length of buffer needed
110  * quantum: buffer size quantum
111  * pbptr:   address of movable pointer into buffer, or 0 if none
112  * whatrtn: name of the calling routine if failure should cause fatal error
113  *
114  * return   0 for realloc failure, !=0 for success
115  */
116 int adjbuf(uchar **pbuf, size_t *psiz, int minlen, int quantum, uchar **pbptr,
117     const char *whatrtn)
118 {
119         if (minlen > *psiz) {
120                 uchar *tbuf;
121                 int rminlen = quantum ? minlen % quantum : 0;
122                 int boff = pbptr ? *pbptr - *pbuf : 0;
123                 /* round up to next multiple of quantum */
124                 if (rminlen)
125                         minlen += quantum - rminlen;
126                 tbuf = (uchar *)realloc(*pbuf, minlen);
127                 dprintf(("adjbuf %s: %d %d (pbuf=%p, tbuf=%p)\n", whatrtn,
128                     *psiz, minlen, (void *)*pbuf, (void *)tbuf));
129                 if (tbuf == NULL) {
130                         if (whatrtn)
131                                 FATAL("out of memory in %s", whatrtn);
132                         return (0);
133                 }
134                 *pbuf = (uchar *)tbuf;
135                 *psiz = minlen;
136                 if (pbptr)
137                         *pbptr = tbuf + boff;
138         }
139         return (1);
140 }
141 #endif /* ! codereview */

143 void
144 run(Node *a)            /* execution of parse tree starts here */
 70 run(Node *a)
145 {
146         stdinit();
147 #endif /* ! codereview */
148         (void) execute(a);
149         closeall();
150 }

152 static Cell *
153 execute(Node *u)        /* execute a node of the parse tree */
 72 r_execute(Node *u)
154 {
155         Cell *(*proc)(Node **, int);
156         Cell *x;
157         Node *a;
 74         register Cell *(*proc)();
 75         register Cell *x;
 76         register Node *a;

159         if (u == NULL)
160                 return (True);
 79                 return (true);
161         for (a = u; ; a = a->nnext) {
```



```
162                 curnode = a;
163                 if (isvalue(a)) {
164                         x = (Cell *)(a->narg[0]);
165                         if (isfld(x) && !donefld)
 83                         x = (Cell *) (a->narg[0]);
 84                         if ((x->tval & FLD) && !donefld)
166                                 fldbld();
167                         else if (isrec(x) && !donerec)
 86                         else if ((x->tval & REC) && !donerec)
168                                 recbld();
169                         return (x);
170                 }
171                 /* probably a Cell* but too risky to print */
172                 if (notlegal(a->nobj))
173                         FATAL("illegal statement");
 92                         ERROR "illegal statement" FATAL;
174                 proc = proctab[a->nobj-FIRSTTOKEN];
175                 x = (*proc)(a->narg, a->nobj);
176                 if (isfld(x) && !donefld)
 95                 if ((x->tval & FLD) && !donefld)
177                         fldbld();
178                 else if (isrec(x) && !donerec)
 97                 else if ((x->tval & REC) && !donerec)
179                         recbld();
180                 if (isexpr(a))
181                         return (x);
101                 /* a statement, goto next statement */
182                 if (isjump(x))
183                         return (x);
184                 if (a->nnext == (Node *)NULL)
185                         return (x);
186                 tempfree(x);
106                 tempfree(x, "execute");
187         }
188 }

190 /*ARGSUSED*/
191 Cell *
192 program(Node **a, int n)        /* execute an awk program */
193 {                               /* a[0] = BEGIN, a[1] = body, a[2] = END */
194         Cell *x;
112 program(Node **a, int n)
113 {
114         register Cell *x;

196         if (setjmp(env) != 0)
197                 goto ex;
198         if (a[0]) {             /* BEGIN */
199                 x = execute(a[0]);
200                 if (isexit(x))
201                         return (True);
121                         return (true);
202                 if (isjump(x)) {
203                         FATAL(
204                 "illegal break, continue, next or nextfile from BEGIN");
123                         ERROR "illegal break, continue or next from BEGIN"
124                             FATAL;
205                 }
206                 tempfree(x);
126                 tempfree(x, "");
207         }

209         if (a[1] || a[2]) {
210                 while (getrec(&record, &record_size, 1) > 0) {
128 loop:
129         if (a[1] || a[2])
```

```
130                 while (getrec(&record, &record_size) > 0) {
211                         x = execute(a[1]);
212                         if (isexit(x))
213                                 break;
214                         tempfree(x);
215                 }
134                         tempfree(x, "");
216         }
217 ex:
218         if (setjmp(env) != 0)   /* handles exit within END */
137         if (setjmp(env) != 0)
219                 goto ex1;
220         if (a[2]) {             /* END */
221                 x = execute(a[2]);
222                 if (isbreak(x) || isnext(x) || iscont(x))
223                         FATAL(
224                 "illegal break, continue, next or nextfile from END");
225                 tempfree(x);
141                 if (iscont(x))  /* read some more */
142                         goto loop;
143                 if (isbreak(x) || isnext(x))
144                         ERROR "illegal break or next from END" FATAL;
145                 tempfree(x, "");
226         }
227 ex1:
228         return (True);
148         return (true);
229 }

231 struct Frame {  /* stack frame for awk function calls */
151 struct Frame {
232         int nargs;      /* number of arguments in this call */
233         Cell *fcncell;  /* pointer to Cell for function */
234         Cell **args;    /* pointer to array of arguments after execute */
235         Cell *retval;   /* return value */
236 };

238 #define NARGS   50      /* max args in a call */
158 #define NARGS   30

240 struct Frame *frame = NULL; /* base of stack frames; dynamically allocated */
241 int     nframe = 0;             /* number of frames allocated */
242 struct Frame *fp = NULL;        /* frame pointer. bottom level unused */

244 /*ARGSUSED*/
245 Cell *
246 call(Node **a, int n)   /* function call.  very kludgy and fragile */
166 call(Node **a, int n)
247 {
248         static Cell newcopycell =
249             { OCELL, CCOPY, 0, (uchar *) "", 0.0, NUM|STR|DONTFREE };
250         int i, ncall, ndef;
251         /* handles potential double freeing when fcn & param share a tempcell */
252         int freed = 0;
170         int i, ncall, ndef, freed = 0;
253         Node *x;
254         Cell *args[NARGS], *oargs[NARGS];       /* BUG: fixed size arrays */
255         Cell *y, *z, *fcn;
172         Cell *args[NARGS], *oargs[NARGS], *y, *z, *fcn;
256         uchar *s;

258         fcn = execute(a[0]);    /* the function itself */
259         s = fcn->nval;
260         if (!isfcn(fcn))
261                 FATAL("calling undefined function %s", s);
177         if (!isfunc(fcn))
```



```
178                 ERROR "calling undefined function %s", s FATAL;
262         if (frame == NULL) {
263                 fp = frame = (struct Frame *)calloc(nframe += 100,
264                     sizeof (struct Frame));
265                 if (frame == NULL) {
266                         FATAL("out of space for stack frames calling %s", s);
183                         ERROR "out of space for stack frames calling %s",
184                             s FATAL;
267                 }
268         }
269         for (ncall = 0, x = a[1]; x != NULL; x = x->nnext) /* args in call */
270                 ncall++;
271         ndef = (int)fcn->fval;                  /* args in defn */
272         dprintf(("calling %s, %d args (%d in defn), fp=%d\n",
273             s, ncall, ndef, (int)(fp-frame)));

191             s, ncall, ndef, fp-frame));
275         if (ncall > ndef) {
276                 WARNING("function %s called with %d args, uses only %d",
277                     s, ncall, ndef);
193                 ERROR "function %s called with %d args, uses only %d",
194                     s, ncall, ndef WARNING;
278         }
279         if (ncall + ndef > NARGS) {
280                 FATAL("function %s has %d arguments, limit %d",
281                     s, ncall + ndef, NARGS);
197                 ERROR "function %s has %d arguments, limit %d",
198                     s, ncall+ndef, NARGS FATAL;
282         }
283         for (i = 0, x = a[1]; x != NULL; i++, x = x->nnext) {
284                 /* get call args */
285                 dprintf(("evaluate args[%d], fp=%d:\n", i, (int)(fp-frame)));
202                 dprintf(("evaluate args[%d], fp=%d:\n", i, fp-frame));
286                 y = execute(x);
287                 oargs[i] = y;
288                 dprintf(("args[%d]: %s %f <%s>, t=%o\n",
289                     i, y->nval, y->fval,
290                     isarr(y) ? "(array)" : (char *)y->sval, y->tval));
291                 if (isfcn(y)) {
292                         FATAL("can't use function %s as argument in %s",
293                             y->nval, s);
208                 if (isfunc(y)) {
209                         ERROR "can't use function %s as argument in %s",
210                             y->nval, s FATAL;
294                 }
295                 if (isarr(y))
296                         args[i] = y;    /* arrays by ref */
297                 else
298                         args[i] = copycell(y);
299                 tempfree(y);
216                 tempfree(y, "callargs");
300         }
301         for (; i < ndef; i++) { /* add null args for ones not provided */
302                 args[i] = gettemp();
219                 args[i] = gettemp("nullargs");
303                 *args[i] = newcopycell;
304         }
305         fp++;   /* now ok to up frame */
306         if (fp >= frame + nframe) {
307                 int dfp = fp - frame;   /* old index */
308                 frame = (struct Frame *)
309                     realloc(frame, (nframe += 100) * sizeof (struct Frame));
310                 if (frame == NULL)
311                         FATAL("out of space for stack frames in %s", s);
228                         ERROR "out of space for stack frames in %s", s FATAL;
312                 fp = frame + dfp;
```

```
313         }
314         fp->fncell = fcn;
315         fp->args = args;
316         fp->nargs = ndef;       /* number defined with (excess are locals) */
317         fp->retval = gettemp();
234         fp->retval = gettemp("retval");

319         dprintf(("start exec of %s, fp=%d\n", s, (int)(fp-frame)));
236         dprintf(("start exec of %s, fp=%d\n", s, fp-frame));
320         /*LINTED align*/
321         y = execute((Node *)(fcn->sval));       /* execute body */
322         dprintf(("finished exec of %s, fp=%d\n", s, (int)(fp-frame)));
239         dprintf(("finished exec of %s, fp=%d\n", s, fp-frame));

324         for (i = 0; i < ndef; i++) {
325                 Cell *t = fp->args[i];
326                 if (isarr(t)) {
327                         if (t->csub == CCOPY) {
328                                 if (i >= ncall) {
329                                         freesymtab(t);
330                                         t->csub = CTEMP;
331                                         tempfree(t);
332 #endif /* ! codereview */
333                                 } else {
334                                         oargs[i]->tval = t->tval;
335                                         oargs[i]->tval &= ~(STR|NUM|DONTFREE);
336                                         oargs[i]->sval = t->sval;
337                                         tempfree(t);
248                                         tempfree(t, "oargsarr");
338                                 }
339                         }
340                 } else if (t != y) {    /* kludge to prevent freeing twice */
341                         t->csub = CTEMP;
342                         tempfree(t);
343                 } else if (t == y && t->csub == CCOPY) {
251                 } else {
344                         t->csub = CTEMP;
345                         tempfree(t);
346                         freed = 1;
253                         tempfree(t, "fp->args");
254                         if (t == y) freed = 1;
347                 }
348         }
349         tempfree(fcn);
257         tempfree(fcn, "call.fcn");
350         if (isexit(y) || isnext(y))
351                 return (y);
352         if (freed == 0) {
353                 tempfree(y);    /* don't free twice! */
354         }
260         if (!freed)
261                 tempfree(y, "fcn ret"); /* this can free twice! */
355         z = fp->retval;                 /* return value */
356         dprintf(("%s returns %g |%s| %o\n",
357             s, getfval(z), getsval(z), z->tval));
358         fp--;
359         return (z);
360 }

362 static Cell *
363 copycell(Cell *x)       /* make a copy of a cell in a temp */
364 {
365         Cell *y;

367         y = gettemp();
274         y = gettemp("copycell");
```



```
368         y->csub = CCOPY;        /* prevents freeing until call is over */
369         y->nval = x->nval;      /* BUG? */
370         if (isstr(x))
371                 y->sval = tostring(x->sval);
276         y->nval = x->nval;
277         y->sval = x->sval ? tostring(x->sval) : NULL;
372         y->fval = x->fval;
373         /* copy is not constant or field is DONTFREE right? */
374         y->tval = x->tval & ~(CON|FLD|REC|DONTFREE);
375         return (y);
376 }

284 /*ARGSUSED*/
378 Cell *
379 arg(Node **a, int n)            /* nth argument of a function */
286 arg(Node **a, int nnn)
380 {
288         int n;

382         n = ptoi(a[0]); /* argument number, counting from 0 */
290         n = (int)a[0];  /* argument number, counting from 0 */
383         dprintf(("arg(%d), fp->nargs=%d\n", n, fp->nargs));
384         if (n + 1 > fp->nargs) {
385                 FATAL("argument #%d of function %s was not supplied",
386                     n + 1, fp->fncell->nval);
292         if (n+1 > fp->nargs) {
293                 ERROR "argument #%d of function %s was not supplied",
294                     n+1, fp->fncell->nval FATAL;
387         }
388         return (fp->args[n]);
389 }

391 Cell *
392 jump(Node **a, int n)   /* break, continue, next, nextfile, return */
300 jump(Node **a, int n)
393 {
394         Cell *y;
302         register Cell *y;

396         switch (n) {
397         case EXIT:
398                 if (a[0] != NULL) {
399                         y = execute(a[0]);
400                         errorflag = (int)getfval(y);
401                         tempfree(y);
309                         tempfree(y, "");
402                 }
403                 longjmp(env, 1);
404                 /*NOTREACHED*/
405         case RETURN:
406                 if (a[0] != NULL) {
407                         y = execute(a[0]);
408                         if ((y->tval & (STR|NUM)) == (STR|NUM)) {
409                                 (void) setsval(fp->retval, getsval(y));
410                                 fp->retval->fval = getfval(y);
411                                 fp->retval->tval |= NUM;
412                         } else if (y->tval & STR)
413                                 (void) setsval(fp->retval, getsval(y));
414                         else if (y->tval & NUM)
415                                 (void) setfval(fp->retval, getfval(y));
416                         else            /* can't happen */
417                                 FATAL("bad type variable %d", y->tval);
418                         tempfree(y);
324                         tempfree(y, "");
419                 }
420                 return (jret);
```

```
421         case NEXT:
422                 return (jnext);
423         case NEXTFILE:
424                 nextfile();
425                 return (jnextfile);
426 #endif /* ! codereview */
427         case BREAK:
428                 return (jbreak);
429         case CONTINUE:
430                 return (jcont);
431         default:        /* can't happen */
432                 FATAL("illegal jump type %d", n);
329                 ERROR "illegal jump type %d", n FATAL;
433         }
434         /*NOTREACHED*/
435         return (NULL);
436 }

438 /*
439  * get next line from specific input
440  *
441  * a[0] is variable, a[1] is operator, a[2] is filename
442  */
443 #endif /* ! codereview */
444 Cell *
445 awkgetline(Node **a, int n)
335 getaline(Node **a, int n)
446 {
447         Cell *r, *x;
448         extern Cell **fldtab;
449         FILE *fp;
337         /* a[0] is variable, a[1] is operator, a[2] is filename */
338         register Cell *r, *x;
450         uchar *buf;
451         size_t bufsize = record_size;
452         int mode;

454         if ((buf = (uchar *)malloc(bufsize)) == NULL)
455                 FATAL("out of memory in getline");
340         FILE *fp;
341         size_t len;

457         (void) fflush(stdout);  /* in case someone is waiting for a prompt */
458         r = gettemp();
344         r = gettemp("");
459         if (a[1] != NULL) {             /* getline < file */
460                 x = execute(a[2]);              /* filename */
461                 mode = ptoi(a[1]);
462                 if (mode == '|')        /* input pipe */
463                         mode = LE;      /* arbitrary flag */
464                 fp = openfile(mode, getsval(x));
465                 tempfree(x);
347                 if ((int)a[1] == '|')   /* input pipe */
348                         a[1] = (Node *)LE;      /* arbitrary flag */
349                 fp = openfile((int)a[1], getsval(x));
350                 tempfree(x, "");
351                 buf = NULL;
466                 if (fp == NULL)
467                         n = -1;
468                 else
469                         n = readrec(&buf, &bufsize, fp);
355                         n = readrec(&buf, &len, fp);
356                 if (n > 0) {
470                 if (a[0] != NULL) {     /* getline var <file */
471                         x = execute(a[0]);
472                         (void) setsval(x, buf);
```



```
473                         tempfree(x);
358                         (void) setsval(execute(a[0]), buf);
474                 } else {                        /* getline <file */
475                         (void) setsval(fldtab[0], buf);
476                         if (is_number(fldtab[0]->sval)) {
477                                 fldtab[0]->fval = atof((char *)fldtab[0]->sval);
478                                 fldtab[0]->tval |= NUM;
360                                 if (!(recloc->tval & DONTFREE))
361                                         xfree(recloc->sval);
362                                 expand_buf(&record, &record_size, len);
363                                 (void) memcpy(record, buf, len);
364                                 record[len] = '\0';
365                                 recloc->sval = record;
366                                 recloc->tval = REC | STR | DONTFREE;
367                                 donerec = 1; donefld = 0;
479                         }
480                 }
370                 if (buf != NULL)
371                         free(buf);
481         } else {                        /* bare getline; use current input */
482                 if (a[0] == NULL)       /* getline */
483                         n = getrec(&record, &record_size, 1);
374                         n = getrec(&record, &record_size);
484                 else {                  /* getline var */
485                         n = getrec(&buf, &bufsize, 0);
486                         x = execute(a[0]);
487                         (void) setsval(x, buf);
488                         tempfree(x);
376                         init_buf(&buf, &len, LINE_INCR);
377                         n = getrec(&buf, &len);
378                         (void) setsval(execute(a[0]), buf);
379                         free(buf);
489                 }
490         }
491         (void) setfval(r, (Awkfloat)n);
492         free(buf);
493 #endif /* ! codereview */
494         return (r);
495 }

497 /*ARGSUSED*/
498 Cell *
499 getnf(Node **a, int n)          /* get NF */
383 getnf(Node **a, int n)
500 {
501         if (donefld == 0)
502                 fldbld();
503         return ((Cell *)a[0]);
504 }

506 /*ARGSUSED*/
507 Cell *
508 array(Node **a, int n)  /* a[0] is symtab, a[1] is list of subscripts */
392 array(Node **a, int n)
509 {
510         Cell *x, *y, *z;
511         uchar *s;
512         Node *np;
394         register Cell *x, *y, *z;
395         register uchar *s;
396         register Node *np;
513         uchar *buf;
514         size_t bufsz = record_size;
515         size_t nsub = strlen((char *)*SUBSEP);

517         if ((buf = (uchar *)malloc(bufsz)) == NULL)
```

```
518                 FATAL("out of memory in array");
398         size_t  bsize, tlen, len, slen;

520         x = execute(a[0]);      /* Cell* for symbol table */
401         init_buf(&buf, &bsize, LINE_INCR);
521         buf[0] = '\0';
403         tlen = 0;
404         slen = strlen((char *)*SUBSEP);
522         for (np = a[1]; np; np = np->nnext) {
523                 y = execute(np);        /* subscript */
524                 s = getsval(y);
525                 if (!adjbuf(&buf, &bufsz, strlen((char *)buf) +
526                     strlen((char *)s) + nsub + 1, record_size, 0, "array"))
527                         FATAL("out of memory for %s[%s...]", x->nval, buf);
528                 (void) strcat((char *)buf, (char *)s);
529                 if (np->nnext)
530                         (void) strcat((char *)buf, (char *)*SUBSEP);
531                 tempfree(y);
408                 len = strlen((char *)s);
409                 expand_buf(&buf, &bsize, tlen + len + slen);
410                 (void) memcpy(&buf[tlen], s, len);
411                 tlen += len;
412                 if (np->nnext) {
413                         (void) memcpy(&buf[tlen], *SUBSEP, slen);
414                         tlen += slen;
415                 }
416                 buf[tlen] = '\0';
417                 tempfree(y, "");
532         }
533         if (!isarr(x)) {
534                 dprintf(("making %s into an array\n", x->nval));
535                 if (freeable(x))
536                         xfree(x->sval);
537                 x->tval &= ~(STR|NUM|DONTFREE);
538                 x->tval |= ARR;
539                 x->sval = (uchar *)makesymtab(NSYMTAB);
425                 x->sval = (uchar *) makesymtab(NSYMTAB);
540         }
541         /*LINTED align*/
542         z = setsymtab(buf, (uchar *)"", 0.0, STR|NUM, (Array *)x->sval);
543         z->ctype = OCELL;
544         z->csub = CVAR;
545         tempfree(x);
431         tempfree(x, "");
546         free(buf);
547         return (z);
548 }

550 /*ARGSUSED*/
551 Cell *
552 awkdelete(Node **a, int n) /* a[0] is symtab, a[1] is list of subscripts */
438 delete(Node **a, int n)
553 {
554         Cell *x, *y;
555         Node *np;
556         uchar *s;
557         size_t nsub = strlen((char *)*SUBSEP);
442         uchar *buf, *s;
443         size_t bsize, tlen, slen, len;

559         x = execute(a[0]);      /* Cell* for symbol table */
560         if (!isarr(x))
561                 return (True);
562         if (a[1] == 0) {        /* delete the elements, not the table */
563                 freesymtab(x);
564                 x->tval &= ~STR;
```



```
565                 x->tval |= ARR;
566                 x->sval = (uchar *)makesymtab(NSYMTAB);
567         } else {
568                 size_t bufsz = record_size;
569                 uchar *buf;
570                 if ((buf = (uchar *)malloc(bufsz)) == NULL)
571                         FATAL("out of memory in delete");
447                 return (true);
448         init_buf(&buf, &bsize, LINE_INCR);
572                 buf[0] = '\0';
450         tlen = 0;
451         slen = strlen((char *)*SUBSEP);
573                 for (np = a[1]; np; np = np->nnext) {
574                         y = execute(np);        /* subscript */
575                         s = getsval(y);
576                         if (!adjbuf(&buf, &bufsz, strlen((char *)buf) +
577                             strlen((char *)s) + nsub + 1, record_size, 0,
578                             "awkdelete")) {
579                                 FATAL("out of memory deleting %s[%s...]",
580                                     x->nval, buf);
581                         }
582                         (void) strcat((char *)buf, (char *)s);
583                         if (np->nnext)
584                                 (void) strcat((char *)buf, (char *)*SUBSEP);
585                         tempfree(y);
455                 len = strlen((char *)s);
456                 expand_buf(&buf, &bsize, tlen + len + slen);
457                 (void) memcpy(&buf[tlen], s, len);
458                 tlen += len;
459                 if (np->nnext) {
460                         (void) memcpy(&buf[tlen], *SUBSEP, slen);
461                         tlen += slen;
462                 }
463                 buf[tlen] = '\0';
464                 tempfree(y, "");
586                 }
587                 freeelem(x, buf);
467         tempfree(x, "");
588                 free(buf);
589         }
590         tempfree(x);
591         return (True);
469         return (true);
592 }

594 /*ARGSUSED*/
595 Cell *
596 intest(Node **a, int n) /* a[0] is index (list), a[1] is symtab */
474 intest(Node **a, int n)
597 {
598         Cell *x, *ap, *k;
476         register Cell *x, *ap, *k;
599         Node *p;
600         uchar *buf;
601         uchar *s;
602         size_t bufsz = record_size;
603         size_t nsub = strlen((char *)*SUBSEP);
480         size_t bsize, tlen, slen, len;

605         ap = execute(a[1]);     /* array name */
606         if (!isarr(ap)) {
607                 dprintf(("making %s into an array\n", ap->nval));
608                 if (freeable(ap))
609                         xfree(ap->sval);
610                 ap->tval &= ~(STR|NUM|DONTFREE);
611                 ap->tval |= ARR;
```

```
612                 ap->sval = (uchar *)makesymtab(NSYMTAB);
613         }
614         if ((buf = (uchar *)malloc(bufsz)) == NULL) {
615                 FATAL("out of memory in intest");
616         }
483         if (!isarr(ap))
484                 ERROR "%s is not an array", ap->nval FATAL;
485         init_buf(&buf, &bsize, LINE_INCR);
617         buf[0] = 0;
487         tlen = 0;
488         slen = strlen((char *)*SUBSEP);
618         for (p = a[0]; p; p = p->nnext) {
619                 x = execute(p); /* expr */
620                 s = getsval(x);
621                 if (!adjbuf(&buf, &bufsz, strlen((char *)buf) +
622                     strlen((char *)s) + nsub + 1, record_size, 0, "intest"))
623                         FATAL("out of memory deleting %s[%s...]", x->nval, buf);
624                 (void) strcat((char *)buf, (char *)s);
625                 tempfree(x);
626                 if (p->nnext)
627                         (void) strcat((char *)buf, (char *)*SUBSEP);
492                 len = strlen((char *)s);
493                 expand_buf(&buf, &bsize, tlen + len + slen);
494                 (void) memcpy(&buf[tlen], s, len);
495                 tlen += len;
496                 tempfree(x, "");
497                 if (p->nnext) {
498                         (void) memcpy(&buf[tlen], *SUBSEP, slen);
499                         tlen += slen;
500                 }
501                 buf[tlen] = '\0';
628         }
629         /*LINTED align*/
630         k = lookup(buf, (Array *)ap->sval);
631         tempfree(ap);
505         tempfree(ap, "");
632         free(buf);
633         if (k == NULL)
634                 return (False);
508                 return (false);
635         else
636                 return (True);
510                 return (true);
637 }


639 Cell *
640 matchop(Node **a, int n)        /* ~ and match() */
515 matchop(Node **a, int n)
641 {
642         Cell *x, *y;
643         uchar *s, *t;
644         int i;
517         register Cell *x, *y;
518         register uchar *s, *t;
519         register int i;
645         fa *pfa;
646         int (*mf)(fa *, const uchar *) = match, mode = 0;
521         int (*mf)() = match, mode = 0;

648         if (n == MATCHFCN) {
649                 mf = pmatch;
650                 mode = 1;
651         }
652         x = execute(a[1]);      /* a[1] = target text */
527         x = execute(a[1]);
```



```
653         s = getsval(x);
654         if (a[0] == 0)          /* a[1] == 0: already-compiled reg expr */
655                 i = (*mf)((fa *)a[2], s);
529         if (a[0] == 0)
530                 i = (*mf)(a[2], s);
656         else {
657                 y = execute(a[2]);      /* a[2] = regular expr */
532                 y = execute(a[2]);
658                 t = getsval(y);
659                 pfa = makedfa(t, mode);
660                 i = (*mf)(pfa, s);
661                 tempfree(y);
536                 tempfree(y, "");
662         }
663         tempfree(x);
538         tempfree(x, "");
664         if (n == MATCHFCN) {
665                 int start = patbeg - s + 1;
666                 if (patlen < 0)
667                         start = 0;
668                 (void) setfval(rstartloc, (Awkfloat)start);
669                 (void) setfval(rlengthloc, (Awkfloat)patlen);
670                 x = gettemp();
545                 x = gettemp("");
671                 x->tval = NUM;
672                 x->fval = start;
673                 return (x);
674         } else if ((n == MATCH && i == 1) || (n == NOTMATCH && i == 0))
675                 return (True);
549         } else if (n == MATCH && i == 1 || n == NOTMATCH && i == 0)
550                 return (true);
676         else
677                 return (False);
552                 return (false);
678 }


680 Cell *
681 boolop(Node **a, int n)         /* a[0] || a[1], a[0] && a[1], !a[0] */
557 boolop(Node **a, int n)
682 {
683         Cell *x, *y;
684         int i;
559         register Cell *x, *y;
560         register int i;

686         x = execute(a[0]);
687         i = istrue(x);
688         tempfree(x);
564         tempfree(x, "");
689         switch (n) {
690         case BOR:
691                 if (i)
692                         return (True);
568                         return (true);
693                 y = execute(a[1]);
694                 i = istrue(y);
695                 tempfree(y);
696                 return (i ? True : False);
571                 tempfree(y, "");
572                 return (i ? true : false);
697         case AND:
698                 if (!i)
699                         return (False);
575                         return (false);
700                 y = execute(a[1]);
```

```
                i = istrue(y);
                tempfree(y);
                return (i ? True : False);
                tempfree(y, "");
                return (i ? true : false);
        case NOT:
                return (i ? False : True);
                return (i ? false : true);
        default:        /* can't happen */
                FATAL("unknown boolean operator %d", n);
                ERROR "unknown boolean operator %d", n FATAL;
        }
        /*NOTREACHED*/
        return (NULL);
}

Cell *
relop(Node **a, int n)          /* a[0 < a[1], etc. */
relop(Node **a, int n)
{
        int i;
        Cell *x, *y;
        register int i;
        register Cell *x, *y;
        Awkfloat j;

        x = execute(a[0]);
        y = execute(a[1]);
        if (x->tval & NUM && y->tval & NUM) {
        if (x->tval&NUM && y->tval&NUM) {
                j = x->fval - y->fval;
                i = j < 0 ? -1: (j > 0 ? 1: 0);
        } else {
                i = strcmp((char *)getsval(x), (char *)getsval(y));
        }
        tempfree(x);
        tempfree(y);
        tempfree(x, "");
        tempfree(y, "");
        switch (n) {
        case LT:        return (i < 0 ? True : False);
        case LE:        return (i <= 0 ? True : False);
        case NE:        return (i != 0 ? True : False);
        case EQ:        return (i == 0 ? True : False);
        case GE:        return (i >= 0 ? True : False);
        case GT:        return (i > 0 ? True : False);
        case LT:        return (i < 0 ? true : false);
        case LE:        return (i <= 0 ? true : false);
        case NE:        return (i != 0 ? true : false);
        case EQ:        return (i == 0 ? true : false);
        case GE:        return (i >= 0 ? true : false);
        case GT:        return (i > 0 ? true : false);
        default:        /* can't happen */
                FATAL("unknown relational operator %d", n);
                ERROR "unknown relational operator %d", n FATAL;
        }
        /*NOTREACHED*/
        return (False);
        return (false);
}

static void
tfree(Cell *a)          /* free a tempcell */
tfree(Cell *a, char *s)
{
        if (freeable(a)) {
```



```
                dprintf(("freeing %s %s %o\n", (char *)a->nval,
                    (char *)a->sval, a->tval));
                xfree(a->sval);
        if (dbg > 1) {
                (void) printf("## tfree %.8s %06lo %s\n",
                    s, (ulong_t)a, a->sval ? a->sval : (uchar *)"");
        }
        if (freeable(a))
                xfree(a->sval);
        if (a == tmps)
                FATAL("tempcell list is curdled");
                ERROR "tempcell list is curdled" FATAL;
        a->cnext = tmps;
        tmps = a;
}

static Cell *
gettemp(void)           /* get a tempcell */
gettemp(char *s)
{
        int i;
        Cell *x;
        register Cell *x;

        if (!tmps) {
                tmps = (Cell *)calloc(100, sizeof (Cell));
                if (!tmps)
                        FATAL("out of space for temporaries");
                        ERROR "no space for temporaries" FATAL;
                for (i = 1; i < 100; i++)
                        tmps[i - 1].cnext = &tmps[i];
                tmps[i - 1].cnext = 0;
                        tmps[i-1].cnext = &tmps[i];
                tmps[i-1].cnext = 0;
        }
        x = tmps;
        tmps = x->cnext;
        *x = tempcell;
        if (dbg > 1)
                (void) printf("## gtemp %.8s %06lo\n", s, (ulong_t)x);
        return (x);
}

/*ARGSUSED*/
Cell *
indirect(Node **a, int n)       /* $( a[0] ) */
indirect(Node **a, int n)
{
        Awkfloat val;
        Cell *x;
        int m;
        uchar *s;
        register Cell *x;
        register int m;
        register uchar *s;

        x = execute(a[0]);
        /* freebsd: defend against super large field numbers */
        val = getfval(x);
        if ((Awkfloat)INT_MAX < val)
                FATAL("trying to access out of range field %s", x->nval);
        m = (int)val;
        m = (int)getfval(x);
        if (m == 0 && !is_number(s = getsval(x)))       /* suspicion! */
                FATAL("illegal field $(%s), name \"%s\"", s, x->nval);
                /* BUG: can x->nval ever be null??? */
```

```
796         tempfree(x);
668                 ERROR "illegal field $(%s)", s FATAL;
669         tempfree(x, "");
797         x = fieldadr(m);
798         x->ctype = OCELL;       /* BUG?  why are these needed? */
671         x->ctype = OCELL;
799         x->csub = CFLD;
800         return (x);
801 }

803 /*ARGSUSED*/
804 Cell *
805 substr(Node **a, int nnn)       /* substr(a[0], a[1], a[2]) */
678 substr(Node **a, int nnn)
806 {
807         int k, m, n;
808         uchar *s;
680         register int k, m, n;
681         register uchar *s;
809         int temp;
810         Cell *x, *y, *z = 0;
683         register Cell *x, *y, *z;

812         x = execute(a[0]);
813         y = execute(a[1]);
814         if (a[2] != 0)
815                 z = execute(a[2]);
816         s = getsval(x);
817         k = strlen((char *)s) + 1;
818         if (k <= 1) {
819                 tempfree(x);
820                 tempfree(y);
821                 if (a[2] != 0) {
822                         tempfree(z);
823                 }
824                 x = gettemp();
692                 tempfree(x, "");
693                 tempfree(y, "");
694                 if (a[2] != 0)
695                         tempfree(z, "");
696                 x = gettemp("");
825                 (void) setsval(x, (uchar *)"");
826                 return (x);
827         }
828         m = (int)getfval(y);
829         if (m <= 0)
830                 m = 1;
831         else if (m > k)
832                 m = k;
833         tempfree(y);
705         tempfree(y, "");
834         if (a[2] != 0) {
835                 n = (int)getfval(z);
836                 tempfree(z);
708                 tempfree(z, "");
837         } else
838                 n = k - 1;
839         if (n < 0)
840                 n = 0;
841         else if (n > k - m)
842                 n = k - m;
843         dprintf(("substr: m=%d, n=%d, s=%s\n", m, n, s));
844         y = gettemp();
716         y = gettemp("");
845         temp = s[n + m - 1];    /* with thanks to John Linderman */
846         s[n + m - 1] = '\0';
```



```
847         (void) setsval(y, s + m - 1);
848         s[n + m - 1] = temp;
849         tempfree(x);
721         tempfree(x, "");
850         return (y);
851 }

853 /*ARGSUSED*/
854 Cell *
855 sindex(Node **a, int nnn)       /* index(a[0], a[1]) */
727 sindex(Node **a, int nnn)
856 {
857         Cell *x, *y, *z;
858         uchar *s1, *s2, *p1, *p2, *q;
729         register Cell *x, *y, *z;
730         register uchar *s1, *s2, *p1, *p2, *q;
859         Awkfloat v = 0.0;

861         x = execute(a[0]);
862         s1 = getsval(x);
863         y = execute(a[1]);
864         s2 = getsval(y);

866         z = gettemp();
738         z = gettemp("");
867         for (p1 = s1; *p1 != '\0'; p1++) {
868                 for (q = p1, p2 = s2; *p2 != '\0' && *q == *p2; q++, p2++)
869                         ;
870                 if (*p2 == '\0') {
871                         v = (Awkfloat)(p1 - s1 + 1);    /* origin 1 */
743                         v = (Awkfloat) (p1 - s1 + 1);   /* origin 1 */
872                         break;
873                 }
874         }
875         tempfree(x);
876         tempfree(y);
747         tempfree(x, "");
748         tempfree(y, "");
877         (void) setfval(z, v);
878         return (z);
879 }

881 #define MAXNUMSIZE      50

883 /*
884  * printf-like conversions
885  */
886 int
887 format(uchar **bufp, size_t *pbufsize, const uchar *s, Node *a)
753 void
754 format(uchar **bufp, uchar *s, Node *a)
888 {
889         uchar *fmt;
890         uchar *p, *t;
891         const uchar *os;
892         Cell *x;
893         int flag = 0, n;
894         int fmtwd; /* format width */
895         size_t fmtsz = record_size;
896         uchar_t *buf = *bufp;
897         size_t bufsize = *pbufsize;
757         register uchar *os;
758         register Cell *x;
759         int flag = 0, len;
760         uchar_t *buf;
761         size_t bufsize, fmtsize, cnt, tcnt, ret;
```

```
763         init_buf(&buf, &bufsize, LINE_INCR);
764         init_buf(&fmt, &fmtsize, LINE_INCR);
899         os = s;
900         p = buf;
901         if ((fmt = (uchar *)malloc(fmtsz)) == NULL)
902                 FATAL("out of memory in format()");
766         cnt = 0;
903         while (*s) {
904                 (void) adjbuf(&buf, &bufsize, MAXNUMSIZE + 1 + p - buf,
905                     record_size, &p, "format1");
906 #endif /* ! codereview */
907                 if (*s != '%') {
908                         *p++ = *s++;
768                         expand_buf(&buf, &bufsize, cnt);
769                         buf[cnt++] = *s++;
909                         continue;
910                 }
911                 if (*(s+1) == '%') {
912                         *p++ = '%';
773                         expand_buf(&buf, &bufsize, cnt);
774                         buf[cnt++] = '%';
913                         s += 2;
914                         continue;
915                 }
916                 /*
917                  * have to be real careful in case this is a huge number,
918                  * eg, %100000d
919                  */
920                 fmtwd = atoi((char *)s + 1);
921                 if (fmtwd < 0)
922                         fmtwd = -fmtwd;
923                 (void) adjbuf(&buf, &bufsize, fmtwd + 1 + p - buf,
924                     record_size, &p, "format2");
925                 for (t = fmt; (*t++ = *s) != '\0'; s++) {
926                         if (!adjbuf(&fmt, &fmtsz, MAXNUMSIZE + 1 + t - fmt,
927                             record_size, &t, "format3"))
928                                 FATAL(
929                         "format item %.30s... ran format() out of memory", os);
930                         if (isalpha((uchar)*s) && *s != 'l' && *s != 'h' &&
931                             *s != 'L')
778                 for (tcnt = 0; ; s++) {
779                         expand_buf(&fmt, &fmtsize, tcnt);
780                         fmt[tcnt++] = *s;
781                         if (*s == '\0')
782                                 break;
783                         if (isalpha(*s) && *s != 'l' && *s != 'h' && *s != 'L')
932                                 break;  /* the ansi panoply */
933                         if (*s == '*') {
786                                 if (a == NULL) {
787                                         ERROR
788                 "not enough args in printf(%s) or sprintf(%s)", os, os FATAL;
789                                 }
934                                 x = execute(a);
935                                 a = a->nnext;
936                                 (void) sprintf((char *)t - 1, "%d",
937                                     fmtwd = (int)getfval(x));
938                                 if (fmtwd < 0)
939                                         fmtwd = -fmtwd;
940                                 (void) adjbuf(&buf, &bufsize, fmtwd + 1 + p -
941                                     buf, record_size, &p, "format");
942                                 t = fmt + strlen((char *)fmt);
943                                 tempfree(x);
792                                 tcnt--;
793                                 expand_buf(&fmt, &fmtsize, tcnt + 12);
794                                 ret = sprintf((char *)&fmt[tcnt], "%d",
```



```
795                                     (int)getfval(x));
796                                 tcnt += ret;
797                                 tempfree(x, "");
944                         }
945                 }
946                 *t = '\0';
947                 if (fmtwd < 0)
948                         fmtwd = -fmtwd;
949                 (void) adjbuf(&buf, &bufsize, fmtwd + 1 + p - buf,
950                     record_size, &p, "format4");
800                 fmt[tcnt] = '\0';

952                 switch (*s) {
953                 case 'f': case 'e': case 'g': case 'E': case 'G':
954                         flag = 'f';
804                         flag = 1;
955                         break;
956                 case 'd': case 'i':
957                         flag = 'd';
807                         flag = 2;
958                         if (*(s-1) == 'l')
959                                 break;
960                         *(t-1) = 'l';
961                         *t = 'd';
962                         *++t = '\0';
810                         fmt[tcnt - 1] = 'l';
811                         expand_buf(&fmt, &fmtsize, tcnt);
812                         fmt[tcnt++] = 'd';
813                         fmt[tcnt] = '\0';
963                         break;
964                 case 'o': case 'x': case 'X': case 'u':
965                         flag = *(s-1) == 'l' ? 'd' : 'u';
816                         flag = *(s-1) == 'l' ? 2 : 3;
966                         break;
967                 case 's':
968                         flag = 's';
819                         flag = 4;
969                         break;
970                 case 'c':
971                         flag = 'c';
822                         flag = 5;
972                         break;
973                 default:
974                         WARNING("weird printf conversion %s", fmt);
975                         flag = '?';
825                         flag = 0;
976                         break;
977                 }
828                 if (flag == 0) {
829                         len = strlen((char *)fmt);
830                         expand_buf(&buf, &bufsize, cnt + len);
831                         (void) memcpy(&buf[cnt], fmt, len);
832                         cnt += len;
833                         buf[cnt] = '\0';
834                         continue;
835                 }
978                 if (a == NULL) {
979                         FATAL(
980         "not enough args in printf(%s)", os);
837                         ERROR
838         "not enough args in printf(%s) or sprintf(%s)", os, os FATAL;
981                 }
982                 x = execute(a);
983                 a = a->nnext;
984                 n = MAXNUMSIZE;
985                 if (fmtwd > n)
```

```
 986                     n = fmtwd;
 987             (void) adjbuf(&buf, &bufsize, 1 + n + p - buf, record_size,
 988                 &p, "format5");

 842             for (;;) {
 843                     /* make sure we have at least 1 byte space */
 844                     expand_buf(&buf, &bufsize, cnt + 1);
 845                     len = bufsize - cnt;
 990             switch (flag) {
 991             case '?':
 992                     /* unknown, so dump it too */
 993                     /* LINTED E_SEC_SPRINTF_UNBOUNDED_COPY */
 994                     (void) sprintf((char *)p, "%s", (char *)fmt);
 995                     t = getsval(x);
 996                     n = strlen((char *)t);
 997                     if (fmtwd > n)
 998                             n = fmtwd;
 999                     (void) adjbuf(&buf, &bufsize, 1 + strlen((char *)p) +
1000                         n + p - buf, record_size, &p, "format6");
1001                     p += strlen((char *)p);
1002                     /* LINTED E_SEC_SPRINTF_UNBOUNDED_COPY */
1003                     (void) sprintf((char *)p, "%s", t);
1004                     break;
1005             case 'f':
1006                     /* LINTED E_SEC_PRINTF_VAR_FMT */
1007                     (void) sprintf((char *)p, (char *)fmt, getfval(x));
1008                     break;
1009             case 'd':
1010                     /* LINTED E_SEC_PRINTF_VAR_FMT */
1011                     (void) sprintf((char *)p, (char *)fmt,
1012                         (long) getfval(x));
1013                     break;
1014             case 'u':
1015                     /* LINTED E_SEC_PRINTF_VAR_FMT */
1016                     (void) sprintf((char *)p, (char *)fmt,
1017                         (int) getfval(x));
 847                     case 1:
 848                             /*LINTED*/
 849                             ret = snprintf((char *)&buf[cnt], len,
 850                                 (char *)fmt, getfval(x));
 851                             break;
 852                     case 2:
 853                             /*LINTED*/
 854                             ret = snprintf((char *)&buf[cnt], len,
 855                                 (char *)fmt, (long)getfval(x));
 856                             break;
 857                     case 3:
 858                             /*LINTED*/
 859                             ret = snprintf((char *)&buf[cnt], len,
 860                                 (char *)fmt, (int)getfval(x));
 861                             break;
 862                     case 4:
 863                             /*LINTED*/
 864                             ret = snprintf((char *)&buf[cnt], len,
 865                                 (char *)fmt, getsval(x));
1018                     break;
1019             case 's':
1020                     t = getsval(x);
1021                     n = strlen((char *)t);
1022                     if (fmtwd > n)
1023                             n = fmtwd;
1024                     if (!adjbuf(&buf, &bufsize, 1+n+p-buf, record_size,
1025                         &p, "format7"))
1026                             FATAL(
1027             "huge string/format (%d chars) in printf %.30s... "
1028             "ran format() out of memory", n, t);
```



```
1029                     /* LINTED E_SEC_PRINTF_VAR_FMT */
1030                     (void) sprintf((char *)p, (char *)fmt, t);
1031                     break;
1032             case 'c':
 867                     case 5:
1033                     if (isnum(x)) {
1034                             if (getfval(x))
1035                                     /* LINTED E_SEC_PRINTF_VAR_FMT */
1036                                     (void) sprintf((char *)p, (char *)fmt,
1037                                         (int) getfval(x));
1038                             else {
1039                                     *p++ = '\0'; /* explicit null byte */
1040                                     /* next output will start here */
1041                                     *p = '\0';
1042                             }
 869                                     /*LINTED*/
 870                                     ret = snprintf((char *)&buf[cnt], len,
 871                                         (char *)fmt, (int)getfval(x));
1043                     } else {
1044                             /* LINTED E_SEC_PRINTF_VAR_FMT */
1045                             (void) sprintf((char *)p, (char *)fmt,
1046                                 getsval(x)[0]);
 873                                     /*LINTED*/
 874                                     ret = snprintf((char *)&buf[cnt], len,
 875                                         (char *)fmt, getsval(x)[0]);
1047                     }
1048                     break;
1049             default:
1050                     FATAL("can't happen: bad conversion %c in format()",
1051                         flag);
 879                             ret = 0;
1052             }
1053             tempfree(x);
1054             p += strlen((char *)p);
 881                     if (ret < len)
 882                             break;
 883                     expand_buf(&buf, &bufsize, cnt + ret);
 884             }
 885             tempfree(x, "");
 886             cnt += ret;
1055             s++;
1056     }
1057     *p = '\0';
1058     free(fmt);
 889     buf[cnt] = '\0';
1059     for (; a; a = a->nnext) /* evaluate any remaining args */
1060             (void) execute(a);
1061     *bufp = buf;
1062     *pbufsize = bufsize;
1063     return (p - buf);
 892     *bufp = tostring(buf);
 893     free(buf);
 894     free(fmt);
1064 }

1066 /*ARGSUSED*/
1067 Cell *
1068 awksprintf(Node **a, int n)     /* sprintf(a[0]) */
 899 a_sprintf(Node **a, int n)
1069 {
1070     Cell *x;
1071     Node *y;
 901     register Cell *x;
 902     register Node *y;
1072     uchar *buf;
1073     size_t bufsz = 3 * record_size;
```

```
#endif /* ! codereview */

        if ((buf = (uchar *)malloc(bufsz)) == NULL)
                FATAL("out of memory in awksprintf");
#endif /* ! codereview */
        y = a[0]->nnext;
        x = execute(a[0]);
        if (format(&buf, &bufsz, getsval(x), y) == -1)
                FATAL("sprintf string %.30s... too long.  can't happen.", buf);
        tempfree(x);
        x = gettemp();
 904    format(&buf, getsval(x), y);
 905    tempfree(x, "");
 906    x = gettemp("");
        x->sval = buf;
        x->tval = STR;
        return (x);
}

/*ARGSUSED*/
Cell *
awkprintf(Node **a, int n)      /* printf */
{       /* a[0] is list of args, starting with format string */
        /* a[1] is redirection operator, a[2] is redirection file */
 914 aprintf(Node **a, int n)
 915 {
        FILE *fp;
        Cell *x;
        Node *y;
 917    register Cell *x;
 918    register Node *y;
        uchar *buf;
        int len;
        size_t bufsz = 3 * record_size;
#endif /* ! codereview */

        if ((buf = (uchar *)malloc(bufsz)) == NULL)
                FATAL("out of memory in awkprintf");
#endif /* ! codereview */
        y = a[0]->nnext;
        x = execute(a[0]);
        if ((len = format(&buf, &bufsz, getsval(x), y)) == -1)
                FATAL("printf string %.30s... too long.  can't happen.", buf);
        tempfree(x);
        if (a[1] == NULL) {
                /* fputs(buf, stdout); */
                (void) fwrite((char *)buf, len, 1, stdout);
                if (ferror(stdout))
                        FATAL("write error on stdout");
        } else {
                fp = redirect(ptoi(a[1]), a[2]);
                /* fputs(buf, fp); */
                (void) fwrite(buf, len, 1, fp);
 920    format(&buf, getsval(x), y);
 921    tempfree(x, "");
 922    if (a[1] == NULL)
 923            (void) fputs((char *)buf, stdout);
 924    else {
 925            fp = redirect((int)a[1], a[2]);
 926            (void) fputs((char *)buf, fp);
                (void) fflush(fp);
                if (ferror(fp))
                        FATAL("write error on %s", filename(fp));
#endif /* ! codereview */
        }
        free(buf);
```



```
        return (True);
 928    return (true);
}

Cell *
arith(Node **a, int n)  /* a[0] + a[1], etc.  also -a[0] */
 932 arith(Node **a, int n)
{
        Awkfloat i, j = 0;
 934    Awkfloat i, j;
        double v;
        Cell *x, *y, *z;
 936    register Cell *x, *y, *z;

        x = execute(a[0]);
        i = getfval(x);
        tempfree(x);
 940    tempfree(x, "");
        if (n != UMINUS) {
                y = execute(a[1]);
                j = getfval(y);
                tempfree(y);
 944            tempfree(y, "");
        }
        z = gettemp();
 946    z = gettemp("");
        switch (n) {
        case ADD:
                i += j;
                break;
        case MINUS:
                i -= j;
                break;
        case MULT:
                i *= j;
                break;
        case DIVIDE:
                if (j == 0)
                        FATAL("division by zero");
 959                    ERROR "division by zero" FATAL;
                i /= j;
                break;
        case MOD:
                if (j == 0)
                        FATAL("division by zero in mod");
 964                    ERROR "division by zero in mod" FATAL;
                (void) modf(i/j, &v);
                i = i - j * v;
                break;
        case UMINUS:
                i = -i;
                break;
        case POWER:
                if (j >= 0 && modf(j, &v) == 0.0) /* pos integer exponent */
                        i = ipow(i, (int)j);
                else
                        i = errcheck(pow(i, j), "pow");
                break;
        default:        /* can't happen */
                FATAL("illegal arithmetic operator %d", n);
 978            ERROR "illegal arithmetic operator %d", n FATAL;
        }
        (void) setfval(z, i);
        return (z);
}
```

```
static double
ipow(double x, int n)   /* x**n.  ought to be done by pow, but isn't always */
ipow(double x, int n)
{
        double v;

        if (n <= 0)
                return (1.0);
        v = ipow(x, n / 2);
        v = ipow(x, n/2);
        if (n % 2 == 0)
                return (v * v);
        else
                return (x * v * v);
}

Cell *
incrdecr(Node **a, int n)       /* a[0]++, etc. */
incrdecr(Node **a, int n)
{
        Cell *x, *z;
        int k;
        register Cell *x, *z;
        register int k;
        Awkfloat xf;

        x = execute(a[0]);
        xf = getfval(x);
        k = (n == PREINCR || n == POSTINCR) ? 1 : -1;
        if (n == PREINCR || n == PREDECR) {
                (void) setfval(x, xf + k);
                return (x);
        }
        z = gettemp();
        z = gettemp("");
        (void) setfval(z, xf);
        (void) setfval(x, xf + k);
        tempfree(x);
        tempfree(x, "");
        return (z);
}

Cell *
assign(Node **a, int n) /* a[0] = a[1], a[0] += a[1], etc. */
{               /* this is subtle; don't muck with it. */
        Cell *x, *y;
assign(Node **a, int n)
{
        register Cell *x, *y;
        Awkfloat xf, yf;
        double v;

        y = execute(a[1]);
        x = execute(a[0]);
        x = execute(a[0]);      /* order reversed from before... */
        if (n == ASSIGN) {      /* ordinary assignment */
                /* self-assignment: */
                /* LINTED E_NOP_IF_STMT */
                if (x == y && !(x->tval & (FLD|REC)))
                        /* leave alone unless it's a field */
                        ;
                else if ((y->tval & (STR|NUM)) == (STR|NUM)) {
                if ((y->tval & (STR|NUM)) == (STR|NUM)) {
                        (void) setsval(x, getsval(y));
                        x->fval = getfval(y);
                        x->tval |= NUM;
```



```
                } else if (isstr(y))
                } else if (y->tval & STR)
                        (void) setsval(x, getsval(y));
                else if (isnum(y))
                else if (y->tval & NUM)
                        (void) setfval(x, getfval(y));
                else
                        funnyvar(y, "read value of");
                tempfree(y);
                tempfree(y, "");
                return (x);
        }
        xf = getfval(x);
        yf = getfval(y);
        switch (n) {
        case ADDEQ:
                xf += yf;
                break;
        case SUBEQ:
                xf -= yf;
                break;
        case MULTEQ:
                xf *= yf;
                break;
        case DIVEQ:
                if (yf == 0)
                        FATAL("division by zero in /=");
                        ERROR "division by zero in /=" FATAL;
                xf /= yf;
                break;
        case MODEQ:
                if (yf == 0)
                        FATAL("division by zero in %%=");
                        ERROR "division by zero in %%=" FATAL;
                (void) modf(xf/yf, &v);
                xf = xf - yf * v;
                break;
        case POWEQ:
                if (yf >= 0 && modf(yf, &v) == 0.0) /* pos integer exponent */
                        xf = ipow(xf, (int)yf);
                else
                        xf = errcheck(pow(xf, yf), "pow");
                break;
        default:
                FATAL("illegal assignment operator %d", n);
                ERROR "illegal assignment operator %d", n FATAL;
                break;
        }
        tempfree(y);
        tempfree(y, "");
        (void) setfval(x, xf);
        return (x);
}

/*ARGSUSED*/
Cell *
cat(Node **a, int q)            /* a[0] cat a[1] */
cat(Node **a, int q)
{
        Cell *x, *y, *z;
        int n1, n2;
        uchar *s;
        register Cell *x, *y, *z;
        register int n1, n2;
        register uchar *s;
```

```
1291         x = execute(a[0]);
1292         y = execute(a[1]);
1293         (void) getsval(x);
1294         (void) getsval(y);
1295         n1 = strlen((char *)x->sval);
1296         n2 = strlen((char *)y->sval);
1297         s = (uchar *)malloc(n1 + n2 + 1);
1298         if (s == NULL) {
1299                 FATAL("out of space concatenating %.15s... and %.15s...",
1300                     x->sval, y->sval);
1096                 ERROR "out of space concatenating %.15s and %.15s",
1097                     x->sval, y->sval FATAL;
1301         }
1302         (void) strcpy((char *)s, (char *)x->sval);
1303         (void) strcpy((char *)s + n1, (char *)y->sval);
1304         tempfree(x);
1305         tempfree(y);
1306         z = gettemp();
1101         tempfree(y, "");
1102         z = gettemp("");
1307         z->sval = s;
1308         z->tval = STR;
1105         tempfree(x, "");
1309         return (z);
1310 }

1312 /*ARGSUSED*/
1313 Cell *
1314 pastat(Node **a, int n)         /* a[0] { a[1] } */
1111 pastat(Node **a, int n)
1315 {
1316         Cell *x;
1113         register Cell *x;

1318         if (a[0] == 0)
1319                 x = execute(a[1]);
1320         else {
1321                 x = execute(a[0]);
1322                 if (istrue(x)) {
1323                         tempfree(x);
1120                         tempfree(x, "");
1324                         x = execute(a[1]);
1325                 }
1326         }
1327         return (x);
1328 }

1330 /*ARGSUSED*/
1331 Cell *
1332 dopa2(Node **a, int n)          /* a[0], a[1] { a[2] } */
1129 dopa2(Node **a, int n)
1333 {
1334         Cell *x;
1335         int pair;
1133         static int      *pairstack = NULL;

1135         if (!pairstack) {
1136                 /* first time */
1137                 dprintf(("paircnt: %d\n", paircnt));
1138                 pairstack = (int *)malloc(sizeof (int) * paircnt);
1139                 if (!pairstack)
1140                         ERROR "out of space in dopa2" FATAL;
1141                 (void) memset(pairstack, 0, sizeof (int) * paircnt);
1142         }

1337         pair = ptoi(a[3]);
```



```
1144         pair = (int)a[3];
1338         if (pairstack[pair] == 0) {
1339                 x = execute(a[0]);
1340                 if (istrue(x))
1341                         pairstack[pair] = 1;
1342                 tempfree(x);
1149                 tempfree(x, "");
1343         }
1344         if (pairstack[pair] == 1) {
1345                 x = execute(a[1]);
1346                 if (istrue(x))
1347                         pairstack[pair] = 0;
1348                 tempfree(x);
1155                 tempfree(x, "");
1349                 x = execute(a[2]);
1350                 return (x);
1351         }
1352         return (False);
1159         return (false);
1353 }

1355 /*ARGSUSED*/
1356 Cell *
1357 split(Node **a, int nnn)        /* split(a[0], a[1], a[2]); a[3] is type */
1164 split(Node **a, int nnn)
1358 {
1359         Cell *x = 0, *y, *ap;
1360         uchar *s, *origs;
1361         int sep;
1362         uchar *t, temp, num[50], *fs = 0;
1363         int n, tempstat, arg3type;
1166         Cell *x, *y, *ap;
1167         register uchar *s;
1168         register int sep;
1169         uchar *t, temp, num[11], *fs;
1170         int n, tempstat;

1365         y = execute(a[0]);      /* source string */
1366         origs = s = (uchar *)strdup((char *)getsval(y));
1367         arg3type = ptoi(a[3]);
1173         s = getsval(y);
1368         if (a[2] == 0)          /* fs string */
1369                 fs = *FS;
1370         else if (arg3type == STRING) {  /* split(str,arr,"string") */
1176         else if ((int)a[3] == STRING) { /* split(str,arr,"string") */
1371                 x = execute(a[2]);
1372                 fs = getsval(x);
1373         } else if (arg3type == REGEXPR)
1179         } else if ((int)a[3] == REGEXPR)
1374                 fs = (uchar *)"(regexpr)";      /* split(str,arr,/regexpr/) */
1375         else
1376                 FATAL("illegal type of split");
1182                 ERROR "illegal type of split()" FATAL;
1377         sep = *fs;
1378         ap = execute(a[1]);     /* array name */
1379         freesymtab(ap);
1380         dprintf(("split: s=|%s|, a=%s, sep=|%s|\n", s, ap->nval, fs));
1381         ap->tval &= ~STR;
1382         ap->tval |= ARR;
1383         ap->sval = (uchar *)makesymtab(NSYMTAB);

1385         n = 0;
1386         if (arg3type == REGEXPR && strlen((char *)((fa *)a[2])->restr) == 0) {
1387                 arg3type = 0;
1388                 fs = (uchar *)"";
1389                 sep = 0;
```

```
1390         }
1391         if (*s != '\0' && (strlen((char *)fs) > 1 || arg3type == REGEXPR)) {
1392         if (*s != '\0' && strlen((char *)fs) > 1 || (int)a[3] == REGEXPR) {
1392                 /* reg expr */
1393                 fa *pfa;
1394                 if (arg3type == REGEXPR) {      /* it's ready already */
1395                 if ((int)a[3] == REGEXPR) {     /* it's ready already */
1395                         pfa = (fa *)a[2];
1396                 } else {
1397                         pfa = makedfa(fs, 1);
1398                 }
1399                 if (nematch(pfa, s)) {
1400                         tempstat = pfa->initstat;
1401                         pfa->initstat = 2;
1402                         do {
1403                                 n++;
1404                                 (void) sprintf((char *)num, "%d", n);
1405                                 temp = *patbeg;
1406                                 *patbeg = '\0';
1407                                 if (is_number(s)) {
1408                                         (void) setsymtab(num, s,
1409                                             atof((char *)s),
1410                                             /*LINTED align*/
1411                                             STR|NUM, (Array *)ap->sval);
1412                                 } else {
1413                                         (void) setsymtab(num, s, 0.0,
1414                                             /*LINTED align*/
1415                                             STR, (Array *)ap->sval);
1416                                 }
1417                                 *patbeg = temp;
1418                                 s = patbeg + patlen;
1419                                 if (*(patbeg+patlen-1) == 0 || *s == 0) {
1420                                         n++;
1421                                         (void) sprintf((char *)num, "%d", n);
1422                                         (void) setsymtab(num, (uchar *)"", 0.0,
1423                                             /*LINTED align*/
1424                                             STR, (Array *)ap->sval);
1425                                         pfa->initstat = tempstat;
1426                                         goto spdone;
1427                                 }
1428                         } while (nematch(pfa, s));
1429                         /* bwk: has to be here to reset */
1430                         pfa->initstat = tempstat;
1431                         /* cf gsub and refldbld */
1432 #endif /* ! codereview */
1433                 }
1434                 n++;
1435                 (void) sprintf((char *)num, "%d", n);
1436                 if (is_number(s)) {
1437                         (void) setsymtab(num, s, atof((char *)s),
1438                             /*LINTED align*/
1439                             STR|NUM, (Array *)ap->sval);
1440                 } else {
1441                         /*LINTED align*/
1442                         (void) setsymtab(num, s, 0.0, STR, (Array *)ap->sval);
1443                 }
1444 spdone:
1445                 pfa = NULL;
1446         } else if (sep == ' ') {
1447                 for (n = 0; ; ) {
1448                         while (*s == ' ' || *s == '\t' || *s == '\n')
1449                                 s++;
1450                         if (*s == 0)
1451                                 break;
1452                         n++;
1453                         t = s;
```



```
1454                         do
1455                                 s++;
1456                         while (*s != ' ' && *s != '\t' &&
1457                             *s != '\n' && *s != '\0')
1458                                 ;
1459                         temp = *s;
1460                         *s = '\0';
1461                         (void) sprintf((char *)num, "%d", n);
1462                         if (is_number(t)) {
1463                                 (void) setsymtab(num, t, atof((char *)t),
1464                                     /*LINTED align*/
1465                                     STR|NUM, (Array *)ap->sval);
1466                         } else {
1467                                 (void) setsymtab(num, t, 0.0,
1468                                     /*LINTED align*/
1469                                     STR, (Array *)ap->sval);
1470                         }
1471                         *s = temp;
1472                         if (*s != 0)
1473                                 s++;
1474                 }
1475         } else if (sep == 0) {  /* new: split(s, a, "") => 1 char/elem */
1476                 for (n = 0; *s != 0; s++) {
1477                         uchar buf[2];
1478                         n++;
1479                         (void) sprintf((char *)num, "%d", n);
1480                         buf[0] = *s;
1481                         buf[1] = 0;
1482                         if (isdigit(buf[0])) {
1483                                 (void) setsymtab(num, buf, atof((char *)buf),
1484                                     /* LINTED align */
1485                                     STR | NUM, (Array *)ap->sval);
1486                         } else {
1487                                 (void) setsymtab(num, buf, 0.0,
1488                                     /* LINTED align */
1489                                     STR, (Array *)ap->sval);
1490                         }
1491                 }
1492 #endif /* ! codereview */
1493         } else if (*s != 0) {
1494                 for (;;) {
1495                         n++;
1496                         t = s;
1497                         while (*s != sep && *s != '\n' && *s != '\0')
1498                                 s++;
1499                         temp = *s;
1500                         *s = '\0';
1501                         (void) sprintf((char *)num, "%d", n);
1502                         if (is_number(t)) {
1503                                 (void) setsymtab(num, t, atof((char *)t),
1504                                     /*LINTED align*/
1505                                     STR|NUM, (Array *)ap->sval);
1506                         } else {
1507                                 (void) setsymtab(num, t, 0.0,
1508                                     /*LINTED align*/
1509                                     STR, (Array *)ap->sval);
1510                         }
1511                         *s = temp;
1512                         if (*s++ == 0)
1513                                 break;
1514                 }
1515         }
1516         tempfree(ap);
1517         tempfree(y);
1518         free(origs);
1519         if (a[2] != 0 && arg3type == STRING) {
```

```
1520                    tempfree(x);
1521            }
1522            x = gettemp();
1230            tempfree(ap, "");
1231            tempfree(y, "");
1232            if (a[2] != 0 && (int)a[3] == STRING)
1233                    tempfree(x, "");
1234            x = gettemp("");
1523            x->tval = NUM;
1524            x->fval = n;
1525            return (x);
1526 }

1528 /*ARGSUSED*/
1529 Cell *
1530 condexpr(Node **a, int n)       /* a[0] ? a[1] : a[2] */
1242 condexpr(Node **a, int n)
1531 {
1532            Cell *x;
1244            register Cell *x;

1534            x = execute(a[0]);
1535            if (istrue(x)) {
1536                    tempfree(x);
1248                    tempfree(x, "");
1537                    x = execute(a[1]);
1538            } else {
1539                    tempfree(x);
1251                    tempfree(x, "");
1540                    x = execute(a[2]);
1541            }
1542            return (x);
1543 }

1545 /*ARGSUSED*/
1546 Cell *
1547 ifstat(Node **a, int n)         /* if (a[0]) a[1]; else a[2] */
1259 ifstat(Node **a, int n)
1548 {
1549            Cell *x;
1261            register Cell *x;

1551            x = execute(a[0]);
1552            if (istrue(x)) {
1553                    tempfree(x);
1265                    tempfree(x, "");
1554                    x = execute(a[1]);
1555            } else if (a[2] != 0) {
1556                    tempfree(x);
1268                    tempfree(x, "");
1557                    x = execute(a[2]);
1558            }
1559            return (x);
1560 }

1562 /*ARGSUSED*/
1563 Cell *
1564 whilestat(Node **a, int n)      /* while (a[0]) a[1] */
1276 whilestat(Node **a, int n)
1565 {
1566            Cell *x;
1278            register Cell *x;

1568            for (;;) {
1569                    x = execute(a[0]);
1570                    if (!istrue(x))
```



```
1571                            return (x);
1572                    tempfree(x);
1284                    tempfree(x, "");
1573                    x = execute(a[1]);
1574                    if (isbreak(x)) {
1575                            x = True;
1287                            x = true;
1576                            return (x);
1577                    }
1578                    if (isnext(x) || isexit(x) || isret(x))
1579                            return (x);
1580                    tempfree(x);
1292                    tempfree(x, "");
1581            }
1582 }

1584 /*ARGSUSED*/
1585 Cell *
1586 dostat(Node **a, int n)         /* do a[0]; while(a[1]) */
1298 dostat(Node **a, int n)
1587 {
1588            Cell *x;
1300            register Cell *x;

1590            for (;;) {
1591                    x = execute(a[0]);
1592                    if (isbreak(x))
1593                            return (True);
1305                            return (true);
1594                    if (isnext(x) || isexit(x) || isret(x))
1595                            return (x);
1596                    tempfree(x);
1308                    tempfree(x, "");
1597                    x = execute(a[1]);
1598                    if (!istrue(x))
1599                            return (x);
1600                    tempfree(x);
1312                    tempfree(x, "");
1601            }
1602 }

1604 /*ARGSUSED*/
1605 Cell *
1606 forstat(Node **a, int n)        /* for (a[0]; a[1]; a[2]) a[3] */
1318 forstat(Node **a, int n)
1607 {
1608            Cell *x;
1320            register Cell *x;

1610            x = execute(a[0]);
1611            tempfree(x);
1323            tempfree(x, "");
1612            for (;;) {
1613                    if (a[1] != 0) {
1614                            x = execute(a[1]);
1615                            if (!istrue(x))
1616                                    return (x);
1617                            else
1618                                    tempfree(x);
1330                                    tempfree(x, "");
1619                    }
1620                    x = execute(a[3]);
1621                    if (isbreak(x))         /* turn off break */
1622                            return (True);
1334                            return (true);
1623                    if (isnext(x) || isexit(x) || isret(x))
```

```
1624                        return (x);
1625                tempfree(x);
1337                tempfree(x, "");
1626                x = execute(a[2]);
1627                tempfree(x);
1339                tempfree(x, "");
1628        }
1629 }

1631 /*ARGSUSED*/
1632 Cell *
1633 instat(Node **a, int n) /* for (a[0] in a[1]) a[2] */
1345 instat(Node **a, int n)
1634 {
1635        Cell *x, *vp, *arrayp, *cp, *ncp;
1347        register Cell *x, *vp, *arrayp, *cp, *ncp;
1636        Array *tp;
1637        int i;

1639        vp = execute(a[0]);
1640        arrayp = execute(a[1]);
1641        if (!isarr(arrayp)) {
1642                return (True);
1643        }
1353        if (!isarr(arrayp))
1354                ERROR "%s is not an array", arrayp->nval FATAL;
1644        /*LINTED align*/
1645        tp = (Array *)arrayp->sval;
1646        tempfree(arrayp);
1357        tempfree(arrayp, "");
1647        for (i = 0; i < tp->size; i++) { /* this routine knows too much */
1648                for (cp = tp->tab[i]; cp != NULL; cp = ncp) {
1649                        (void) setsval(vp, cp->nval);
1650                        ncp = cp->cnext;
1651                        x = execute(a[2]);
1652                        if (isbreak(x)) {
1653                                tempfree(vp);
1654                                return (True);
1364                                tempfree(vp, "");
1365                                return (true);
1655                        }
1656                        if (isnext(x) || isexit(x) || isret(x)) {
1657                                tempfree(vp);
1368                                tempfree(vp, "");
1658                                return (x);
1659                        }
1660                        tempfree(x);
1371                        tempfree(x, "");
1661                }
1662        }
1663        return (True);
1374        return (true);
1664 }

1666 /*
1667  * builtin functions. a[0] is type, a[1] is arg list
1668  */
1669 #endif /* ! codereview */
1670 /*ARGSUSED*/
1671 Cell *
1672 bltin(Node **a, int n)
1673 {
1674        Cell *x, *y;
1377        register Cell *x, *y;
1675        Awkfloat u;
1676        int t;
```



```
1677        Awkfloat tmp;
1379        register int t;
1678        uchar *p, *buf;
1679        Node *nextarg;
1680        FILE *fp;
1681 #endif /* ! codereview */

1683        t = ptoi(a[0]);
1382        t = (int)a[0];
1684        x = execute(a[1]);
1685        nextarg = a[1]->nnext;
1686        switch (t) {
1687        case FLENGTH:
1688                if (isarr(x)) {
1689                        /* GROT.  should be function */
1690                        /* LINTED align */
1691                        u = (Awkfloat)((Array *)x->sval)->nelem;
1692                } else {
1693                        u = (Awkfloat)strlen((char *)getsval(x));
1694                }
1695                break;
1387                u = (Awkfloat)strlen((char *)getsval(x)); break;
1696        case FLOG:
1697                u = errcheck(log(getfval(x)), "log"); break;
1698        case FINT:
1699                (void) modf(getfval(x), &u); break;
1700        case FEXP:
1701                u = errcheck(exp(getfval(x)), "exp"); break;
1702        case FSQRT:
1703                u = errcheck(sqrt(getfval(x)), "sqrt"); break;
1704        case FSIN:
1705                u = sin(getfval(x)); break;
1706        case FCOS:
1707                u = cos(getfval(x)); break;
1708        case FATAN:
1709                if (nextarg == 0) {
1710                        WARNING("atan2 requires two arguments; returning 1.0");
1402                        ERROR "atan2 requires two arguments; returning 1.0"
1403                            WARNING;
1711                        u = 1.0;
1712                } else {
1713                        y = execute(a[1]->nnext);
1714                        u = atan2(getfval(x), getfval(y));
1715                        tempfree(y);
1408                        tempfree(y, "");
1716                        nextarg = nextarg->nnext;
1717                }
1718                break;
1719        case FSYSTEM:
1720                /* in case something is buffered already */
1721                (void) fflush(stdout);
1722                /* 256 is unix-dep */
1723                u = (Awkfloat)system((char *)getsval(x)) / 256;
1724                break;
1725        case FRAND:
1726                /* in principle, rand() returns something in 0..RAND_MAX */
1727                u = (Awkfloat)(rand() % RAND_MAX) / RAND_MAX;
1419                u = (Awkfloat)(rand() % 32767) / 32767.0;
1728                break;
1729        case FSRAND:
1730                if (isrec(x))   /* no argument provided */
1422                if (x->tval & REC)      /* no argument provided */
1731                        u = time((time_t *)0);
1732                else
1733                        u = getfval(x);
1734                tmp = u;
```

```
1735                    srand((unsigned int)u);
1736                    u = srand_seed;
1737                    srand_seed = tmp;
1426                    srand((int)u); u = (int)u;
1738                    break;
1739            case FTOUPPER:
1740            case FTOLOWER:
1741                    buf = tostring(getsval(x));
1742                    if (t == FTOUPPER) {
1743                            for (p = buf; *p; p++)
1744                                    if (islower(*p))
1745                                            *p = toupper(*p);
1746                    } else {
1747                            for (p = buf; *p; p++)
1748                                    if (isupper(*p))
1749                                            *p = tolower(*p);
1750                    }
1751                    tempfree(x);
1752                    x = gettemp();
1440                    tempfree(x, "");
1441                    x = gettemp("");
1753                    (void) setsval(x, buf);
1754                    free(buf);
1755                    return (x);
1756            case FFLUSH:
1757                    if (isrec(x) || strlen((char *)getsval(x)) == 0) {
1758                            /* fflush() or fflush("") -> all */
1759                            flush_all();
1760                            u = 0;
1761                    } else if ((fp = openfile(FFLUSH, getsval(x))) == NULL)
1762                            u = EOF;
1763                    else
1764                            u = fflush(fp);
1765                    break;
1766 #endif /* ! codereview */
1767            default:        /* can't happen */
1768                    FATAL("illegal function type %d", t);
1445                    ERROR "illegal function type %d", t FATAL;
1769                    break;
1770            }
1771            tempfree(x);
1772            x = gettemp();
1448            tempfree(x, "");
1449            x = gettemp("");
1773            (void) setfval(x, u);
1774            if (nextarg != 0) {
1775                    WARNING("warning: function has too many arguments");
1452                    ERROR "warning: function has too many arguments" WARNING;
1776                    for (; nextarg; nextarg = nextarg->nnext)
1777                            (void) execute(nextarg);
1778            }
1779            return (x);
1780 }

1782 /*ARGSUSED*/
1783 Cell *
1784 printstat(Node **a, int n)              /* print a[0] */
1461 print(Node **a, int n)
1785 {
1786            Node *x;
1787            Cell *y;
1463            register Node *x;
1464            register Cell *y;
1788            FILE *fp;

1790            if (a[1] == 0)  /* a[1] is redirection operator, a[2] is file */
```



```
1467            if (a[1] == 0)
1791                    fp = stdout;
1792            else
1793                    fp = redirect(ptoi(a[1]), a[2]);
1470                    fp = redirect((int)a[1], a[2]);
1794            for (x = a[0]; x != NULL; x = x->nnext) {
1795                    y = execute(x);
1796                    (void) fputs((char *)getsval(y), fp);
1797                    tempfree(y);
1474                    tempfree(y, "");
1798                    if (x->nnext == NULL)
1799                            (void) fputs((char *)*ORS, fp);
1800                    else
1801                            (void) fputs((char *)*OFS, fp);
1802            }
1803            if (a[1] != 0)
1804                    (void) fflush(fp);
1805            if (ferror(fp))
1806                    FATAL("write error on %s", filename(fp));
1807            return (True);
1482            return (true);
1808 }
______unchanged_portion_omitted_

1492 struct {
1493            FILE    *fp;
1494            uchar   *fname;
1495            int     mode;   /* '|', 'a', 'w' */
1496 } files[FOPEN_MAX];

1817 static FILE *
1818 redirect(int a, Node *b)        /* set up all i/o redirections */
1499 redirect(int a, Node *b)
1819 {
1820            FILE *fp;
1821            Cell *x;
1822            uchar *fname;

1824            x = execute(b);
1825            fname = getsval(x);
1826            fp = openfile(a, fname);
1827            if (fp == NULL)
1828                    FATAL("can't open file %s", fname);
1829            tempfree(x);
1509                    ERROR "can't open file %s", fname FATAL;
1510            tempfree(x, "");
1830            return (fp);
1831 }

1833 struct files {
1834            FILE    *fp;
1835            const uchar     *fname;
1836            int     mode;   /* '|', 'a', 'w' => LE/LT, GT */
1837 } *files;

1839 int nfiles;

1841 static void
1842 stdinit(void)   /* in case stdin, etc., are not constants */
1843 {
1844            nfiles = FOPEN_MAX;
1845            files = calloc(nfiles, sizeof (*files));
1846            if (files == NULL)
1847                    FATAL("can't allocate file memory for %u files", nfiles);
1848            files[0].fp = stdin;
1849            files[0].fname = (uchar *)"/dev/stdin";
```

```
1850         files[0].mode = LT;
1851         files[1].fp = stdout;
1852         files[1].fname = (uchar *)"/dev/stdout";
1853         files[1].mode = GT;
1854         files[2].fp = stderr;
1855         files[2].fname = (uchar *)"/dev/stderr";
1856         files[2].mode = GT;
1857 }

1859 #endif /* ! codereview */
1860 static FILE *
1861 openfile(int a, const uchar *us)
1514 openfile(int a, uchar *s)
1862 {
1863         const uchar *s = us;
1864         int i, m;
1865         FILE *fp = 0;
1516         register int i, m;
1517         register FILE *fp;

1867         if (*s == '\0')
1868                 FATAL("null file name in print or getline");
1869         for (i = 0; i < nfiles; i++) {
1520                 ERROR "null file name in print or getline" FATAL;
1521         for (i = 0; i < FOPEN_MAX; i++) {
1870                 if (files[i].fname &&
1871                     (strcmp((char *)s, (char *)files[i].fname) == 0)) {
1523                     strcmp((char *)s, (char *)files[i].fname) == 0) {
1872                         if (a == files[i].mode ||
1873                             (a == APPEND && files[i].mode == GT)) {
1525                             a == APPEND && files[i].mode == GT) {
1874                                 return (files[i].fp);
1875                         }
1876                         if (a == FFLUSH)
1877                                 return (files[i].fp);
1878 #endif /* ! codereview */
1879                 }
1880         }
1881         if (a == FFLUSH)        /* didn't find it, so don't create it! */
1882                 return (NULL);

1884         for (i = 0; i < nfiles; i++) {
1528         for (i = 0; i < FOPEN_MAX; i++) {
1885                 if (files[i].fp == 0)
1886                         break;
1887         }
1888         if (i >= nfiles) {
1889                 struct files *nf;
1890                 int nnf = nfiles + FOPEN_MAX;
1891                 nf = realloc(files, nnf * sizeof (*nf));
1892                 if (nf == NULL)
1893                         FATAL("cannot grow files for %s and %d files", s, nnf);
1894                 (void) memset(&nf[nfiles], 0, FOPEN_MAX * sizeof (*nf));
1895                 nfiles = nnf;
1896                 files = nf;
1897         }
1532         if (i >= FOPEN_MAX)
1533                 ERROR "%s makes too many open files", s FATAL;
1898         (void) fflush(stdout);  /* force a semblance of order */
1899         m = a;
1900         if (a == GT) {
1901                 fp = fopen((char *)s, "w");
1902         } else if (a == APPEND) {
1903                 fp = fopen((char *)s, "a");
1904                 m = GT; /* so can mix > and >> */
1905         } else if (a == '|') {  /* output pipe */
```



```
1906                 fp = popen((char *)s, "w");
1907         } else if (a == LE) {   /* input pipe */
1908                 fp = popen((char *)s, "r");
1909         } else if (a == LT) {   /* getline <file */
1910                 fp = strcmp((char *)s, "-") == 0 ?
1911                     stdin : fopen((char *)s, "r");      /* "-" is stdin */
1912         } else  /* can't happen */
1913                 FATAL("illegal redirection %d", a);
1549                 ERROR "illegal redirection" FATAL;
1914         if (fp != NULL) {
1915                 files[i].fname = tostring(s);
1916                 files[i].fp = fp;
1917                 files[i].mode = m;
1918         }
1919         return (fp);
1920 }

1922 static const char *
1923 filename(FILE *fp)
1924 {
1925         int i;

1927         for (i = 0; i < nfiles; i++) {
1928                 if (fp == files[i].fp)
1929                         return ((char *)files[i].fname);
1930         }

1932         return ("???");
1933 }

1935 #endif /* ! codereview */
1936 /*ARGSUSED*/
1937 Cell *
1938 closefile(Node **a, int n)
1939 {
1940         Cell *x;
1558         register Cell *x;
1941         int i, stat;

1943         x = execute(a[0]);
1944         (void) getsval(x);
1945         stat = -1;
1946         for (i = 0; i < nfiles; i++) {
1563         for (i = 0; i < FOPEN_MAX; i++) {
1947                 if (files[i].fname &&
1948                     strcmp((char *)x->sval, (char *)files[i].fname) == 0) {
1949                         if (ferror(files[i].fp)) {
1950                                 WARNING("i/o error occurred on %s",
1951                                     files[i].fname);
1567                                 ERROR "i/o error occurred on %s",
1568                                     files[i].fname WARNING;
1952                         }
1953                         if (files[i].mode == '|' || files[i].mode == LE)
1954                                 stat = pclose(files[i].fp);
1955                         else
1956                                 stat = fclose(files[i].fp);
1957                         if (stat == EOF) {
1958                                 WARNING("i/o error occurred closing %s",
1959                                     files[i].fname);
1575                                 ERROR "i/o error occurred closing %s",
1576                                     files[i].fname WARNING;
1960                         }
1961                         if (i > 2)      /* don't do /dev/std... */
1962 #endif /* ! codereview */
1963                                 xfree(files[i].fname);
1964                         /* watch out for ref thru this */
```

```
1965                            files[i].fname = NULL;
1966                            files[i].fp = NULL;
1967                    }
1968            }
1969            tempfree(x);
1970            x = gettemp();
1971            (void) setfval(x, (Awkfloat)stat);
1972            return (x);
1578            tempfree(x, "close");
1579            return (true);
1973 }

1975 static void
1976 closeall(void)
1977 {
1978            int i, stat;

1980            for (i = 0; i < FOPEN_MAX; i++) {
1981                    if (files[i].fp) {
1982                            if (ferror(files[i].fp)) {
1983                                    WARNING("i/o error occurred on %s",
1984                                        files[i].fname);
1590                                    ERROR "i/o error occurred on %s",
1591                                        files[i].fname WARNING;
1985                            }
1986                            if (files[i].mode == '|' || files[i].mode == LE)
1987                                    stat = pclose(files[i].fp);
1988                            else
1989                                    stat = fclose(files[i].fp);
1990                            if (stat == EOF) {
1991                                    WARNING("i/o error occurred while closing %s",
1992                                        files[i].fname);
1598                                    ERROR "i/o error occurred while closing %s",
1599                                        files[i].fname WARNING;
1993                            }
1994                    }
1995            }
1996 }

1998 static void
1999 flush_all(void)
2000 {
2001            int i;

2003            for (i = 0; i < nfiles; i++)
2004                    if (files[i].fp)
2005                            (void) fflush(files[i].fp);
2006 }

2008 void backsub(uchar **pb_ptr, uchar **sptr_ptr);

2010 #endif /* ! codereview */
2011 /*ARGSUSED*/
2012 Cell *
2013 sub(Node **a, int nnn)  /* substitute command */
1605 sub(Node **a, int nnn)
2014 {
2015            uchar *sptr, *pb, *q;
2016            Cell *x, *y, *result;
2017            uchar *t, *buf;
1607            register uchar *sptr;
1608            register Cell *x, *y, *result;
1609            uchar *buf, *t;
2018            fa *pfa;
2019            size_t  bufsz = record_size;
1611            size_t  bsize, cnt, len;
```



```
2021            if ((buf = (uchar *)malloc(bufsz)) == NULL)
2022                    FATAL("out of memory in sub");
2023 #endif /* ! codereview */
2024            x = execute(a[3]);      /* target string */
2025            t = getsval(x);
2026            if (a[0] == 0)          /* 0 => a[1] is already-compiled regexpr */
1613            if (a[0] == 0)
2027                    pfa = (fa *)a[1];       /* regular expression */
2028            else {
2029                    y = execute(a[1]);
2030                    pfa = makedfa(getsval(y), 1);
2031                    tempfree(y);
1618                    tempfree(y, "");
2032            }
2033            y = execute(a[2]);      /* replacement string */
2034            result = False;
1621            result = false;
2035            if (pmatch(pfa, t)) {
1623                    init_buf(&buf, &bsize, LINE_INCR);
1624                    cnt = 0;
2036                    sptr = t;
2037                    (void) adjbuf(&buf, &bufsz, 1+patbeg-sptr, record_size,
2038                        0, "sub");
2039                    pb = buf;
2040                    while (sptr < patbeg)
2041                            *pb++ = *sptr++;
1626                    len = patbeg - sptr;
1627                    if (len > 0) {
1628                            expand_buf(&buf, &bsize, cnt + len);
1629                            (void) memcpy(buf, sptr, len);
1630                            cnt += len;
1631                    }
2042                    sptr = getsval(y);
2043                    while (*sptr != 0) {
2044                            (void) adjbuf(&buf, &bufsz, 5 + pb - buf, record_size,
2045                                &pb, "sub");
2046                            if (*sptr == '\\') {
2047                                    backsub(&pb, &sptr);
1634                            expand_buf(&buf, &bsize, cnt);
1635                            if (*sptr == '\\' &&
1636                                (*(sptr+1) == '&' || *(sptr+1) == '\\')) {
1637                                    sptr++;         /* skip \, */
1638                                    buf[cnt++] = *sptr++; /* add & or \ */
2048                            } else if (*sptr == '&') {
1640                                    expand_buf(&buf, &bsize, cnt + patlen);
2049                                    sptr++;
2050                                    (void) adjbuf(&buf, &bufsz, 1+patlen+pb-buf,
2051                                        record_size, &pb, "sub");
2052                                    for (q = patbeg; q < patbeg + patlen; )
2053                                            *pb++ = *q++;
1642                                    (void) memcpy(&buf[cnt], patbeg, patlen);
1643                                    cnt += patlen;
2054                            } else {
2055                                    *pb++ = *sptr++;
1645                                    buf[cnt++] = *sptr++;
2056                            }
2057                    }
2058                    *pb = '\0';
2059                    if (pb > buf + bufsz)
2060                            FATAL("sub result1 %.30s too big; can't happen", buf);
2061 #endif /* ! codereview */
2062                    sptr = patbeg + patlen;
2063                    if ((patlen == 0 && *patbeg) || (patlen && *(sptr-1))) {
2064                            (void) adjbuf(&buf, &bufsz, 1 + strlen((char *)sptr) +
2065                                pb - buf, 0, &pb, "sub");
```

```
2066                        while ((*pb++ = *sptr++) != 0)
2067                                ;
1648                        len = strlen((char *)sptr);
1649                        expand_buf(&buf, &bsize, cnt + len);
1650                        (void) memcpy(&buf[cnt], sptr, len);
1651                        cnt += len;
2068                }
2069                if (pb > buf + bufsz)
2070                        FATAL("sub result2 %.30s too big; can't happen", buf);
2071                /* BUG: should be able to avoid copy */
1653                buf[cnt] = '\0';
2072                (void) setsval(x, buf);
2073                result = True;
2074        }
2075        tempfree(x);
2076        tempfree(y);
2077 #endif /* ! codereview */
2078        free(buf);
1655                result = true;
1656        }
1657        tempfree(x, "");
1658        tempfree(y, "");
2079        return (result);
2080 }

2082 /*ARGSUSED*/
2083 Cell *
2084 gsub(Node **a, int nnn)         /* global substitute */
1664 gsub(Node **a, int nnn)
2085 {
2086        Cell *x, *y;
2087        uchar *rptr, *sptr, *t, *pb, *q;
1666        register Cell *x, *y;
1667        register uchar *rptr, *sptr, *t;
2088        uchar *buf;
2089        fa *pfa;
1669        register fa *pfa;
2090        int mflag, tempstat, num;
2091        size_t bufsz = record_size;
1671        size_t  bsize, cnt, len;

2093        if ((buf = (uchar *)malloc(bufsz)) == NULL)
2094                FATAL("out of memory in gsub");
2095 #endif /* ! codereview */
2096        mflag = 0;      /* if mflag == 0, can replace empty string */
2097        num = 0;
2098        x = execute(a[3]);      /* target string */
2099        t = getsval(x);
2100        if (a[0] == 0)          /* 0 => a[1] is already-compiled regexpr */
2101                pfa = (fa *)a[1];       /* regular expression */
1673        if (a[0] == 0)
1674                pfa = (fa *) a[1];      /* regular expression */
2102        else {
2103                y = execute(a[1]);
2104                pfa = makedfa(getsval(y), 1);
2105                tempfree(y);
1678                tempfree(y, "");
2106        }
2107        y = execute(a[2]);      /* replacement string */
2108        if (pmatch(pfa, t)) {
2109                tempstat = pfa->initstat;
2110                pfa->initstat = 2;
2111                pb = buf;
1684                init_buf(&buf, &bsize, LINE_INCR);
2112                rptr = getsval(y);
1686                cnt = 0;
```



```
2113                do {
2114                        if (patlen == 0 && *patbeg != 0) {
2115                                /* matched empty string */
2116                                if (mflag == 0) {       /* can replace empty */
2117                                        num++;
2118                                        sptr = rptr;
2119                                        while (*sptr != 0) {
2120                                                (void) adjbuf(&buf, &bufsz, 5+pb-buf,
2121                                                    record_size, &pb, "gsub");
2122                                                if (*sptr == '\\') {
2123                                                        backsub(&pb, &sptr);
1694                                                    expand_buf(&buf, &bsize, cnt);
1695                                                    if (*sptr == '\\' &&
1696                                                        (*(sptr+1) == '&' ||
1697                                                        *(sptr+1) == '\\')) {
1698                                                            sptr++;
1699                                                            buf[cnt++] = *sptr++;
2124                                                } else if (*sptr == '&') {
1701                                                            expand_buf(&buf,
1702                                                                &bsize,
1703                                                                cnt + patlen);
2125                                                        sptr++;
2126                                                        (void) adjbuf(&buf, &bufsz, 1 +
2127                                                            patlen+pb-buf, record_size,
2128                                                            &pb, "gsub");
2129                                                for (q = patbeg; q < patbeg+patlen; )
2130                                                        *pb++ = *q++;
1705                                                        (void) memcpy(&buf[cnt],
1706                                                            patbeg, patlen);
1707                                                        cnt += patlen;
2131                                                } else {
2132                                                        *pb++ = *sptr++;
1709                                                        buf[cnt++] = *sptr++;
2133                                                }
2134                                                }
2135                                }
2136                                if (*t == 0)    /* at end */
2137                                        goto done;
2138                                (void) adjbuf(&buf, &bufsz, 2 + pb - buf,
2139                                    record_size, &pb, "gsub");
2140                                *pb++ = *t++;
2141                                /* BUG: not sure of this test */
2142                                if (pb > buf + bufsz)
2143                                        FATAL(
2144                        "gsub result0 %.30s too big; can't happen", buf);
1715                                expand_buf(&buf, &bsize, cnt);
1716                                buf[cnt++] = *t++;
2145                                mflag = 0;
2146                        } else {        /* matched nonempty string */
2147                                num++;
2148                                sptr = t;
2149                                (void) adjbuf(&buf, &bufsz, 1 +
2150                                    (patbeg - sptr) + pb - buf, record_size,
2151                                    &pb, "gsub");
2152                                while (sptr < patbeg)
2153                                        *pb++ = *sptr++;
1721                                len = patbeg - sptr;
1722                                if (len > 0) {
1723                                        expand_buf(&buf, &bsize, cnt + len);
1724                                        (void) memcpy(&buf[cnt], sptr, len);
1725                                        cnt += len;
1726                                }
2154                                sptr = rptr;
2155                                while (*sptr != 0) {
2156                                        (void) adjbuf(&buf, &bufsz, 5 + pb -
2157                                            buf, record_size, &pb, "gsub");
```

```
2158                                    if (*sptr == '\\') {
2159                                            backsub(&pb, &sptr);
1729                                    expand_buf(&buf, &bsize, cnt);
1730                                    if (*sptr == '\\' &&
1731                                        (*(sptr+1) == '&' ||
1732                                        *(sptr+1) == '\\')) {
1733                                            sptr++;
1734                                            buf[cnt++] = *sptr++;
2160                                    } else if (*sptr == '&') {
1736                                            expand_buf(&buf, &bsize,
1737                                                cnt + patlen);
2161                                            sptr++;
2162                                    (void) adjbuf(&buf, &bufsz, 1 +
2163                                        patlen + pb - buf, record_size,
2164                                        &pb, "gsub");
2165                                for (q = patbeg; q < patbeg + patlen; )
2166                                            *pb++ = *q++;
1739                                    (void) memcpy(&buf[cnt],
1740                                        patbeg, patlen);
1741                                    cnt += patlen;
2167                                    } else {
2168                                            *pb++ = *sptr++;
1743                                            buf[cnt++] = *sptr++;
2169                                    }
2170                                }
2171                                t = patbeg + patlen;
2172                                if (patlen == 0 || *t == 0 || *(t - 1) == 0)
1747                                if ((*(t-1) == 0) || (*t == 0))
2173                                        goto done;
2174                                if (pb > buf + bufsz)
2175                                        FATAL(
2176                        "gsub result1 %.30s too big; can't happen", buf);
2177 #endif /* ! codereview */
2178                                mflag = 1;
2179                                }
2180                } while (pmatch(pfa, t));
2181                sptr = t;
2182                (void) adjbuf(&buf, &bufsz, 1 + strlen((char *)sptr) + pb -
2183                    buf, 0, &pb, "gsub");
2184                while ((*pb++ = *sptr++) != 0)
2185                        ;
2186        done:   if (pb < buf + bufsz)
2187                        *pb = '\0';
2188                else if (*(pb-1) != '\0')
2189                        FATAL("gsub result2 %.30s truncated; can't happen",
2190                            buf);
2191                /* BUG: should be able to avoid copy + free */
1749                len = strlen((char *)sptr);
1750                expand_buf(&buf, &bsize, len + cnt);
1751                (void) memcpy(&buf[cnt], sptr, len);
1752                cnt += len;
1753        done:
1754                buf[cnt] = '\0';
2192                (void) setsval(x, buf);
1756                free(buf);
2193                pfa->initstat = tempstat;
2194        }
2195        tempfree(x);
2196        tempfree(y);
2197        x = gettemp();
1759        tempfree(x, "");
1760        tempfree(y, "");
1761        x = gettemp("");
2198        x->tval = NUM;
2199        x->fval = num;
2200        free(buf);
```



```
2201 #endif /* ! codereview */
2202         return (x);
2203 }

2205 void
2206 backsub(uchar **pb_ptr, uchar **sptr_ptr)       /* handle \\& variations */
2207 {                                       /* sptr[0] == '\\' */
2208         uchar *pb = *pb_ptr, *sptr = *sptr_ptr;

2210         if (sptr[1] == '\\') {
2211                 if (sptr[2] == '\\' && sptr[3] == '&') { /* \\\& -> \& */
2212                         *pb++ = '\\';
2213                         *pb++ = '&';
2214                         sptr += 4;
2215                 } else if (sptr[2] == '&') {    /* \\& -> \ + matched */
2216                         *pb++ = '\\';
2217                         sptr += 2;
2218                 } else {                        /* \\x -> \\x */
2219                         *pb++ = *sptr++;
2220                         *pb++ = *sptr++;
2221                 }
2222         } else if (sptr[1] == '&') {    /* literal & */
2223                 sptr++;
2224                 *pb++ = *sptr++;
2225         } else                          /* literal \ */
2226                 *pb++ = *sptr++;

2228         *pb_ptr = pb;
2229         *sptr_ptr = sptr;
2230 }
2231 #endif /* ! codereview */
```

```
**********************************************************
   14250 Sat May 11 15:20:46 2013
new/usr/src/cmd/awk/tran.c
3731 Update nawk to version 20121220
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright 2006 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * Copyright (C) Lucent Technologies 1997
 * All Rights Reserved
 *
 * Permission to use, copy, modify, and distribute this software and
 * its documentation for any purpose and without fee is hereby
 * granted, provided that the above copyright notice appear in all
 * copies and that both that the copyright notice and this
 * permission notice and warranty disclaimer appear in supporting
 * documentation, and that the name Lucent Technologies or any of
 * its entities not be used in advertising or publicity pertaining
 * to distribution of the software without specific, written prior
 * permission.
 *
 * LUCENT DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
 * INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS.
 * IN NO EVENT SHALL LUCENT OR ANY OF ITS ENTITIES BE LIABLE FOR ANY
 * SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
 * WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER
 * IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION,
 * ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF
 * THIS SOFTWARE.
 */
/*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
/*        All Rights Reserved   */

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

#define DEBUG
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include "awk.h"
```



```
#include "y.tab.h"

#define FULLTAB 2       /* rehash when table gets this x full */
#define GROWTAB 4       /* grow table by this factor */

Array   *symtab;        /* main symbol table */

uchar   **FS;           /* initial field sep */
uchar   **RS;           /* initial record sep */
uchar   **OFS;          /* output field sep */
uchar   **ORS;          /* output record sep */
uchar   **OFMT;         /* output format for numbers */
uchar   **CONVFMT;      /* format for conversions in getsval */
#endif /* ! codereview */
Awkfloat *NF;           /* number of fields in current record */
Awkfloat *NR;           /* number of current record */
Awkfloat *FNR;          /* number of current record in current file */
uchar   **FILENAME;     /* current filename argument */
Awkfloat *ARGC;         /* number of arguments from command line */
uchar   **SUBSEP;       /* subscript separator for a[i,j,k]; default \034 */
Awkfloat *RSTART;       /* start of re matched with ~; origin 1 (!) */
Awkfloat *RLENGTH;      /* length of same */

Cell    *fsloc;         /* FS */
Cell    *recloc;        /* location of record */
Cell    *nrloc;         /* NR */
Cell    *nfloc;         /* NF */
Cell    *fnrloc;        /* FNR */
Array   *ARGVtab;       /* symbol table containing ARGV[...] */
Array   *ENVtab;        /* symbol table containing ENVIRON[...] */
Cell    *rstartloc;     /* RSTART */
Cell    *rlengthloc;    /* RLENGTH */
Cell    *symtabloc;     /* SYMTAB */

Cell    *nullloc;       /* a guaranteed empty cell */
Cell    *nullloc;
Node    *nullnode;      /* zero&null, converted into a node for comparisons */
Cell    *literal0;
#endif /* ! codereview */

static  void    rehash(Array *);

void
syminit(void)           /* initialize symbol table with builtin vars */
syminit(void)
{
        literal0 = setsymtab((uchar *)"0", (uchar *)"0", 0.0,
        init_buf(&record, &record_size, LINE_INCR);

        /* initialize $0 */
        recloc = getfld(0);
        recloc->nval = (uchar *)"$0";
        recloc->sval = record;
        recloc->tval = REC|STR|DONTFREE;

        symtab = makesymtab(NSYMTAB);
        (void) setsymtab((uchar *)"0", (uchar *)"0", 0.0,
            NUM|STR|CON|DONTFREE, symtab);
        /* this is used for if(x)... tests: */
        nullloc = setsymtab((uchar *)"$zero&null", (uchar *)"", 0.0,
            NUM|STR|CON|DONTFREE, symtab);
        nullnode = celltonode(nullloc, CCON);

        fsloc = setsymtab((uchar *)"FS", (uchar *)" ", 0.0,
            STR|DONTFREE, symtab);
        FS = &fsloc->sval;
```

```
 78         nullnode = valtonode(nullloc, CCON);
 79         FS = &setsymtab((uchar *)"FS", (uchar *)" ", 0.0,
 80             STR|DONTFREE, symtab)->sval;
107         RS = &setsymtab((uchar *)"RS", (uchar *)"\n", 0.0,
108             STR|DONTFREE, symtab)->sval;
109         OFS = &setsymtab((uchar *)"OFS", (uchar *)" ", 0.0,
110             STR|DONTFREE, symtab)->sval;
111         ORS = &setsymtab((uchar *)"ORS", (uchar *)"\n", 0.0,
112             STR|DONTFREE, symtab)->sval;
113         OFMT = &setsymtab((uchar *)"OFMT", (uchar *)"%.6g", 0.0,
114             STR|DONTFREE, symtab)->sval;
115         CONVFMT = &setsymtab((uchar *)"CONVFMT", (uchar *)"%.6g", 0.0,
116             STR|DONTFREE, symtab)->sval;
117 #endif /* ! codereview */
118         FILENAME = &setsymtab((uchar *)"FILENAME", (uchar *)"-", 0.0,
119             STR|DONTFREE, symtab)->sval;
120         nfloc = setsymtab((uchar *)"NF", (uchar *)"", 0.0, NUM, symtab);
121         NF = &nfloc->fval;
122         nrloc = setsymtab((uchar *)"NR", (uchar *)"", 0.0, NUM, symtab);
123         NR = &nrloc->fval;
124         fnrloc = setsymtab((uchar *)"FNR", (uchar *)"", 0.0, NUM, symtab);
125         FNR = &fnrloc->fval;
126         SUBSEP = &setsymtab((uchar *)"SUBSEP", (uchar *)"\034", 0.0,
127             STR|DONTFREE, symtab)->sval;
128         rstartloc = setsymtab((uchar *)"RSTART", (uchar *)"", 0.0,
129             NUM, symtab);
130         RSTART = &rstartloc->fval;
131         rlengthloc = setsymtab((uchar *)"RLENGTH", (uchar *)"", 0.0,
132             NUM, symtab);
133         RLENGTH = &rlengthloc->fval;
134         symtabloc = setsymtab((uchar *)"SYMTAB", (uchar *)"", 0.0, ARR, symtab);
135         symtabloc->sval = (uchar *)symtab;
136 }

138 void
139 arginit(int ac, uchar *av[]) /* set up ARGV and ARGC */
 89 arginit(int ac, uchar *av[])
140 {
141         Cell *cp;
142         int i;
143         uchar temp[50];
 93         uchar temp[11];

 95         /* first make FILENAME first real argument */
 96         for (i = 1; i < ac; i++) {
 97                 if (!isclvar(av[i])) {
 98                         (void) setsval(lookup((uchar *)"FILENAME", symtab),
 99                             av[i]);
100                         break;
101                 }
102         }
145         ARGC = &setsymtab((uchar *)"ARGC", (uchar *)"", (Awkfloat)ac,
146             NUM, symtab)->fval;
147         cp = setsymtab((uchar *)"ARGV", (uchar *)"", 0.0, ARR, symtab);
148         ARGVtab = makesymtab(NSYMTAB);  /* could be (int) ARGC as well */
149         cp->sval = (uchar *)ARGVtab;
107         cp->sval = (uchar *) ARGVtab;
150         for (i = 0; i < ac; i++) {
151                 (void) sprintf((char *)temp, "%d", i);
152                 if (is_number(*av)) {
153                         (void) setsymtab(temp, *av, atof((const char *)*av),
154                             STR|NUM, ARGVtab);
155                 } else {
156                         (void) setsymtab(temp, *av, 0.0, STR, ARGVtab);
157                 }
158                 av++;
```



```
159         }
160 }

162 void
163 envinit(uchar **envp)           /* set up ENVIRON variable */
121 envinit(uchar *envp[])
164 {
165         Cell *cp;
166         uchar *p;

168         cp = setsymtab((uchar *)"ENVIRON", (uchar *)"", 0.0, ARR, symtab);
169         ENVtab = makesymtab(NSYMTAB);
170         cp->sval = (uchar *)ENVtab;
128         cp->sval = (uchar *) ENVtab;
171         for (; *envp; envp++) {
172                 if ((p = (uchar *)strchr((char *)*envp, '=')) == NULL)
173                         continue;
174                 if (p == *envp) /* no left hand side name in env string */
175                         continue;
176 #endif /* ! codereview */
177                 *p++ = 0;       /* split into two strings at = */
178                 if (is_number(p)) {
179                         (void) setsymtab(*envp, p, atof((const char *)p),
180                             STR|NUM, ENVtab);
181                 } else {
182                         (void) setsymtab(*envp, p, 0.0, STR, ENVtab);
183                 }
184                 /* restore in case env is passed down to a shell */
185                 p[-1] = '=';
186         }
187 }

189 Array *
190 makesymtab(int n)       /* make a new symbol table */
132 makesymtab(int n)
191 {
192         Array *ap;
193         Cell **tp;

195         ap = (Array *)malloc(sizeof (Array));
196         tp = (Cell **)calloc(n, sizeof (Cell *));
197         if (ap == NULL || tp == NULL)
198                 FATAL("out of space in makesymtab");
140                 ERROR "out of space in makesymtab" FATAL;
199         ap->nelem = 0;
200         ap->size = n;
201         ap->tab = tp;
202         return (ap);
203 }

205 void
206 freesymtab(Cell *ap)    /* free symbol table */
207 {
208         Cell *cp, *temp;
150         Cell *cp, *next;
209         Array *tp;
210         int i;

212         if (!isarr(ap))
213                 return;
214         /*LINTED align*/
215         tp = (Array *)ap->sval;
216         if (tp == NULL)
217                 return;
218         for (i = 0; i < tp->size; i++) {
219                 for (cp = tp->tab[i]; cp != NULL; cp = temp) {
```

```
161                 for (cp = tp->tab[i]; cp != NULL; cp = next) {
162                         next = cp->cnext;
220                         xfree(cp->nval);
221                         if (freeable(cp))
222                                 xfree(cp->sval);
223                         /* avoids freeing then using */
224                         temp = cp->cnext;
225 #endif /* ! codereview */
226                         free(cp);
227                         tp->nelem--;
228 #endif /* ! codereview */
229                 }
230                 tp->tab[i] = 0;
231 #endif /* ! codereview */
232         }
233         if (tp->nelem != 0)
234                 WARNING("can't happen: inconsistent element count freeing %s",
235                     ap->nval);
236 #endif /* ! codereview */
237         free(tp->tab);
238         free(tp);
239 }

241 void
242 freeelem(Cell *ap, const uchar *s) /* free elem s from ap (i.e., ap["s"] */
166 freeelem(Cell *ap, uchar *s)           /* free elem s from ap (i.e., ap["s"] */
243 {
244         Array *tp;
245         Cell *p, *prev = NULL;
246         int h;

248         /*LINTED align*/
249         tp = (Array *)ap->sval;
250         h = hash(s, tp->size);
251         for (p = tp->tab[h]; p != NULL; prev = p, p = p->cnext)
252                 if (strcmp((char *)s, (char *)p->nval) == 0) {
253                         if (prev == NULL)       /* 1st one */
254                                 tp->tab[h] = p->cnext;
255                         else                    /* middle somewhere */
256                                 prev->cnext = p->cnext;
257                         if (freeable(p))
258                                 xfree(p->sval);
259                         free(p->nval);
260                         free(p);
261                         tp->nelem--;
262                         return;
263                 }
264 }

266 Cell *
267 setsymtab(const uchar *n, const uchar *s, Awkfloat f, unsigned int t,
268     Array *tp)
191 setsymtab(uchar *n, uchar *s, Awkfloat f, unsigned int t, Array *tp)
269 {
270         int h;
271         Cell *p;
193         register int h;
194         register Cell *p;

273         if (n != NULL && (p = lookup(n, tp)) != NULL) {
274                 dprintf(("setsymtab found %p: n=%s s=\"%s\" f=%g t=%o\n",
275                     (void *)p, p->nval, p->sval, p->fval, p->tval));
197                 dprintf(("setsymtab found %p: n=%s", (void *)p, p->nval));
198                 dprintf((" s=\"%s\" f=%g t=%p\n",
199                     p->sval, p->fval, (void *)p->tval));
276                 return (p);
```



```
277         }
278         p = (Cell *)malloc(sizeof (Cell));
279         if (p == NULL)
280                 FATAL("out of space for symbol table at %s", n);
204                 ERROR "symbol table overflow at %s", n FATAL;
281         p->nval = tostring(n);
282         p->sval = s ? tostring(s) : tostring((uchar *)"");
283         p->fval = f;
284         p->tval = t;
285         p->csub = CUNK;
286         p->ctype = OCELL;
209         p->csub = 0;

287         tp->nelem++;
288         if (tp->nelem > FULLTAB * tp->size)
289                 rehash(tp);
290         h = hash(n, tp->size);
291         p->cnext = tp->tab[h];
292         tp->tab[h] = p;
293         dprintf(("setsymtab set %p: n=%s s=\"%s\" f=%g t=%o\n",
294             (void *)p, p->nval, p->sval, p->fval, p->tval));
217         dprintf(("setsymtab set %p: n=%s", (void *)p, p->nval));
218         dprintf((" s=\"%s\" f=%g t=%p\n", p->sval, p->fval, (void *)p->tval));
295         return (p);
296 }

298 int
299 hash(const uchar *s, int n)     /* form hash value for string s */
223 hash(uchar *s, int n)   /* form hash value for string s */
300 {
301         unsigned hashval;
225         register unsigned hashval;

303         for (hashval = 0; *s != '\0'; s++)
304                 hashval = (*s + 31 * hashval);
305         return (hashval % n);
306 }

308 static void
309 rehash(Array *tp)       /* rehash items in small table into big one */
310 {
311         int i, nh, nsz;
312         Cell *cp, *op, **np;

314         nsz = GROWTAB * tp->size;
315         np = (Cell **)calloc(nsz, sizeof (Cell *));
316         if (np == NULL)         /* can't do it, but can keep running. */
317                 return;         /* someone else will run out later. */
240         if (np == NULL)
241                 ERROR "out of space in rehash" FATAL;
318         for (i = 0; i < tp->size; i++) {
319                 for (cp = tp->tab[i]; cp; cp = op) {
320                         op = cp->cnext;
321                         nh = hash(cp->nval, nsz);
322                         cp->cnext = np[nh];
323                         np[nh] = cp;
324                 }
325         }
326         free(tp->tab);
327         tp->tab = np;
328         tp->size = nsz;
329 }

331 Cell *
332 lookup(const uchar *s, Array *tp)       /* look for s in tp */
256 lookup(uchar *s, Array *tp)     /* look for s in tp */
```

```
{
        Cell *p;
        register Cell *p;
        int h;

        h = hash(s, tp->size);
        for (p = tp->tab[h]; p != NULL; p = p->cnext) {
                if (strcmp((char *)s, (char *)p->nval) == 0)
                        return (p);     /* found it */
        }
        return (NULL);                  /* not found */
}

Awkfloat
setfval(Cell *vp, Awkfloat f)   /* set float val of a Cell */
setfval(Cell *vp, Awkfloat f)
{
        int fldno;
        int     i;

        if ((vp->tval & (NUM | STR)) == 0)
                funnyvar(vp, "assign to");
        if (isfld(vp)) {
        if (vp->tval & FLD) {
                donerec = 0;    /* mark $0 invalid */
                fldno = atoi((char *)vp->nval);
                if (fldno > *NF)
                        newfld(fldno);
                dprintf(("setting field %d to %g\n", fldno, f));
        } else if (isrec(vp)) {
                i = fldidx(vp);
                if (i > *NF)
                        newfld(i);
                dprintf(("setting field %d to %g\n", i, f));
        } else if (vp->tval & REC) {
                donefld = 0;    /* mark $1... invalid */
                donerec = 1;
        }
        if (freeable(vp))
                xfree(vp->sval); /* free any previous string */
#endif /* ! codereview */
        vp->tval &= ~STR;       /* mark string invalid */
        vp->tval |= NUM;        /* mark number ok */
        if (f == -0)  /* who would have thought this possible? */
                f = 0;
        dprintf(("setfval %p: %s = %g, t=%o\n", (void *)vp,
            vp->nval, f, vp->tval));
        dprintf(("setfval %p: %s = %g, t=%p\n", (void *)vp,
            vp->nval ? vp->nval : (unsigned char *)"NULL",
            f, (void *)vp->tval));
        return (vp->fval = f);
}

void
funnyvar(Cell *vp, char *rw)
{
        if (isarr(vp))
                FATAL("can't %s %s; it's an array name.", rw, vp->nval);
        if (vp->tval & ARR)
                ERROR "can't %s %s; it's an array name.", rw, vp->nval FATAL;
        if (vp->tval & FCN)
                FATAL("can't %s %s; it's a function.", rw, vp->nval);
        WARNING("funny variable %p: n=%s s=\"%s\" f=%g t=%o",
            vp, vp->nval, vp->sval, vp->fval, vp->tval);
                ERROR "can't %s %s; it's a function.", rw, vp->nval FATAL;
        ERROR "funny variable %o: n=%s s=\"%s\" f=%g t=%o",
```



```
            vp, vp->nval, vp->sval, vp->fval, vp->tval CONT;
}

uchar *
setsval(Cell *vp, const uchar *s)       /* set string val of a Cell */
setsval(Cell *vp, uchar *s)
{
        uchar *t;
        int fldno;
        int     i;

        dprintf(("starting setsval %p: %s = \"%s\", t=%o, r,f=%d,%d\n",
            (void*)vp, vp->nval, s, vp->tval, donerec, donefld));
#endif /* ! codereview */
        if ((vp->tval & (NUM | STR)) == 0)
                funnyvar(vp, "assign to");
        if (isfld(vp)) {
        if (vp->tval & FLD) {
                donerec = 0;    /* mark $0 invalid */
                fldno = atoi((const char *)vp->nval);
                if (fldno > *NF)
                        newfld(fldno);
                dprintf(("setting field %d to %s (%p)\n", fldno, s, (void *)s));
        } else if (isrec(vp)) {
                i = fldidx(vp);
                if (i > *NF)
                        newfld(i);
                dprintf(("setting field %d to %s\n", i, s));
        } else if (vp->tval & REC) {
                donefld = 0;    /* mark $1... invalid */
                donerec = 1;
        }
        t = tostring(s);        /* in case it's self-assign */
        if (freeable(vp))
                xfree(vp->sval);
#endif /* ! codereview */
        vp->tval &= ~NUM;
        vp->tval |= STR;
        if (freeable(vp))
                xfree(vp->sval);
        vp->tval &= ~DONTFREE;
        dprintf(("setsval %p: %s = \"%s (%p) \", t=%o r,f=%d,%d\n",
            (void *)vp, vp->nval, t, (void *)t, vp->tval, donerec, donefld));
        return (vp->sval = t);
        dprintf(("setsval %p: %s = \"%s\", t=%p\n",
            (void *)vp,
            vp->nval ? (char *)vp->nval : "",
            s,
            (void *)(vp->tval ? (char *)vp->tval : "")));
        return (vp->sval = tostring(s));
}

Awkfloat
getfval(Cell *vp)       /* get float val of a Cell */
r_getfval(Cell *vp)
{
        if ((vp->tval & (NUM | STR)) == 0)
                funnyvar(vp, "read value of");
        if (isfld(vp) && donefld == 0)
        if ((vp->tval & FLD) && donefld == 0)
                fldbld();
        else if (isrec(vp) && donerec == 0)
        else if ((vp->tval & REC) && donerec == 0)
                recbld();
        if (!isnum(vp)) {       /* not a number */
                vp->fval = atof((const char *)vp->sval);        /* best guess */
```

```
429                 if (is_number(vp->sval) && !(vp->tval & CON))
340                 if (is_number(vp->sval) && !(vp->tval&CON))
430                         vp->tval |= NUM;        /* make NUM only sparingly */
431         }
432         dprintf(("getfval %p: %s = %g, t=%o\n",
433             (void *)vp, vp->nval, vp->fval, vp->tval));
343         dprintf(("getfval %p: %s = %g, t=%p\n",
344             (void *)vp, vp->nval, vp->fval, (void *)vp->tval));
434         return (vp->fval);
435 }

437 static uchar *
438 get_str_val(Cell *vp, uchar **fmt)      /* get string val of a Cell */
348 uchar *
349 r_getsval(Cell *vp)
439 {
440         uchar s[100];   /* BUG: unchecked */
441         double dtemp;
351         uchar s[256];

443         if ((vp->tval & (NUM | STR)) == 0)
444                 funnyvar(vp, "read value of");
445         if (isfld(vp) && donefld == 0)
355         if ((vp->tval & FLD) && donefld == 0)
446                 fldbld();
447         else if (isrec(vp) && donerec == 0)
357         else if ((vp->tval & REC) && donerec == 0)
448                 recbld();
449         if (isstr(vp) == 0) {
450                 if (freeable(vp))
359         if ((vp->tval & STR) == 0) {
360                 if (!(vp->tval&DONTFREE))
451                         xfree(vp->sval);
452                 /* it's integral */
453                 if (modf((long long)vp->fval, &dtemp) == 0) {
362                 if ((long long)vp->fval == vp->fval) {
454                         (void) snprintf((char *)s, sizeof (s),
455                             "%.30g", vp->fval);
364                             "%.20g", vp->fval);
456                 } else {
457                         /*LINTED*/
458                         (void) snprintf((char *)s, sizeof (s),
459                             (char *)*fmt, vp->fval);
368                             (char *)*OFMT, vp->fval);
460                 }
461                 vp->sval = tostring(s);
462                 vp->tval &= ~DONTFREE;
463                 vp->tval |= STR;
464         }
465         dprintf(("getsval %p: %s = \"%s (%p)\", t=%o\n",
466             (void *)vp, vp->nval, vp->sval, (void *)vp->sval, vp->tval));
374         dprintf(("getsval %p: %s = \"%s\", t=%p\n",
375             (void *)vp,
376             vp->nval ? (char *)vp->nval : "",
377             vp->sval ? (char *)vp->sval : "",
378             (void *)vp->tval));
467         return (vp->sval);
468 }

470 uchar *
471 getsval(Cell *vp)       /* get string val of a Cell */
472 {
473         return (get_str_val(vp, CONVFMT));
474 }

476 uchar *
```



```
477 getpssval(Cell *vp)     /* get string val of a Cell for print */
478 {
479         return (get_str_val(vp, OFMT));
480 }

482 uchar *
483 tostring(const uchar *s)        /* make a copy of string s */
383 tostring(uchar *s)
484 {
485         uchar *p;
385         register uchar *p;

487         p = (uchar *)malloc(strlen((char *)s)+1);
488         if (p == NULL)
489                 FATAL("out of space in tostring on %s", s);
389                 ERROR "out of space in tostring on %s", s FATAL;
490         (void) strcpy((char *)p, (char *)s);
491         return (p);
492 }

494 uchar *
495 qstring(const uchar *is, int delim)     /* collect string up to delim */
395 qstring(uchar *s, int delim)    /* collect string up to delim */
496 {
497         const uchar *os = is;
397         uchar *cbuf, *ret;
498         int c, n;
499         uchar *s = (uchar *)is;
500         uchar *buf, *bp;
399         size_t  cbufsz, cnt;

401         init_buf(&cbuf, &cbufsz, LINE_INCR);

502         if ((buf = (uchar *)malloc(strlen((char *)is)+3)) == NULL)
503                 FATAL("out of space in qstring(%s)", s);
504         for (bp = buf; (c = *s) != delim; s++) {
403         for (cnt = 0; (c = *s) != delim; s++) {
505                 if (c == '\n') {
506                         SYNTAX("newline in string %.20s...", os);
405                         ERROR "newline in string %.10s...", cbuf SYNTAX;
507                 } else if (c != '\\') {
508                         *bp++ = c;
407                         expand_buf(&cbuf, &cbufsz, cnt);
408                         cbuf[cnt++] = c;
509                 } else {        /* \something */
510                         c = *++s;
511                         if (c == 0) {   /* \ at end */
512                                 *bp++ = '\\';
513                                 break;  /* for loop */
514                         }
515                         switch (c) {
516                         case '\\':      *bp++ = '\\'; break;
517                         case 'n':       *bp++ = '\n'; break;
518                         case 't':       *bp++ = '\t'; break;
519                         case 'b':       *bp++ = '\b'; break;
520                         case 'f':       *bp++ = '\f'; break;
521                         case 'r':       *bp++ = '\r'; break;
410                         expand_buf(&cbuf, &cbufsz, cnt);
411                         switch (c = *++s) {
412                         case '\\':      cbuf[cnt++] = '\\'; break;
413                         case 'n':       cbuf[cnt++] = '\n'; break;
414                         case 't':       cbuf[cnt++] = '\t'; break;
415                         case 'b':       cbuf[cnt++] = '\b'; break;
416                         case 'f':       cbuf[cnt++] = '\f'; break;
417                         case 'r':       cbuf[cnt++] = '\r'; break;
522                         default:
```

```
523                                 if (!isdigit(c)) {
524                                         *bp++ = c;
420                                         cbuf[cnt++] = c;
525                                         break;
526                                 }
527                                 n = c - '0';
528                                 if (isdigit(s[1])) {
529                                         n = 8 * n + *++s - '0';
530                                         if (isdigit(s[1]))
531                                                 n = 8 * n + *++s - '0';
532                                 }
533                                 *bp++ = n;
429                                 cbuf[cnt++] = n;
534                                 break;
535                         }
536                 }
537         }
538         *bp++ = 0;
539         return (buf);
434         cbuf[cnt] = '\0';
435         ret = tostring(cbuf);
436         free(cbuf);
437         return (ret);
540 }
______unchanged_portion_omitted_
```