メールもビトマのソース（LGPL）

（C）宇山　靖政

2012.04.26

メールもビトマのソースファイルのうちで、LGPL 規定に従うものを公開します。

内容の変更はしていませんので、本来は公開の必要はありません。公知の技術とするために使用する全てのソースファイルを公開します。規定は LGPL であり、GPL ではありませんのでご注意下さい。

ファイルは２つです。_regex.h と regex.cpp です。

最初は、_regex.h です。

_regex.h

```
//////////////////////////////////////////////////////
/* Definitions for data structures and routines for the regular
   expression library, version 0.12.
   Copyright (C) 1985,89,90,91,92,93,95,96,97 Free Software Foundation, Inc.

   This file is part of the GNU C Library.  Its master source is NOT part of
   the C library, however.  The master source lives in /gd/gnu/lib.

   The GNU C Library is free software; you can redistribute it and/or
   modify it under the terms of the GNU Library General Public License as
   published by the Free Software Foundation; either version 2 of the
   License, or (at your option) any later version.

   The GNU C Library is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
   Library General Public License for more details.

   You should have received a copy of the GNU Library General Public
   License along with the GNU C Library; see the file COPYING.LIB.  If not,
   write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330,
   Boston, MA 02111-1307, USA.  */

#ifndef __REGEXP_LIBRARY_H__
#define __REGEXP_LIBRARY_H__

/* Allow the use in C++ code.  */
#ifdef __cplusplus
extern "C" {
#endif

/* POSIX says that <sys/types.h> must be included (by the caller) before
   <regex.h>.  */

#if !defined (_POSIX_C_SOURCE) && !defined (_POSIX_SOURCE) && defined (VMS)
```

```
/* VMS doesn't have `size_t' in <sys/types.h>, even though POSIX says it
   should be there.  */
#include <stddef.h>
#endif

/* The following two types have to be signed and unsigned integer type
   wide enough to hold a value of a pointer.  For most ANSI compilers
   ptrdiff_t and size_t should be likely OK.  Still size of these two
   types is 2 for Microsoft C.  Ugh... */
typedef long int s_reg_t;
typedef unsigned long int active_reg_t;

/* The following bits are used to determine the regexp syntax we
   recognize.  The set/not-set meanings are chosen so that Emacs syntax
   remains the value 0.  The bits are given in alphabetical order, and
   the definitions shifted by one from the previous bit; thus, when we
   add or remove a bit, only one other definition need change.  */
typedef unsigned long int reg_syntax_t;

/* If this bit is not set, then ¥ inside a bracket expression is literal.
   If set, then such a ¥ quotes the following character.  */
#define RE_BACKSLASH_ESCAPE_IN_LISTS ((unsigned long int) 1)

/* If this bit is not set, then + and ? are operators, and ¥+ and ¥? are
     literals.
   If set, then ¥+ and ¥? are operators and + and ? are literals.  */
#define RE_BK_PLUS_QM (RE_BACKSLASH_ESCAPE_IN_LISTS << 1)

/* If this bit is set, then character classes are supported.  They are:
     [:alpha:], [:upper:], [:lower:],  [:digit:], [:alnum:], [:xdigit:],
     [:space:], [:print:], [:punct:], [:graph:], and [:cntrl:].
   If not set, then character classes are not supported.  */
#define RE_CHAR_CLASSES (RE_BK_PLUS_QM << 1)

/* If this bit is set, then ^ and $ are always anchors (outside bracket
     expressions, of course).
   If this bit is not set, then it depends:
        ^  is an anchor if it is at the beginning of a regular
           expression or after an open-group or an alternation operator;
        $  is an anchor if it is at the end of a regular expression, or
           before a close-group or an alternation operator.

   This bit could be (re)combined with RE_CONTEXT_INDEP_OPS, because
   POSIX draft 11.2 says that * etc. in leading positions is undefined.
   We already implemented a previous draft which made those constructs
   invalid, though, so we haven't changed the code back.  */
#define RE_CONTEXT_INDEP_ANCHORS (RE_CHAR_CLASSES << 1)

/* If this bit is set, then special characters are always special
     regardless of where they are in the pattern.
   If this bit is not set, then special characters are special only in
     some contexts; otherwise they are ordinary.  Specifically,
```

```
     * + ? and intervals are only special when not after the beginning,
     open-group, or alternation operator.  */
#define RE_CONTEXT_INDEP_OPS (RE_CONTEXT_INDEP_ANCHORS << 1)

/* If this bit is set, then *, +, ?, and { cannot be first in an re or
     immediately after an alternation or begin-group operator.  */
#define RE_CONTEXT_INVALID_OPS (RE_CONTEXT_INDEP_OPS << 1)

/* If this bit is set, then . matches newline.
   If not set, then it doesn't.  */
#define RE_DOT_NEWLINE (RE_CONTEXT_INVALID_OPS << 1)

/* If this bit is set, then . doesn't match NUL.
   If not set, then it does.  */
#define RE_DOT_NOT_NULL (RE_DOT_NEWLINE << 1)

/* If this bit is set, nonmatching lists [^...] do not match newline.
   If not set, they do.  */
#define RE_HAT_LISTS_NOT_NEWLINE (RE_DOT_NOT_NULL << 1)

/* If this bit is set, either ¥{...¥} or {...} defines an
     interval, depending on RE_NO_BK_BRACES.
   If not set, ¥{, ¥}, {, and } are literals.  */
#define RE_INTERVALS (RE_HAT_LISTS_NOT_NEWLINE << 1)

/* If this bit is set, +, ? and | aren't recognized as operators.
   If not set, they are.  */
#define RE_LIMITED_OPS (RE_INTERVALS << 1)

/* If this bit is set, newline is an alternation operator.
   If not set, newline is literal.  */
#define RE_NEWLINE_ALT (RE_LIMITED_OPS << 1)

/* If this bit is set, then `{...}' defines an interval, and ¥{ and ¥}
     are literals.
  If not set, then `¥{...¥}' defines an interval.  */
#define RE_NO_BK_BRACES (RE_NEWLINE_ALT << 1)

/* If this bit is set, (...) defines a group, and ¥( and ¥) are literals.
   If not set, ¥(...¥) defines a group, and ( and ) are literals.  */
#define RE_NO_BK_PARENS (RE_NO_BK_BRACES << 1)

/* If this bit is set, then ¥<digit> matches <digit>.
   If not set, then ¥<digit> is a back-reference.  */
#define RE_NO_BK_REFS (RE_NO_BK_PARENS << 1)

/* If this bit is set, then | is an alternation operator, and ¥| is literal.
   If not set, then ¥| is an alternation operator, and | is literal.  */
#define RE_NO_BK_VBAR (RE_NO_BK_REFS << 1)

/* If this bit is set, then an ending range point collating higher
     than the starting range point, as in [z-a], is invalid.
```

```
   If not set, then when ending range point collates higher than the
     starting range point, the range is ignored.  */
#define RE_NO_EMPTY_RANGES (RE_NO_BK_VBAR << 1)

/* If this bit is set, then an unmatched ) is ordinary.
   If not set, then an unmatched ) is invalid.  */
#define RE_UNMATCHED_RIGHT_PAREN_ORD (RE_NO_EMPTY_RANGES << 1)

/* If this bit is set, succeed as soon as we match the whole pattern,
   without further backtracking.  */
#define RE_NO_POSIX_BACKTRACKING (RE_UNMATCHED_RIGHT_PAREN_ORD << 1)

/* If this bit is set, do not process the GNU regex operators.
   If not set, then the GNU regex operators are recognized. */
#define RE_NO_GNU_OPS (RE_NO_POSIX_BACKTRACKING << 1)

/* If this bit is set, turn on internal regex debugging.
   If not set, and debugging was on, turn it off.
   This only works if regex.c is compiled -DDEBUG.
   We define this bit always, so that all that's needed to turn on
   debugging is to recompile regex.c; the calling code can always have
   this bit set, and it won't affect anything in the normal case. */
#define RE_DEBUG (RE_NO_GNU_OPS << 1)

/* This global variable defines the particular regexp syntax to use (for
   some interfaces).  When a regexp is compiled, the syntax used is
   stored in the pattern buffer, so changing this does not affect
   already-compiled regexps.  */
extern reg_syntax_t re_syntax_options;
@par /* Define combinations of the above bits for the standard possibilities.
   (The [[[ comments delimit what gets put into the Texinfo file, so
   don't delete them!)  */
/* [[[begin syntaxes]]] */
#define RE_SYNTAX_EMACS 0

#define RE_SYNTAX_AWK                                                   ¥
  (RE_BACKSLASH_ESCAPE_IN_LISTS   | RE_DOT_NOT_NULL                     ¥
   | RE_NO_BK_PARENS              | RE_NO_BK_REFS                  ¥
   | RE_NO_BK_VBAR                | RE_NO_EMPTY_RANGES                 ¥
   | RE_DOT_NEWLINE                 | RE_CONTEXT_INDEP_ANCHORS         ¥
   | RE_UNMATCHED_RIGHT_PAREN_ORD | RE_NO_GNU_OPS)

#define RE_SYNTAX_GNU_AWK                                               ¥
  ((RE_SYNTAX_POSIX_EXTENDED | RE_BACKSLASH_ESCAPE_IN_LISTS | RE_DEBUG)     ¥
   & ~(RE_DOT_NOT_NULL | RE_INTERVALS | RE_CONTEXT_INDEP_OPS))

#define RE_SYNTAX_POSIX_AWK                                             ¥
  (RE_SYNTAX_POSIX_EXTENDED | RE_BACKSLASH_ESCAPE_IN_LISTS         ¥
   | RE_INTERVALS            | RE_NO_GNU_OPS)

#define RE_SYNTAX_GREP                                                  ¥
  (RE_BK_PLUS_QM              | RE_CHAR_CLASSES                        ¥
```

```
   | RE_HAT_LISTS_NOT_NEWLINE | RE_INTERVALS                                ¥
   | RE_NEWLINE_ALT)

#define RE_SYNTAX_EGREP                                                    ¥
  (RE_CHAR_CLASSES        | RE_CONTEXT_INDEP_ANCHORS                       ¥
   | RE_CONTEXT_INDEP_OPS | RE_HAT_LISTS_NOT_NEWLINE                       ¥
   | RE_NEWLINE_ALT       | RE_NO_BK_PARENS                               ¥
   | RE_NO_BK_VBAR)

#define RE_SYNTAX_POSIX_EGREP                                              ¥
  (RE_SYNTAX_EGREP | RE_INTERVALS | RE_NO_BK_BRACES)

/* P1003.2/D11.2, section 4.20.7.1, lines 5078ff.  */
#define RE_SYNTAX_ED RE_SYNTAX_POSIX_BASIC

#define RE_SYNTAX_SED RE_SYNTAX_POSIX_BASIC

/* Syntax bits common to both basic and extended POSIX regex syntax.  */
#define _RE_SYNTAX_POSIX_COMMON                                            ¥
  (RE_CHAR_CLASSES | RE_DOT_NEWLINE      | RE_DOT_NOT_NULL         ¥
   | RE_INTERVALS  | RE_NO_EMPTY_RANGES)

#define RE_SYNTAX_POSIX_BASIC                                              ¥
  (_RE_SYNTAX_POSIX_COMMON | RE_BK_PLUS_QM)

/* Differs from ..._POSIX_BASIC only in that RE_BK_PLUS_QM becomes
   RE_LIMITED_OPS, i.e., ¥? ¥+ ¥| are not recognized.  Actually, this
   isn't minimal, since other operators, such as ¥`, aren't disabled.  */
#define RE_SYNTAX_POSIX_MINIMAL_BASIC                                      ¥
  (_RE_SYNTAX_POSIX_COMMON | RE_LIMITED_OPS)

#define RE_SYNTAX_POSIX_EXTENDED                                           ¥
  (_RE_SYNTAX_POSIX_COMMON | RE_CONTEXT_INDEP_ANCHORS                      ¥
   | RE_CONTEXT_INDEP_OPS  | RE_NO_BK_BRACES                              ¥
   | RE_NO_BK_PARENS       | RE_NO_BK_VBAR                                ¥
   | RE_UNMATCHED_RIGHT_PAREN_ORD)

/* Differs from ..._POSIX_EXTENDED in that RE_CONTEXT_INVALID_OPS
   replaces RE_CONTEXT_INDEP_OPS and RE_NO_BK_REFS is added.  */
#define RE_SYNTAX_POSIX_MINIMAL_EXTENDED                                   ¥
  (_RE_SYNTAX_POSIX_COMMON  | RE_CONTEXT_INDEP_ANCHORS                     ¥
   | RE_CONTEXT_INVALID_OPS | RE_NO_BK_BRACES                              ¥
   | RE_NO_BK_PARENS        | RE_NO_BK_REFS                                ¥
   | RE_NO_BK_VBAR           | RE_UNMATCHED_RIGHT_PAREN_ORD)
/* [[[end syntaxes]]] */
¥par /* Maximum number of duplicates an interval can allow.  Some systems
   (erroneously) define this in other header files, but we want our
   value, so remove any previous define.  */
#ifdef RE_DUP_MAX
#undef RE_DUP_MAX
#endif
/* If sizeof(int) == 2, then ((1 << 15) - 1) overflows.  */
```

```
#define RE_DUP_MAX (0x7fff)


/* POSIX `cflags' bits (i.e., information for `regcomp').  */

/* If this bit is set, then use extended regular expression syntax.
   If not set, then use basic regular expression syntax.  */
#define REG_EXTENDED 1

/* If this bit is set, then ignore case when matching.
   If not set, then case is significant.  */
#define REG_ICASE (REG_EXTENDED << 1)

/* If this bit is set, then anchors do not match at newline
     characters in the string.
   If not set, then anchors do match at newlines.  */
#define REG_NEWLINE (REG_ICASE << 1)

/* If this bit is set, then report only success or fail in regexec.
   If not set, then returns differ between not matching and errors.  */
#define REG_NOSUB (REG_NEWLINE << 1)


/* POSIX `eflags' bits (i.e., information for regexec).  */

/* If this bit is set, then the beginning-of-line operator doesn't match
     the beginning of the string (presumably because it's not the
     beginning of a line).
   If not set, then the beginning-of-line operator does match the
     beginning of the string.  */
#define REG_NOTBOL 1

/* Like REG_NOTBOL, except for the end-of-line.  */
#define REG_NOTEOL (1 << 1)


/* If any error codes are removed, changed, or added, update the
   `re_error_msg' table in regex.c.  */
typedef enum
{
  REG_NOERROR = 0,        /* Success.  */
  REG_NOMATCH,            /* Didn't find a match (for regexec).  */

  /* POSIX regcomp return error codes.  (In the order listed in the
     standard.)  */
  REG_BADPAT,             /* Invalid pattern.  */
  REG_ECOLLATE,           /* Not implemented.  */
  REG_ECTYPE,             /* Invalid character class name.  */
  REG_EESCAPE,            /* Trailing backslash.  */
  REG_ESUBREG,            /* Invalid back reference.  */
  REG_EBRACK,             /* Unmatched left bracket.  */
  REG_EPAREN,             /* Parenthesis imbalance.  */
```

```
  REG_EBRACE,             /* Unmatched ¥{.  */
  REG_BADBR,              /* Invalid contents of ¥{¥}.  */
  REG_ERANGE,             /* Invalid range end.  */
  REG_ESPACE,             /* Ran out of memory.  */
  REG_BADRPT,             /* No preceding re for repetition op.  */

  /* Error codes we've added.  */
  REG_EEND,               /* Premature end.  */
  REG_ESIZE,              /* Compiled pattern bigger than 2^16 bytes.  */
  REG_ERPAREN             /* Unmatched ) or ¥); not returned from regcomp.  */
} reg_errcode_t;
¥par /* This data structure represents a compiled pattern.  Before calling
   the pattern compiler, the fields `buffer', `allocated', `fastmap',
   `translate', and `no_sub' can be set.  After the pattern has been
   compiled, the `re_nsub' field is available.  All other fields are
   private to the regex routines.  */

#ifndef RE_TRANSLATE_TYPE
#define RE_TRANSLATE_TYPE char *
#endif

struct re_pattern_buffer
{
/* [[[begin pattern_buffer]]] */
        /* Space that holds the compiled pattern.  It is declared as
           `unsigned char *' because its elements are
           sometimes used as array indexes.  */
  unsigned char *buffer;

        /* Number of bytes to which `buffer' points.  */
  unsigned long int allocated;

        /* Number of bytes actually used in `buffer'.  */
  unsigned long int used;

       /* Syntax setting with which the pattern was compiled.  */
  reg_syntax_t syntax;

       /* Pointer to a fastmap, if any, otherwise zero.  re_search uses
          the fastmap, if there is one, to skip over impossible
          starting points for matches.  */
  char *fastmap;

       /* Either a translate table to apply to all characters before
          comparing them, or zero for no translation.  The translation
          is applied to a pattern when it is compiled and to a string
          when it is matched.  */
  RE_TRANSLATE_TYPE translate;

       /* Number of subexpressions found by the compiler.  */
  size_t re_nsub;
```

```
        /* Zero if this pattern cannot match the empty string, one else.
           Well, in truth it's used only in `re_search_2', to see
           whether or not we should use the fastmap, so we don't set
           this absolutely perfectly; see `re_compile_fastmap' (the
           `duplicate' case).  */
  unsigned can_be_null : 1;

        /* If REGS_UNALLOCATED, allocate space in the `regs' structure
             for `max (RE_NREGS, re_nsub + 1)' groups.
           If REGS_REALLOCATE, reallocate space if necessary.
           If REGS_FIXED, use what's there.  */
#define REGS_UNALLOCATED 0
#define REGS_REALLOCATE 1
#define REGS_FIXED 2
  unsigned regs_allocated : 2;

        /* Set to zero when `regex_compile' compiles a pattern; set to one
           by `re_compile_fastmap' if it updates the fastmap.  */
  unsigned fastmap_accurate : 1;

        /* If set, `re_match_2' does not return information about
           subexpressions.  */
  unsigned no_sub : 1;

        /* If set, a beginning-of-line anchor doesn't match at the
           beginning of the string.  */
  unsigned not_bol : 1;

        /* Similarly for an end-of-line anchor.  */
  unsigned not_eol : 1;

        /* If true, an anchor at a newline matches.  */
  unsigned newline_anchor : 1;

/* [[[end pattern_buffer]]] */
};

typedef struct re_pattern_buffer regex_t;
typedef int regoff_t; /* Type for byte offsets within the string.  POSIX mandates this.  */


/* This is the structure we store register match data in.  See
   regex.texinfo for a full description of what registers match.  */
struct re_registers
{
  unsigned num_regs;
  regoff_t *start;
  regoff_t *end;
};
```

```
/* If `regs_allocated' is REGS_UNALLOCATED in the pattern buffer,
   `re_match_2' returns information about at least this many registers
   the first time a `regs' structure is passed.  */
#ifndef RE_NREGS
#define RE_NREGS 30
#endif


/* POSIX specification for registers.  Aside from the different names than
   `re_registers', POSIX uses an array of structures, instead of a
   structure of arrays.  */
typedef struct
{
  regoff_t rm_so;  /* Byte offset from string's start to substring's start.  */
  regoff_t rm_eo;  /* Byte offset from string's start to substring's end.  */
} regmatch_t;

/* Declarations for routines.  */

/* To avoid duplicating every routine declaration -- once with a
   prototype (if we are ANSI), and once without (if we aren't) -- we
   use the following macro to declare argument types.  This
   unfortunately clutters up the declarations a bit, but I think it's
   worth it.  */

#ifdef __STDC__

#define _RE_ARGS(args) args

#else /* not __STDC__ */

#define _RE_ARGS(args) ()

#endif /* not __STDC__ */

/* Sets the current default syntax to SYNTAX, and return the old syntax.
   You can also simply assign to the `re_syntax_options' variable.  */
extern reg_syntax_t re_set_syntax _RE_ARGS ((reg_syntax_t syntax));

/* Compile the regular expression PATTERN, with length LENGTH
   and syntax given by the global `re_syntax_options', into the buffer
   BUFFER.  Return NULL if successful, and an error string if not.  */
extern const char *re_compile_pattern
  (const char *pattern, size_t length,
              struct re_pattern_buffer *buffer);


/* Compile a fastmap for the compiled pattern in BUFFER; used to
   accelerate searches.  Return 0 if successful and -2 if was an
   internal error.  */
extern int re_compile_fastmap _RE_ARGS ((struct re_pattern_buffer *buffer));
```

```
/* Search in the string STRING (with length LENGTH) for the pattern
   compiled into BUFFER.  Start searching at position START, for RANGE
   characters.  Return the starting position of the match, -1 for no
   match, or -2 for an internal error.  Also return register
   information in REGS (if REGS and BUFFER->no_sub are nonzero).  */
extern int re_search
  _RE_ARGS ((struct re_pattern_buffer *buffer, const char *string,
             int length, int start, int range, struct re_registers *regs));


/* Like `re_search', but search in the concatenation of STRING1 and
   STRING2.  Also, stop searching at index START + STOP.  */
extern int re_search_2
  (struct re_pattern_buffer *buffer, const char *string1,
             int length1, const char *string2, int length2,
             int start, int range, struct re_registers *regs, int stop);


/* Like `re_search', but return how many characters in STRING the regexp
   in BUFFER matched, starting at position START.  */
extern int re_match
  (struct re_pattern_buffer *buffer, const char *string,
             int length, int start, struct re_registers *regs);


/* Relates to `re_match' as `re_search_2' relates to `re_search'.  */
extern int re_match_2
  _RE_ARGS ((struct re_pattern_buffer *buffer, const char *string1,
             int length1, const char *string2, int length2,
             int start, struct re_registers *regs, int stop));


/* Set REGS to hold NUM_REGS registers, storing them in STARTS and
   ENDS.  Subsequent matches using BUFFER and REGS will use this memory
   for recording register information.  STARTS and ENDS must be
   allocated with malloc, and must each be at least `NUM_REGS * sizeof
   (regoff_t)' bytes long.

   If NUM_REGS == 0, then subsequent matches should allocate their own
   register data.

   Unless this function is called, the first search or match using
   PATTERN_BUFFER will allocate its own register data, without
   freeing the old data.  */
extern void re_set_registers
  _RE_ARGS ((struct re_pattern_buffer *buffer, struct re_registers *regs,
             unsigned num_regs, regoff_t *starts, regoff_t *ends));

#ifdef _REGEX_RE_COMP
#ifndef _CRAY
/* 4.2 bsd compatibility.  */
extern char *re_comp _RE_ARGS ((const char *));
```

```
extern int re_exec _RE_ARGS ((const char *));
#endif
#endif

/* POSIX compatibility.  */
extern int regcomp (regex_t *preg, const char *pattern, int cflags);
extern int regexec
  (const regex_t *preg, const char *string, size_t nmatch,
            regmatch_t pmatch[], int eflags);
extern size_t regerror
  (int errcode, const regex_t *preg, char *errbuf,
            size_t errbuf_size);
/* below is changed by hiranaka */
//extern void regfree _RE_ARGS ((regex_t *preg));
extern void regfree (regex_t *preg);


#ifdef __cplusplus
}
#endif  /* C++ */

#endif /* not __REGEXP_LIBRARY_H__ */
ﾀpar /*
Local variables:
make-backup-files: t
version-control: t
trim-versions-without-asking: nil
End:
*/
```

次は、ファイルは、regex.cpp です。

Regex.cpp

```
//////////////////////////////////////////////////////////////////////
/* Extended regular expression matching and search library,
 * version 0.12.
 * (Implements POSIX draft P1003.2/D11.2, except for some of the
 * internationalization features.)
 *
 * Copyright (C) 1993, 1994, 1995, 1996, 1997 Free Software Foundation, Inc.
 *
```

```
 * This file is part of the GNU C Library.    Its master source is NOT part of
 * the C library, however.    The master source lives in /gd/gnu/lib.
 *
 * The GNU C Library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Library General Public License as
 * published by the Free Software Foundation; either version 2 of the
 * License, or (at your option) any later version.
 *
 * The GNU C Library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.    See the GNU
 * Library General Public License for more details.
 *
 * You should have received a copy of the GNU Library General Public
 * License along with the GNU C Library; see the file COPYING.LIB.    If not,
 * write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330,
 * Boston, MA 02111-1307, USA.
 */

/*
 * Modifications:
 *
 * Use _regex.h instead of regex.h.    tlr, 1999-01-06
 * Make REGEX_MALLOC depend on HAVE_ALLOCA &c.
 *                                              tlr, 1999-02-14
 * Don't switch on regex debugging when debugging mutt.
 *                                              tlr, 1999-02-25
 */

/* The following doesn't mix too well with autoconfiguring
 * the use of alloca.    So let's disable it for AIX.
 */

#include "stdafx.h"
```

```
#if 0

/* AIX requires this to be the first thing in the file. */
# if defined (_AIX) && !defined (REGEX_MALLOC)
#    pragma alloca
# endif

#endif

#undef   _GNU_SOURCE
#define _GNU_SOURCE

#ifdef HAVE_CONFIG_H
# include <config.h>
#endif

#undef DEBUG

#if (defined(HAVE_ALLOCA_H) && !defined(_AIX))
# include <alloca.h>
#endif

#if (!defined(HAVE_ALLOCA) || defined(_AIX))
# define REGEX_MALLOC
#endif

#if defined(STDC_HEADERS) && !defined(emacs)
#include <stddef.h>
#else
/* We need this for `regex.h', and perhaps for the Emacs include files.    */
#include <sys/types.h>
#endif
```

```
/* For platform which support the ISO C amendement 1 functionality we
   support user defined character classes.  */
#if defined _LIBC || (defined HAVE_WCTYPE_H && defined HAVE_WCHAR_H)
# include <wctype.h>
# include <wchar.h>
#endif

/* This is for other GNU distributions with internationalized messages.  */
#if HAVE_LIBINTL_H || defined (_LIBC)
# include <libintl.h>
#else
# define gettext(msgid) (msgid)
#endif

#ifndef gettext_noop
/* This define is so xgettext can find the internationalizable
   strings.  */
#define gettext_noop(String) String
#endif

/* The `emacs' switch turns on certain matching commands
   that make sense only in Emacs. */
#ifdef emacs

#include "lisp.h"
#include "buffer.h"
#include "syntax.h"

#else  /* not emacs */

/* If we are not linking with Emacs proper,
   we can't use the relocating allocator
   even if config.h says that we can.  */
#undef REL_ALLOC
```

```
#if defined (STDC_HEADERS) || defined (_LIBC)
#include <stdlib.h>
#else
char *malloc ();
char *realloc ();
#endif

/* When used in Emacs's lib-src, we need to get bzero and bcopy somehow.
   If nothing else has been done, use the method below.  */
#ifdef INHIBIT_STRING_HEADER
#if !(defined (HAVE_BZERO) && defined (HAVE_BCOPY))
#if !defined (bzero) && !defined (bcopy)
#undef INHIBIT_STRING_HEADER
#endif
#endif
#endif

/* This is the normal way of making sure we have a bcopy and a bzero.
   This is used in most programs--a few other programs avoid this
   by defining INHIBIT_STRING_HEADER.  */
#ifndef INHIBIT_STRING_HEADER
//#if defined (HAVE_STRING_H) || defined (STDC_HEADERS) || defined (_LIBC)
#if 1
#include <string.h>
#ifndef bcmp
#define bcmp(s1, s2, n)     memcmp ((s1), (s2), (n))
#endif
#ifndef bcopy
#define bcopy(s, d, n)      memcpy ((d), (s), (n))
#endif
#ifndef bzero
#define bzero(s, n)         memset ((s), 0, (n))
#endif
```

```
#else
#include <strings.h>
#endif
#endif

/* Define the syntax stuff for ¥<, ¥>, etc.  */

/* This must be nonzero for the wordchar and notwordchar pattern
   commands in re_match_2.  */
#ifndef Sword
#define Sword 1
#endif

#ifdef SWITCH_ENUM_BUG
#define SWITCH_ENUM_CAST(x) ((int)(x))
#else
#define SWITCH_ENUM_CAST(x) (x)
#endif

#ifdef SYNTAX_TABLE

extern char *re_syntax_table;

#else /* not SYNTAX_TABLE */

/* How many characters in the character set.  */
#define CHAR_SET_SIZE 256

static char re_syntax_table[CHAR_SET_SIZE];

static void
init_syntax_once ()
{
   register int c;
```

```
    static int done = 0;

    if (done)
      return;

    bzero (re_syntax_table, sizeof re_syntax_table);

    for (c = 'a'; c <= 'z'; c++)
      re_syntax_table[c] = Sword;

    for (c = 'A'; c <= 'Z'; c++)
      re_syntax_table[c] = Sword;

    for (c = '0'; c <= '9'; c++)
      re_syntax_table[c] = Sword;

    re_syntax_table['_'] = Sword;

    done = 1;
}

#endif /* not SYNTAX_TABLE */

#define SYNTAX(c) re_syntax_table[c]

#endif /* not emacs */

/* Get the interface, including the syntax bits.  */

/* Changed to fit into mutt - tlr, 1999-01-06 */

#include "_regex.h"

/* isalpha etc. are used for the character classes.  */
```

```
#include <ctype.h>

/* Jim Meyering writes:

    "... Some ctype macros are valid only for character codes that
    isascii says are ASCII (SGI's IRIX-4.0.5 is one such system --when
    using /bin/cc or gcc but without giving an ansi option).  So, all
    ctype uses should be through macros like ISPRINT...  If
    STDC_HEADERS is defined, then autoconf has verified that the ctype
    macros don't need to be guarded with references to isascii. ...
    Defining isascii to 1 should let any compiler worth its salt
    eliminate the && through constant folding."  */

#if defined (STDC_HEADERS) || (!defined (isascii) && !defined (HAVE_ISASCII))
#define ISASCII(c) 1
#else
#define ISASCII(c) isascii(c)
#endif

#ifdef isblank
#define ISBLANK(c) (ISASCII (c) && isblank (c))
#else
#define ISBLANK(c) ((c) == ' ' || (c) == '¥t')
#endif
#ifdef isgraph
#define ISGRAPH(c) (ISASCII (c) && isgraph (c))
#else
#define ISGRAPH(c) (ISASCII (c) && isprint (c) && !isspace (c))
#endif

#define ISPRINT(c) (ISASCII (c) && isprint (c))
#define ISDIGIT(c) (ISASCII (c) && isdigit (c))
#define ISALNUM(c) (ISASCII (c) && isalnum (c))
#define ISALPHA(c) (ISASCII (c) && isalpha (c))
```

```
#define ISCNTRL(c) (ISASCII (c) && iscntrl (c))
#define ISLOWER(c) (ISASCII (c) && islower (c))
#define ISPUNCT(c) (ISASCII (c) && ispunct (c))
#define ISSPACE(c) (ISASCII (c) && isspace (c))
#define ISUPPER(c) (ISASCII (c) && isupper (c))
#define ISXDIGIT(c) (ISASCII (c) && isxdigit (c))

#ifndef NULL
#define NULL (void *)0
#endif

/* We remove any previous definition of `SIGN_EXTEND_CHAR',
   since ours (we hope) works properly with all combinations of
   machines, compilers, `char' and `unsigned char' argument types.
   (Per Bothner suggested the basic approach.)  */
#undef SIGN_EXTEND_CHAR
#if __STDC__
#define SIGN_EXTEND_CHAR(c) ((signed char) (c))
#else  /* not __STDC__ */
/* As in Harbison and Steele.  */
#define SIGN_EXTEND_CHAR(c) ((((unsigned char) (c)) ^ 128) - 128)
#endif
```

```
/* Should we use malloc or alloca?  If REGEX_MALLOC is not defined, we
   use `alloca' instead of `malloc'.  This is because using malloc in
   re_search* or re_match* could cause memory leaks when C-g is used in
   Emacs; also, malloc is slower and causes storage fragmentation.  On
   the other hand, malloc is more portable, and easier to debug.

   Because we sometimes use alloca, some routines have to be macros,
   not functions -- `alloca'-allocated space disappears at the end of the
   function it is called in.  */

#ifdef REGEX_MALLOC

#define REGEX_ALLOCATE malloc
#define REGEX_REALLOCATE(source, osize, nsize) realloc (source, nsize)
#define REGEX_FREE free

#else /* not REGEX_MALLOC  */

/* Emacs already defines alloca, sometimes.  */
#ifndef alloca

/* Make alloca work the best possible way.  */
#ifdef __GNUC__
#define alloca __builtin_alloca
#else /* not __GNUC__ */
#if HAVE_ALLOCA_H
#include <alloca.h>
#else /* not __GNUC__ or HAVE_ALLOCA_H */
#if 0 /* It is a bad idea to declare alloca.  We always cast the result.  */
#ifndef _AIX /* Already did AIX, up at the top.  */
char *alloca ();
#endif /* not _AIX */
#endif
```

```
#endif /* not HAVE_ALLOCA_H */
#endif /* not __GNUC__ */

#endif /* not alloca */

#define REGEX_ALLOCATE alloca

/* Assumes a `char *destination' variable.  */
#define REGEX_REALLOCATE(source, osize, nsize)                          ¥
  (destination = (char *) alloca (nsize),                               ¥
   bcopy (source, destination, osize),                                  ¥
   destination)

/* No need to do anything to free, after alloca.  */
#define REGEX_FREE(arg) ((void)0) /* Do nothing!  But inhibit gcc warning.  */

#endif /* not REGEX_MALLOC */

/* Define how to allocate the failure stack.  */

#if defined (REL_ALLOC) && defined (REGEX_MALLOC)

#define REGEX_ALLOCATE_STACK(size)                                      ¥
  r_alloc (&failure_stack_ptr, (size))
#define REGEX_REALLOCATE_STACK(source, osize, nsize)                    ¥
  r_re_alloc (&failure_stack_ptr, (nsize))
#define REGEX_FREE_STACK(ptr)                                           ¥
  r_alloc_free (&failure_stack_ptr)

#else /* not using relocating allocator */

#ifdef REGEX_MALLOC

#define REGEX_ALLOCATE_STACK malloc
```

```
#define REGEX_REALLOCATE_STACK(source, osize, nsize) realloc (source, nsize)
#define REGEX_FREE_STACK free

#else /* not REGEX_MALLOC */

#define REGEX_ALLOCATE_STACK alloca

#define REGEX_REALLOCATE_STACK(source, osize, nsize)                    ¥
    REGEX_REALLOCATE (source, osize, nsize)
/* No need to explicitly free anything.  */
#define REGEX_FREE_STACK(arg)

#endif /* not REGEX_MALLOC */
#endif /* not using relocating allocator */



/* True if `size1' is non-NULL and PTR is pointing anywhere inside
    `string1' or just past its end.  This works if PTR is NULL, which is
    a good thing.  */
#define FIRST_STRING_P(ptr)                                    ¥
  (size1 && string1 <= (ptr) && (ptr) <= string1 + size1)

/* (Re)Allocate N items of type T using malloc, or fail.  */
#define TALLOC(n, t) ((t *) malloc ((n) * sizeof (t)))
#define RETALLOC(addr, n, t) ((addr) = (t *) realloc (addr, (n) * sizeof (t)))
#define RETALLOC_IF(addr, n, t) ¥
  if (addr) RETALLOC((addr), (n), t); else (addr) = TALLOC ((n), t)
#define REGEX_TALLOC(n, t) ((t *) REGEX_ALLOCATE ((n) * sizeof (t)))

#define BYTEWIDTH 8 /* In bits.  */

#define STREQ(s1, s2) ((strcmp (s1, s2) == 0))

#undef MAX
```

```
#undef MIN
#define MAX(a, b) ((a) > (b) ? (a) : (b))
#define MIN(a, b) ((a) < (b) ? (a) : (b))

//typedef char boolean;
#define false 0
#define true 1

static int re_match_2_internal (struct re_pattern_buffer *bufp,
        const char *string1, int size1, const char *string2,
        int size2, int pos, struct re_registers *regs, int stop);

/* These are the command codes that appear in compiled regular
     expressions.  Some opcodes are followed by argument bytes.  A
     command code can specify any interpretation whatsoever for its
     arguments.  Zero bytes may appear in the compiled regular expression.  */

typedef enum
{
   no_op = 0,

   /* Succeed right away--no more backtracking.  */
   succeed,

         /* Followed by one byte giving n, then by n literal bytes.  */
   exactn,

         /* Matches any (more or less) character.  */
   anychar,

         /* Matches any one char belonging to specified set.  First
            following byte is number of bitmap bytes.  Then come bytes
            for a bitmap saying which chars are in.  Bits in each byte
            are ordered low-bit-first.  A character is in the set if its
```

```
            bit is 1.  A character too large to have a bit in the map is
            automatically not in the set.  */
charset,

        /* Same parameters as charset, but match any character that is
            not one of those specified.  */
charset_not,

        /* Start remembering the text that is matched, for storing in a
            register.  Followed by one byte with the register number, in
            the range 0 to one less than the pattern buffer's re_nsub
            field.  Then followed by one byte with the number of groups
            inner to this one.  (This last has to be part of the
            start_memory only because we need it in the on_failure_jump
            of re_match_2.)  */
start_memory,

        /* Stop remembering the text that is matched and store it in a
            memory register.  Followed by one byte with the register
            number, in the range 0 to one less than `re_nsub' in the
            pattern buffer, and one byte with the number of inner groups,
            just like `start_memory'.  (We need the number of inner
            groups here because we don't have any easy way of finding the
            corresponding start_memory when we're at a stop_memory.)  */
stop_memory,

        /* Match a duplicate of something remembered. Followed by one
            byte containing the register number.  */
duplicate,

        /* Fail unless at beginning of line.  */
begline,

        /* Fail unless at end of line.  */
```

```
endline,

        /* Succeeds if at beginning of buffer (if emacs) or at beginning
           of string to be matched (if not).  */
begbuf,

        /* Analogously, for end of buffer/string.  */
endbuf,

        /* Followed by two byte relative address to which to jump.  */
jump,

        /* Same as jump, but marks the end of an alternative.  */
jump_past_alt,

        /* Followed by two-byte relative address of place to resume at
           in case of failure.  */
on_failure_jump,

        /* Like on_failure_jump, but pushes a placeholder instead of the
           current string position when executed.  */
on_failure_keep_string_jump,

        /* Throw away latest failure point and then jump to following
           two-byte relative address.  */
pop_failure_jump,

        /* Change to pop_failure_jump if know won't have to backtrack to
           match; otherwise change to jump.  This is used to jump
           back to the beginning of a repeat.  If what follows this jump
           clearly won't match what the repeat does, such that we can be
           sure that there is no use backtracking out of repetitions
           already matched, then we change it to a pop_failure_jump.
           Followed by two-byte address.  */
```

```
maybe_pop_jump,

        /* Jump to following two-byte address, and push a dummy failure
           point. This failure point will be thrown away if an attempt
           is made to use it for a failure.  A `+' construct makes this
           before the first repeat.  Also used as an intermediary kind
           of jump when compiling an alternative.  */
dummy_failure_jump,

        /* Push a dummy failure point and continue.  Used at the end of
           alternatives.  */
push_dummy_failure,

        /* Followed by two-byte relative address and two-byte number n.
           After matching N times, jump to the address upon failure.  */
succeed_n,

        /* Followed by two-byte relative address, and two-byte number n.
           Jump to the address N times, then fail.  */
jump_n,

        /* Set the following two-byte relative address to the
           subsequent two-byte number.  The address *includes* the two
           bytes of number.  */
set_number_at,

wordchar,       /* Matches any word-constituent character.  */
notwordchar,    /* Matches any char that is not a word-constituent.  */

wordbeg,        /* Succeeds if at word beginning.  */
wordend,        /* Succeeds if at word end.  */

wordbound,      /* Succeeds if at a word boundary.  */
notwordbound /* Succeeds if not at a word boundary.  */
```

```
#ifdef emacs
   ,before_dot,      /* Succeeds if before point.   */
   at_dot,           /* Succeeds if at point.   */
   after_dot,        /* Succeeds if after point.   */

         /* Matches any character whose syntax is specified.   Followed by
            a byte which contains a syntax code, e.g., Sword.   */
   syntaxspec,

         /* Matches any character whose syntax is not that specified.   */
   notsyntaxspec
#endif /* emacs */
} re_opcode_t;
```

```
/* Common operations on the compiled pattern.  */

/* Store NUMBER in two contiguous bytes starting at DESTINATION.  */

#define STORE_NUMBER(destination, number)                               ¥
  do {                                                                  ¥
    (destination)[0] = (number) & 0377;                                 ¥
    (destination)[1] = (number) >> 8;                                   ¥
  } while (0)

/* Same as STORE_NUMBER, except increment DESTINATION to
    the byte after where the number is stored.  Therefore, DESTINATION
    must be an lvalue.  */

#define STORE_NUMBER_AND_INCR(destination, number)                              ¥
  do {                                                                  ¥
    STORE_NUMBER (destination, number);                                         ¥
    (destination) += 2;                                                 ¥
  } while (0)

/* Put into DESTINATION a number stored in two contiguous bytes starting
    at SOURCE.  */

#define EXTRACT_NUMBER(destination, source)                             ¥
  do {                                                                  ¥
    (destination) = *(source) & 0377;                                   ¥
    (destination) += SIGN_EXTEND_CHAR (*((source) + 1)) << 8;           ¥
  } while (0)

#ifdef DEBUG
static void extract_number _RE_ARGS ((int *dest, unsigned char *source));
static void
extract_number (dest, source)
```

```
    int *dest;
    unsigned char *source;
{
  int temp = SIGN_EXTEND_CHAR (*(source + 1));
  *dest = *source & 0377;
  *dest += temp << 8;
}

#ifndef EXTRACT_MACROS /* To debug the macros.  */
#undef EXTRACT_NUMBER
#define EXTRACT_NUMBER(dest, src) extract_number (&dest, src)
#endif /* not EXTRACT_MACROS */

#endif /* DEBUG */

/* Same as EXTRACT_NUMBER, except increment SOURCE to after the number.
   SOURCE must be an lvalue.  */

#define EXTRACT_NUMBER_AND_INCR(destination, source)                    ¥
  do {                                                                  ¥
    EXTRACT_NUMBER (destination, source);                               ¥
    (source) += 2;                                                      ¥
  } while (0)

#ifdef DEBUG
static void extract_number_and_incr _RE_ARGS ((int *destination,
                                                unsigned char **source));
static void
extract_number_and_incr (destination, source)
    int *destination;
    unsigned char **source;
{
  extract_number (destination, *source);
  *source += 2;
```

```
}

#ifndef EXTRACT_MACROS
#undef EXTRACT_NUMBER_AND_INCR
#define EXTRACT_NUMBER_AND_INCR(dest, src) ¥
   extract_number_and_incr (&dest, &src)
#endif /* not EXTRACT_MACROS */

#endif /* DEBUG */
```

```
/* If DEBUG is defined, Regex prints many voluminous messages about what
   it is doing (if the variable `debug' is nonzero).  If linked with the
   main program in `iregex.c', you can enter patterns and strings
   interactively.  And if linked with the main program in `main.c' and
   the other test files, you can run the already-written tests.  */

#ifdef DEBUG

/* We use standard I/O for debugging.  */
#include <stdio.h>

/* It is useful to test things that ``must'' be true when debugging.  */
#include <assert.h>

static int debug = 0;

#define DEBUG_STATEMENT(e) e
#define DEBUG_PRINT1(x) if (debug) printf (x)
#define DEBUG_PRINT2(x1, x2) if (debug) printf (x1, x2)
#define DEBUG_PRINT3(x1, x2, x3) if (debug) printf (x1, x2, x3)
#define DEBUG_PRINT4(x1, x2, x3, x4) if (debug) printf (x1, x2, x3, x4)
#define DEBUG_PRINT_COMPILED_PATTERN(p, s, e)                          ¥
  if (debug) print_partial_compiled_pattern (s, e)
#define DEBUG_PRINT_DOUBLE_STRING(w, s1, sz1, s2, sz2)                 ¥
  if (debug) print_double_string (w, s1, sz1, s2, sz2)


/* Print the fastmap in human-readable form.  */

void
print_fastmap (fastmap)
    char *fastmap;
{
```

```
  unsigned was_a_range = 0;
  unsigned i = 0;

  while (i < (1 << BYTEWIDTH))
    {
      if (fastmap[i++])
        {
          was_a_range = 0;
          putchar (i - 1);
          while (i < (1 << BYTEWIDTH)  &&  fastmap[i])
            {
              was_a_range = 1;
              i++;
            }
          if (was_a_range)
            {
              printf ("-");
              putchar (i - 1);
            }
        }
    }
  putchar ('¥n');
}


/* Print a compiled pattern string in human-readable form, starting at
    the START pointer into it and ending just before the pointer END.  */

void
print_partial_compiled_pattern (start, end)
     unsigned char *start;
     unsigned char *end;
{
  int mcnt, mcnt2;
```

```
unsigned char *p1;
unsigned char *p = start;
unsigned char *pend = end;

if (start == NULL)
  {
    printf ("(null)¥n");
    return;
  }

/* Loop over pattern commands.    */
while (p < pend)
  {
    printf ("%d:¥t", p - start);

    switch ((re_opcode_t) *p++)
      {
      case no_op:
        printf ("/no_op");
        break;

      case exactn:
        mcnt = *p++;
        printf ("/exactn/%d", mcnt);
        do
          {
            putchar ('/');
            putchar (*p++);
          }
        while (--mcnt);
        break;

      case start_memory:
        mcnt = *p++;
```

```
    printf ("/start_memory/%d/%d", mcnt, *p++);
    break;

case stop_memory:
    mcnt = *p++;
    printf ("/stop_memory/%d/%d", mcnt, *p++);
    break;

case duplicate:
    printf ("/duplicate/%d", *p++);
    break;

case anychar:
    printf ("/anychar");
    break;

case charset:
case charset_not:
    {
      register int c, last = -100;
      register int in_range = 0;

      printf ("/charset [%s",
              (re_opcode_t) *(p - 1) == charset_not ? "^" : "");

      assert (p + *p < pend);

      for (c = 0; c < 256; c++)
        if (c / 8 < *p
            && (p[1 + (c/8)] & (1 << (c % 8))))
          {
            /* Are we starting a range?  */
            if (last + 1 == c && ! in_range)
              {
```

```
                putchar ('-');
                in_range = 1;
              }
            /* Have we broken a range?  */
            else if (last + 1 != c && in_range)
          {
                putchar (last);
                in_range = 0;
              }

            if (! in_range)
              putchar (c);

            last = c;
          }

        if (in_range)
          putchar (last);

        putchar (']');

        p += 1 + *p;
      }
      break;

  case begline:
      printf ("/begline");
      break;

  case endline:
      printf ("/endline");
      break;

  case on_failure_jump:
```

```
    extract_number_and_incr (&mcnt, &p);
    printf ("/on_failure_jump to %d", p + mcnt - start);
    break;

case on_failure_keep_string_jump:
    extract_number_and_incr (&mcnt, &p);
    printf ("/on_failure_keep_string_jump to %d", p + mcnt - start);
    break;

case dummy_failure_jump:
    extract_number_and_incr (&mcnt, &p);
    printf ("/dummy_failure_jump to %d", p + mcnt - start);
    break;

case push_dummy_failure:
    printf ("/push_dummy_failure");
    break;

case maybe_pop_jump:
    extract_number_and_incr (&mcnt, &p);
    printf ("/maybe_pop_jump to %d", p + mcnt - start);
    break;

case pop_failure_jump:
    extract_number_and_incr (&mcnt, &p);
    printf ("/pop_failure_jump to %d", p + mcnt - start);
    break;

case jump_past_alt:
    extract_number_and_incr (&mcnt, &p);
    printf ("/jump_past_alt to %d", p + mcnt - start);
    break;

case jump:
```

```
    extract_number_and_incr (&mcnt, &p);
    printf ("/jump to %d", p + mcnt - start);
    break;

case succeed_n:
    extract_number_and_incr (&mcnt, &p);
    p1 = p + mcnt;
    extract_number_and_incr (&mcnt2, &p);
    printf ("/succeed_n to %d, %d times", p1 - start, mcnt2);
    break;

case jump_n:
    extract_number_and_incr (&mcnt, &p);
    p1 = p + mcnt;
    extract_number_and_incr (&mcnt2, &p);
    printf ("/jump_n to %d, %d times", p1 - start, mcnt2);
    break;

case set_number_at:
    extract_number_and_incr (&mcnt, &p);
    p1 = p + mcnt;
    extract_number_and_incr (&mcnt2, &p);
    printf ("/set_number_at location %d to %d", p1 - start, mcnt2);
    break;

case wordbound:
    printf ("/wordbound");
    break;

case notwordbound:
    printf ("/notwordbound");
    break;

case wordbeg:
```

```
            printf ("/wordbeg");
            break;

          case wordend:
            printf ("/wordend");

#ifdef emacs
          case before_dot:
            printf ("/before_dot");
            break;

          case at_dot:
            printf ("/at_dot");
            break;

          case after_dot:
            printf ("/after_dot");
            break;

          case syntaxspec:
            printf ("/syntaxspec");
            mcnt = *p++;
            printf ("/%d", mcnt);
            break;

          case notsyntaxspec:
            printf ("/notsyntaxspec");
            mcnt = *p++;
            printf ("/%d", mcnt);
            break;
#endif /* emacs */

          case wordchar:
            printf ("/wordchar");
```

```
            break;

          case notwordchar:
            printf ("/notwordchar");
            break;

          case begbuf:
            printf ("/begbuf");
            break;

          case endbuf:
            printf ("/endbuf");
            break;

          default:
            printf ("?%d", *(p-1));
          }

        putchar ('¥n');
      }

  printf ("%d:¥tend of pattern.¥n", p - start);
}


void
print_compiled_pattern (bufp)
     struct re_pattern_buffer *bufp;
{
  unsigned char *buffer = bufp->buffer;

  print_partial_compiled_pattern (buffer, buffer + bufp->used);
  printf ("%ld bytes used/%ld bytes allocated.¥n",
          bufp->used, bufp->allocated);
```

```
  if (bufp->fastmap_accurate && bufp->fastmap)
    {
      printf ("fastmap: ");
      print_fastmap (bufp->fastmap);
    }

  printf ("re_nsub: %d¥t", bufp->re_nsub);
  printf ("regs_alloc: %d¥t", bufp->regs_allocated);
  printf ("can_be_null: %d¥t", bufp->can_be_null);
  printf ("newline_anchor: %d¥n", bufp->newline_anchor);
  printf ("no_sub: %d¥t", bufp->no_sub);
  printf ("not_bol: %d¥t", bufp->not_bol);
  printf ("not_eol: %d¥t", bufp->not_eol);
  printf ("syntax: %lx¥n", bufp->syntax);
  /* Perhaps we should print the translate table?  */
}


void
print_double_string (where, string1, size1, string2, size2)
    const char *where;
    const char *string1;
    const char *string2;
    int size1;
    int size2;
{
  int this_char;

  if (where == NULL)
    printf ("(null)");
  else
    {
      if (FIRST_STRING_P (where))
```

```
            {
               for (this_char = where - string1; this_char < size1; this_char++)
                  putchar (string1[this_char]);

               where = string2;
            }

         for (this_char = where - string2; this_char < size2; this_char++)
            putchar (string2[this_char]);
      }
}

void
printchar (c)
        int c;
{
   putc (c, stderr);
}

#else /* not DEBUG */

#undef assert
#define assert(e)

#define DEBUG_STATEMENT(e)
#define DEBUG_PRINT1(x)
#define DEBUG_PRINT2(x1, x2)
#define DEBUG_PRINT3(x1, x2, x3)
#define DEBUG_PRINT4(x1, x2, x3, x4)
#define DEBUG_PRINT_COMPILED_PATTERN(p, s, e)
#define DEBUG_PRINT_DOUBLE_STRING(w, s1, sz1, s2, sz2)

#endif /* not DEBUG */
```

```
/* Set by `re_set_syntax' to the current regexp syntax to recognize.  Can
     also be assigned to arbitrarily: each pattern buffer stores its own
     syntax, so it can be changed between regex compilations.  */
/* This has no initializer because initialized variables in Emacs
     become read-only after dumping.  */
reg_syntax_t re_syntax_options;


/* Specify the precise syntax of regexps for compilation.  This provides
     for compatibility for various utilities which historically have
     different, incompatible syntaxes.

     The argument SYNTAX is a bit mask comprised of the various bits
     defined in regex.h.  We return the old syntax.  */

reg_syntax_t
re_set_syntax (reg_syntax_t syntax)
{
   reg_syntax_t ret = re_syntax_options;

   re_syntax_options = syntax;
#ifdef DEBUG
   if (syntax & RE_DEBUG)
     debug = 1;
   else if (debug) /* was on but now is not */
     debug = 0;
#endif /* DEBUG */
   return ret;
}
```

```
/* This table gives an error message for each of the error codes listed
   in regex.h.   Obviously the order here has to be same as there.
   POSIX doesn't require that we do anything for REG_NOERROR,
   but why not be nice?  */

static const char *re_error_msgid[] =
  {
    gettext_noop ("Success"),        /* REG_NOERROR */
    gettext_noop ("No match"),       /* REG_NOMATCH */
    gettext_noop ("Invalid regular expression"), /* REG_BADPAT */
    gettext_noop ("Invalid collation character"), /* REG_ECOLLATE */
    gettext_noop ("Invalid character class name"), /* REG_ECTYPE */
    gettext_noop ("Trailing backslash"), /* REG_EESCAPE */
    gettext_noop ("Invalid back reference"), /* REG_ESUBREG */
    gettext_noop ("Unmatched [ or [^"),      /* REG_EBRACK */
    gettext_noop ("Unmatched ( or ¥¥("), /* REG_EPAREN */
    gettext_noop ("Unmatched ¥¥{"), /* REG_EBRACE */
    gettext_noop ("Invalid content of ¥¥{¥¥}"), /* REG_BADBR */
    gettext_noop ("Invalid range end"),      /* REG_ERANGE */
    gettext_noop ("Memory exhausted"), /* REG_ESPACE */
    gettext_noop ("Invalid preceding regular expression"), /* REG_BADRPT */
    gettext_noop ("Premature end of regular expression"), /* REG_EEND */
    gettext_noop ("Regular expression too big"), /* REG_ESIZE */
    gettext_noop ("Unmatched ) or ¥¥)"), /* REG_ERPAREN */
  };
```

```
/* Avoiding alloca during matching, to placate r_alloc.  */

/* Define MATCH_MAY_ALLOCATE unless we need to make sure that the
   searching and matching functions should not call alloca.  On some
   systems, alloca is implemented in terms of malloc, and if we're
   using the relocating allocator routines, then malloc could cause a
   relocation, which might (if the strings being searched are in the
   ralloc heap) shift the data out from underneath the regexp
   routines.

   Here's another reason to avoid allocation: Emacs
   processes input from X in a signal handler; processing X input may
   call malloc; if input arrives while a matching routine is calling
   malloc, then we're scrod.  But Emacs can't just block input while
   calling matching routines; then we don't notice interrupts when
   they come in.  So, Emacs blocks input around all regexp calls
   except the matching calls, which it leaves unprotected, in the
   faith that they will not malloc.  */

/* Normally, this is fine.  */
#define MATCH_MAY_ALLOCATE

/* When using GNU C, we are not REALLY using the C alloca, no matter
   what config.h may say.  So don't take precautions for it.  */
#ifdef __GNUC__
#undef C_ALLOCA
#endif

/* The match routines may not allocate if (1) they would do it with malloc
   and (2) it's not safe for them to use malloc.
   Note that if REL_ALLOC is defined, matching would not use malloc for the
   failure stack, but we would still use it for the register vectors;
   so REL_ALLOC should not affect this.  */
```

```
#if (defined (C_ALLOCA) || defined (REGEX_MALLOC)) && defined (emacs)
#undef MATCH_MAY_ALLOCATE
#endif
```

```
/* Failure stack declarations and macros; both re_compile_fastmap and
     re_match_2 use a failure stack.  These have to be macros because of
     REGEX_ALLOCATE_STACK.  */


/* Number of failure points for which to initially allocate space
     when matching.  If this number is exceeded, we allocate more
     space, so it is not a hard limit.  */
#ifndef INIT_FAILURE_ALLOC
#define INIT_FAILURE_ALLOC 5
#endif

/* Roughly the maximum number of failure points on the stack.  Would be
     exactly that if always used MAX_FAILURE_ITEMS items each time we failed.
     This is a variable only so users of regex can assign to it; we never
     change it ourselves.  */

#ifdef INT_IS_16BIT

#if defined (MATCH_MAY_ALLOCATE)
/* 4400 was enough to cause a crash on Alpha OSF/1,
     whose default stack limit is 2mb.  */
long int re_max_failures = 4000;
#else
long int re_max_failures = 2000;
#endif

union fail_stack_elt
{
   unsigned char *pointer;
   long int integer;
};
```

```
typedef union fail_stack_elt fail_stack_elt_t;

typedef struct
{
  fail_stack_elt_t *stack;
  unsigned long int size;
  unsigned long int avail;            /* Offset of next open position.  */
} fail_stack_type;

#else /* not INT_IS_16BIT */

#if defined (MATCH_MAY_ALLOCATE)
/* 4400 was enough to cause a crash on Alpha OSF/1,
   whose default stack limit is 2mb.  */
int re_max_failures = 20000;
#else
int re_max_failures = 2000;
#endif

union fail_stack_elt
{
  unsigned char *pointer;
  int integer;
};

typedef union fail_stack_elt fail_stack_elt_t;

typedef struct
{
  fail_stack_elt_t *stack;
  unsigned size;
  unsigned avail;                     /* Offset of next open position.  */
} fail_stack_type;
```

```
#endif /* INT_IS_16BIT */

#define FAIL_STACK_EMPTY()         (fail_stack.avail == 0)
#define FAIL_STACK_PTR_EMPTY() (fail_stack_ptr->avail == 0)
#define FAIL_STACK_FULL()          (fail_stack.avail == fail_stack.size)


/* Define macros to initialize and free the failure stack.
    Do `return -2' if the alloc fails.    */

#ifdef MATCH_MAY_ALLOCATE
#define INIT_FAIL_STACK()                                                    ¥
  do {                                                                       ¥
    fail_stack.stack = (fail_stack_elt_t *)                                  ¥
      REGEX_ALLOCATE_STACK (INIT_FAILURE_ALLOC * sizeof (fail_stack_elt_t)); ¥
                                                                             ¥
    if (fail_stack.stack == NULL)                                            ¥
      return -2;                                                             ¥
                                                                             ¥
    fail_stack.size = INIT_FAILURE_ALLOC;                                    ¥
    fail_stack.avail = 0;                                                    ¥
  } while (0)

#define RESET_FAIL_STACK()    REGEX_FREE_STACK (fail_stack.stack)
#else
#define INIT_FAIL_STACK()                                                    ¥
  do {                                                                       ¥
    fail_stack.avail = 0;                                                    ¥
  } while (0)

#define RESET_FAIL_STACK()
#endif
```

```
/* Double the size of FAIL_STACK, up to approximately `re_max_failures' items.

    Return 1 if succeeds, and 0 if either ran out of memory
    allocating space for it or it was already too large.

    REGEX_REALLOCATE_STACK requires `destination' be declared.   */

#define DOUBLE_FAIL_STACK(fail_stack)                                   ¥
  ((fail_stack).size > (unsigned) (re_max_failures * MAX_FAILURE_ITEMS)  ¥
   ? 0                                                                  ¥
   : ((fail_stack).stack = (fail_stack_elt_t *)                         ¥
        REGEX_REALLOCATE_STACK ((fail_stack).stack,                     ¥
          (fail_stack).size * sizeof (fail_stack_elt_t),                ¥
          ((fail_stack).size << 1) * sizeof (fail_stack_elt_t)),        ¥
                                                                        ¥
      (fail_stack).stack == NULL                                        ¥
      ? 0                                                               ¥
      : ((fail_stack).size <<= 1,                                       ¥
         1)))


/* Push pointer POINTER on FAIL_STACK.
    Return 1 if was able to do so and 0 if ran out of memory allocating
    space to do so.   */
#define PUSH_PATTERN_OP(POINTER, FAIL_STACK)                            ¥
  ((FAIL_STACK_FULL ()                                                  ¥
    && !DOUBLE_FAIL_STACK (FAIL_STACK))                                 ¥
   ? 0                                                                  ¥
   : ((FAIL_STACK).stack[(FAIL_STACK).avail++].pointer = POINTER,       ¥
      1))

/* Push a pointer value onto the failure stack.
    Assumes the variable `fail_stack'.   Probably should only
    be called from within `PUSH_FAILURE_POINT'.   */
```

```
#define PUSH_FAILURE_POINTER(item)                                      ¥
  fail_stack.stack[fail_stack.avail++].pointer = (unsigned char *) (item)

/* This pushes an integer-valued item onto the failure stack.
    Assumes the variable `fail_stack'.  Probably should only
    be called from within `PUSH_FAILURE_POINT'.  */
#define PUSH_FAILURE_INT(item)                                          ¥
  fail_stack.stack[fail_stack.avail++].integer = (item)

/* Push a fail_stack_elt_t value onto the failure stack.
    Assumes the variable `fail_stack'.  Probably should only
    be called from within `PUSH_FAILURE_POINT'.  */
#define PUSH_FAILURE_ELT(item)                                          ¥
  fail_stack.stack[fail_stack.avail++] =  (item)

/* These three POP... operations complement the three PUSH... operations.
    All assume that `fail_stack' is nonempty.  */
#define POP_FAILURE_POINTER() fail_stack.stack[--fail_stack.avail].pointer
#define POP_FAILURE_INT() fail_stack.stack[--fail_stack.avail].integer
#define POP_FAILURE_ELT() fail_stack.stack[--fail_stack.avail]

/* Used to omit pushing failure point id's when we're not debugging.  */
#ifdef DEBUG
#define DEBUG_PUSH PUSH_FAILURE_INT
#define DEBUG_POP(item_addr) (item_addr)->integer = POP_FAILURE_INT ()
#else
#define DEBUG_PUSH(item)
#define DEBUG_POP(item_addr)
#endif


/* Push the information about the state we will need
    if we ever fail back to it.
```

```
    Requires variables fail_stack, regstart, regend, reg_info, and
    num_regs be declared.    DOUBLE_FAIL_STACK requires `destination' be
    declared.

    Does `return FAILURE_CODE' if runs out of memory.    */

#define PUSH_FAILURE_POINT(pattern_place, string_place, failure_code)    ¥
  do {                                                                   ¥
    /* Must be int, so when we don't save any registers, the arithmetic  ¥
       of 0 + -1 isn't done as unsigned.    */                          ¥
    /* Can't be int, since there is not a shred of a guarantee that int  ¥
       is wide enough to hold a value of something to which pointer can  ¥
       be assigned */                                                    ¥
    s_reg_t this_reg;                                                   ¥
                                                                        ¥
    DEBUG_STATEMENT (failure_id++);                                     ¥
    DEBUG_STATEMENT (nfailure_points_pushed++);                                 ¥
    DEBUG_PRINT2 ("¥nPUSH_FAILURE_POINT #%u:¥n", failure_id);                   ¥
    DEBUG_PRINT2 ("    Before push, next avail: %d¥n", (fail_stack).avail);¥
    DEBUG_PRINT2 ("                                size: %d¥n", (fail_stack).size);¥
                                                                        ¥
    DEBUG_PRINT2 ("    slots needed: %d¥n", NUM_FAILURE_ITEMS);                 ¥
    DEBUG_PRINT2 ("         available: %d¥n", REMAINING_AVAIL_SLOTS);           ¥
                                                                        ¥
    /* Ensure we have enough space allocated for what we will push.    */      ¥
    while (REMAINING_AVAIL_SLOTS < NUM_FAILURE_ITEMS)                           ¥
      {                                                                 ¥
        if (!DOUBLE_FAIL_STACK (fail_stack))                            ¥
          return failure_code;                                          ¥
                                                                        ¥
        DEBUG_PRINT2 ("¥n    Doubled stack; size now: %d¥n",             ¥
                      (fail_stack).size);                               ¥
        DEBUG_PRINT2 ("    slots available: %d¥n", REMAINING_AVAIL_SLOTS);¥
      }                                                                 ¥
```

```
                                                                    ¥
/* Push the info, starting with the registers.  */                  ¥
DEBUG_PRINT1 ("¥n");                                                ¥
                                                                    ¥
if (1)                                                              ¥
  for (this_reg = lowest_active_reg; this_reg <= (int)highest_active_reg; ¥
       this_reg++)                                                  ¥
    {                                                               ¥
      DEBUG_PRINT2 ("  Pushing reg: %d¥n", this_reg);               ¥
      DEBUG_STATEMENT (num_regs_pushed++);                                  ¥
                                                                    ¥
      DEBUG_PRINT2 ("    start: 0x%x¥n", regstart[this_reg]);       ¥
      PUSH_FAILURE_POINTER (regstart[this_reg]);                    ¥
                                                                    ¥
      DEBUG_PRINT2 ("    end: 0x%x¥n", regend[this_reg]);           ¥
      PUSH_FAILURE_POINTER (regend[this_reg]);                      ¥
                                                                    ¥
      DEBUG_PRINT2 ("    info: 0x%x¥n      ", reg_info[this_reg]);  ¥
      DEBUG_PRINT2 (" match_null=%d",                               ¥
                    REG_MATCH_NULL_STRING_P (reg_info[this_reg]));  ¥
      DEBUG_PRINT2 (" active=%d", IS_ACTIVE (reg_info[this_reg]));  ¥
      DEBUG_PRINT2 (" matched_something=%d",                        ¥
                    MATCHED_SOMETHING (reg_info[this_reg]));        ¥
      DEBUG_PRINT2 (" ever_matched=%d",                             ¥
                    EVER_MATCHED_SOMETHING (reg_info[this_reg]));   ¥
      DEBUG_PRINT1 ("¥n");                                                  ¥
      PUSH_FAILURE_ELT (reg_info[this_reg].word);                   ¥
    }                                                               ¥
                                                                    ¥
DEBUG_PRINT2 ("  Pushing  low active reg: %d¥n", lowest_active_reg);¥
PUSH_FAILURE_INT (lowest_active_reg);                               ¥
                                                                    ¥
DEBUG_PRINT2 ("  Pushing high active reg: %d¥n", highest_active_reg);¥
PUSH_FAILURE_INT (highest_active_reg);                              ¥
```

```
                                                                    ¥
      DEBUG_PRINT2 ("    Pushing pattern 0x%x:¥n", pattern_place);    ¥
      DEBUG_PRINT_COMPILED_PATTERN (bufp, pattern_place, pend);              ¥
      PUSH_FAILURE_POINTER (pattern_place);                         ¥
                                                                    ¥
      DEBUG_PRINT2 ("    Pushing string 0x%x: `", string_place);    ¥
      DEBUG_PRINT_DOUBLE_STRING (string_place, string1, size1, string2,     ¥
                                 size2);                            ¥
      DEBUG_PRINT1 ("'¥n");                                         ¥
      PUSH_FAILURE_POINTER (string_place);                          ¥
                                                                    ¥
      DEBUG_PRINT2 ("    Pushing failure id: %u¥n", failure_id);    ¥
      DEBUG_PUSH (failure_id);                                      ¥
    } while (0)

/* This is the number of items that are pushed and popped on the stack
     for each register.    */
#define NUM_REG_ITEMS    3

/* Individual items aside from the registers.    */
#ifdef DEBUG
#define NUM_NONREG_ITEMS 5 /* Includes failure point id.    */
#else
#define NUM_NONREG_ITEMS 4
#endif

/* We push at most this many items on the stack.    */
/* We used to use (num_regs - 1), which is the number of registers
     this regexp will save; but that was changed to 5
     to avoid stack overflow for a regexp with lots of parens.    */
#define MAX_FAILURE_ITEMS (5 * NUM_REG_ITEMS + NUM_NONREG_ITEMS)

/* We actually push this many items.    */
#define NUM_FAILURE_ITEMS                                       ¥
```

```
   (((0                                                    ¥
      ? 0 : highest_active_reg - lowest_active_reg + 1)    ¥
     * NUM_REG_ITEMS)                                          ¥
    + NUM_NONREG_ITEMS)

/* How many items can still be added to the stack without overflowing it.   */
#define REMAINING_AVAIL_SLOTS ((fail_stack).size - (fail_stack).avail)



/* Pops what PUSH_FAIL_STACK pushes.

    We restore into the parameters, all of which should be lvalues:
      STR -- the saved data position.
      PAT -- the saved pattern position.
      LOW_REG, HIGH_REG -- the highest and lowest active registers.
      REGSTART, REGEND -- arrays of string positions.
      REG_INFO -- array of information about each subexpression.

    Also assumes the variables `fail_stack' and (if debugging), `bufp',
    `pend', `string1', `size1', `string2', and `size2'.   */

#define POP_FAILURE_POINT(str, pat, low_reg, high_reg, regstart, regend, reg_info)¥
{                                                                    ¥
   DEBUG_STATEMENT (fail_stack_elt_t failure_id;)                    ¥
   s_reg_t this_reg;                                                 ¥
   const unsigned char *string_temp;                                 ¥
                                                                     ¥
   assert (!FAIL_STACK_EMPTY ());                                    ¥
                                                                     ¥
   /* Remove failure points and point to how many regs pushed.   */  ¥
   DEBUG_PRINT1 ("POP_FAILURE_POINT:¥n");                            ¥
   DEBUG_PRINT2 ("   Before pop, next avail: %d¥n", fail_stack.avail);  ¥
   DEBUG_PRINT2 ("                           size: %d¥n", fail_stack.size);  ¥
                                                                     ¥
```

```
assert (fail_stack.avail >= NUM_NONREG_ITEMS);                          ¥
                                                                        ¥
DEBUG_POP (&failure_id);                                                ¥
DEBUG_PRINT2 ("    Popping failure id: %u¥n", failure_id);              ¥
                                                                        ¥
/* If the saved string location is NULL, it came from an               ¥
     on_failure_keep_string_jump opcode, and we want to throw away the  ¥
     saved NULL, thus retaining our current position in the string.    */ ¥
string_temp = POP_FAILURE_POINTER ();                                       ¥
if (string_temp != NULL)                                                ¥
   str = (const char *) string_temp;                                    ¥
                                                                        ¥
DEBUG_PRINT2 ("    Popping string 0x%x: `", str);                       ¥
DEBUG_PRINT_DOUBLE_STRING (str, string1, size1, string2, size2);        ¥
DEBUG_PRINT1 ("'¥n");                                                       ¥
                                                                        ¥
pat = (unsigned char *) POP_FAILURE_POINTER ();                         ¥
DEBUG_PRINT2 ("    Popping pattern 0x%x:¥n", pat);                      ¥
DEBUG_PRINT_COMPILED_PATTERN (bufp, pat, pend);                             ¥
                                                                        ¥
/* Restore register info.    */                                        ¥
high_reg = (active_reg_t) POP_FAILURE_INT ();                           ¥
DEBUG_PRINT2 ("    Popping high active reg: %d¥n", high_reg);           ¥
                                                                        ¥
low_reg = (active_reg_t) POP_FAILURE_INT ();                            ¥
DEBUG_PRINT2 ("    Popping    low active reg: %d¥n", low_reg);          ¥
                                                                        ¥
if (1)                                                                  ¥
   for (this_reg = high_reg; this_reg >= (int)low_reg; this_reg--)      ¥
     {                                                                  ¥
       DEBUG_PRINT2 ("       Popping reg: %d¥n", this_reg);             ¥
                                                                        ¥
       reg_info[this_reg].word = POP_FAILURE_ELT ();                    ¥
       DEBUG_PRINT2 ("          info: 0x%x¥n", reg_info[this_reg]);¥
```

```
                                                                    ¥
        regend[this_reg] = (const char *) POP_FAILURE_POINTER ();   ¥
        DEBUG_PRINT2 ("          end: 0x%x¥n", regend[this_reg]);   ¥
                                                                    ¥
        regstart[this_reg] = (const char *) POP_FAILURE_POINTER (); ¥
        DEBUG_PRINT2 ("          start: 0x%x¥n", regstart[this_reg]); ¥
      }                                                             ¥
  else                                                              ¥
    {                                                               ¥
      for (this_reg = highest_active_reg; this_reg > (int)high_reg; this_reg--) ¥
        {                                                           ¥
          reg_info[this_reg].word.integer = 0;                      ¥
          regend[this_reg] = 0;                                     ¥
          regstart[this_reg] = 0;                                   ¥
        }                                                           ¥
      highest_active_reg = high_reg;                                ¥
    }                                                               ¥
                                                                    ¥
  set_regs_matched_done = 0;                                        ¥
  DEBUG_STATEMENT (nfailure_points_popped++);                               ¥
} /* POP_FAILURE_POINT */
```

```
/* Structure for per-register (a.k.a. per-group) information.
     Other register information, such as the
     starting and ending positions (which are addresses), and the list of
     inner groups (which is a bits list) are maintained in separate
     variables.

     We are making a (strictly speaking) nonportable assumption here: that
     the compiler will pack our bit fields into something that fits into
     the type of `word', i.e., is something that fits into one item on the
     failure stack.   */


/* Declarations and macros for re_match_2.   */

typedef union
{
   fail_stack_elt_t word;
   struct
   {
         /* This field is one if this group can match the empty string,
               zero if not.   If not yet determined,   `MATCH_NULL_UNSET_VALUE'.   */
#define MATCH_NULL_UNSET_VALUE 3
       unsigned match_null_string_p : 2;
       unsigned is_active : 1;
       unsigned matched_something : 1;
       unsigned ever_matched_something : 1;
   } bits;
} register_info_type;

#define REG_MATCH_NULL_STRING_P(R)   ((R).bits.match_null_string_p)
#define IS_ACTIVE(R)   ((R).bits.is_active)
#define MATCHED_SOMETHING(R)   ((R).bits.matched_something)
#define EVER_MATCHED_SOMETHING(R)   ((R).bits.ever_matched_something)
```

```
/* Call this when have matched a real character; it sets `matched' flags
    for the subexpressions which we are currently inside.  Also records
    that those subexprs have matched.  */
#define SET_REGS_MATCHED()                                          ¥
  do                                                                ¥
    {                                                               ¥
      if (!set_regs_matched_done)                                   ¥
        {                                                           ¥
          active_reg_t r;                                           ¥
          set_regs_matched_done = 1;                                ¥
          for (r = lowest_active_reg; r <= highest_active_reg; r++) ¥
            {                                                       ¥
              MATCHED_SOMETHING (reg_info[r])                                 ¥
                = EVER_MATCHED_SOMETHING (reg_info[r])                        ¥
                = 1;                                                ¥
            }                                                       ¥
        }                                                           ¥
    }                                                               ¥
  while (0)

/* Registers are set to a sentinel when they haven't yet matched.  */
static char reg_unset_dummy;
#define REG_UNSET_VALUE (&reg_unset_dummy)
#define REG_UNSET(e) ((e) == REG_UNSET_VALUE)
```

```
/* Subroutine declarations and macros for regex_compile.  */

static reg_errcode_t regex_compile _RE_ARGS ((const char *pattern, size_t size,
                                               reg_syntax_t syntax,
                                               struct re_pattern_buffer *bufp));
static void store_op1(re_opcode_t op, unsigned char *loc, int arg);
static void store_op2(re_opcode_t op, unsigned char *loc,
                                          int arg1, int arg2);
static void insert_op1(re_opcode_t op, unsigned char *loc,
                                           int arg, unsigned char *end);
static void insert_op2(re_opcode_t op, unsigned char *loc,
                                           int arg1, int arg2, unsigned char *end);
static boolean at_begline_loc_p(const char *pattern, const char *p,
                                                 reg_syntax_t syntax);
static boolean at_endline_loc_p(const char *p, const char *pend,
                                                 reg_syntax_t syntax);
static reg_errcode_t compile_range(const char **p_ptr,
                                               const char *pend,
                                               char *translate,
                                               reg_syntax_t syntax,
                                               unsigned char *b);

/* Fetch the next character in the uncompiled pattern---translating it
    if necessary.  Also cast from a signed character in the constant
    string passed to us by the user to an unsigned char that we can use
    as an array index (in, e.g., `translate').  */
#ifndef PATFETCH
#define PATFETCH(c)                                                    ¥
  do {if (p == pend) return REG_EEND;                                  ¥
    c = (unsigned char) *p++;                                          ¥
    if (translate) c = (unsigned char) translate[c];                   ¥
  } while (0)
#endif
```

```
/* Fetch the next character in the uncompiled pattern, with no
     translation.    */
#define PATFETCH_RAW(c)                                                    ¥
   do {if (p == pend) return REG_EEND;                                     ¥
       c = (unsigned char) *p++;                                           ¥
   } while (0)

/* Go backwards one character in the pattern.    */
#define PATUNFETCH p--



/* If `translate' is non-null, return translate[D], else just D.    We
     cast the subscript to translate because some data is declared as
     `char *', to avoid warnings when a string constant is passed.    But
     when we use a character as a subscript we must make it unsigned.    */
#ifndef TRANSLATE
#define TRANSLATE(d) ¥
   (translate ? (char) translate[(unsigned char) (d)] : (d))
#endif



/* Macros for outputting the compiled pattern into `buffer'.    */

/* If the buffer isn't allocated when it comes in, use this.    */
#define INIT_BUF_SIZE    32

/* Make sure we have at least N more bytes of space in buffer.    */
#define GET_BUFFER_SPACE(n)                                                ¥
       while ((unsigned long) (b - bufp->buffer + (n)) > bufp->allocated)  ¥
          EXTEND_BUFFER ()

/* Make sure we have one more byte of buffer space and then add C to it.    */
#define BUF_PUSH(c)                                                        ¥
```

```
  do {                                                    ¥
    GET_BUFFER_SPACE (1);                                 ¥
    *b++ = (unsigned char) (c);                           ¥
  } while (0)


/* Ensure we have two more bytes of buffer space and then append C1 and C2.  */
#define BUF_PUSH_2(c1, c2)                                ¥
  do {                                                    ¥
    GET_BUFFER_SPACE (2);                                 ¥
    *b++ = (unsigned char) (c1);                          ¥
    *b++ = (unsigned char) (c2);                          ¥
  } while (0)


/* As with BUF_PUSH_2, except for three bytes.  */
#define BUF_PUSH_3(c1, c2, c3)                            ¥
  do {                                                    ¥
    GET_BUFFER_SPACE (3);                                 ¥
    *b++ = (unsigned char) (c1);                          ¥
    *b++ = (unsigned char) (c2);                          ¥
    *b++ = (unsigned char) (c3);                          ¥
  } while (0)


/* Store a jump with opcode OP at LOC to location TO.  We store a
    relative address offset by the three bytes the jump itself occupies.  */
#define STORE_JUMP(op, loc, to) ¥
  store_op1 (op, loc, (int) ((to) - (loc) - 3))

/* Likewise, for a two-argument jump.  */
#define STORE_JUMP2(op, loc, to, arg) ¥
  store_op2 (op, loc, (int) ((to) - (loc) - 3), arg)
```

```
/* Like `STORE_JUMP', but for inserting.  Assume `b' is the buffer end.  */
#define INSERT_JUMP(op, loc, to) ¥
  insert_op1 (op, loc, (int) ((to) - (loc) - 3), b)

/* Like `STORE_JUMP2', but for inserting.  Assume `b' is the buffer end.  */
#define INSERT_JUMP2(op, loc, to, arg) ¥
  insert_op2 (op, loc, (int) ((to) - (loc) - 3), arg, b)



/* This is not an arbitrary limit: the arguments which represent offsets
    into the pattern are two bytes long.  So if 2^16 bytes turns out to
    be too small, many things would have to change.  */
/* Any other compiler which, like MSC, has allocation limit below 2^16
    bytes will have to use approach similar to what was done below for
    MSC and drop MAX_BUF_SIZE a bit.  Otherwise you may end up
    reallocating to 0 bytes.  Such thing is not going to work too well.
    You have been warned!!  */
#if defined(_MSC_VER)  && !defined(WIN32)
/* Microsoft C 16-bit versions limit malloc to approx 65512 bytes.
    The REALLOC define eliminates a flurry of conversion warnings,
    but is not required. */
#define MAX_BUF_SIZE  65500L
#define REALLOC(p,s) realloc ((p), (size_t) (s))
#else
#define MAX_BUF_SIZE (1L << 16)
#define REALLOC(p,s) realloc ((p), (s))
#endif

/* Extend the buffer by twice its current size via realloc and
    reset the pointers that pointed into the old block to point to the
    correct places in the new one.  If extending the buffer results in it
    being larger than MAX_BUF_SIZE, then flag memory exhausted.  */
#define EXTEND_BUFFER()                                                 ¥
  do {                                                                  ¥
```

```
      unsigned char *old_buffer = bufp->buffer;                          ¥
      if (bufp->allocated == MAX_BUF_SIZE)                               ¥
        return REG_ESIZE;                                                      ¥
      bufp->allocated <<= 1;                                             ¥
      if (bufp->allocated > MAX_BUF_SIZE)                                ¥
        bufp->allocated = MAX_BUF_SIZE;                                        ¥
      bufp->buffer = (unsigned char *) REALLOC (bufp->buffer, bufp->allocated);¥
      if (bufp->buffer == NULL)                                          ¥
        return REG_ESPACE;                                               ¥
      /* If the buffer moved, move all the pointers into it.  */         ¥
      if (old_buffer != bufp->buffer)                                    ¥
        {                                                                ¥
          b = (b - old_buffer) + bufp->buffer;                           ¥
          begalt = (begalt - old_buffer) + bufp->buffer;                 ¥
          if (fixup_alt_jump)                                            ¥
            fixup_alt_jump = (fixup_alt_jump - old_buffer) + bufp->buffer;¥
          if (laststart)                                                 ¥
            laststart = (laststart - old_buffer) + bufp->buffer;         ¥
          if (pending_exact)                                             ¥
            pending_exact = (pending_exact - old_buffer) + bufp->buffer; ¥
        }                                                                ¥
    } while (0)


/* Since we have one byte reserved for the register number argument to
     {start,stop}_memory, the maximum number of groups we can report
     things about is what fits in that byte.  */
#define MAX_REGNUM 255

/* But patterns can have more than `MAX_REGNUM' registers.  We just
     ignore the excess.  */
typedef unsigned regnum_t;
```

```
/* Macros for the compile stack.  */

/* Since offsets can go either forwards or backwards, this type needs to
     be able to hold values from -(MAX_BUF_SIZE - 1) to MAX_BUF_SIZE - 1.  */
/* int may be not enough when sizeof(int) == 2.  */
typedef long pattern_offset_t;

typedef struct
{
  pattern_offset_t begalt_offset;
  pattern_offset_t fixup_alt_jump;
  pattern_offset_t inner_group_offset;
  pattern_offset_t laststart_offset;
  regnum_t regnum;
} compile_stack_elt_t;



typedef struct
{
  compile_stack_elt_t *stack;
  unsigned size;
  unsigned avail;                       /* Offset of next open position.  */
} compile_stack_type;



#define INIT_COMPILE_STACK_SIZE 32

#define COMPILE_STACK_EMPTY  (compile_stack.avail == 0)
#define COMPILE_STACK_FULL  (compile_stack.avail == compile_stack.size)

/* The next available element.  */
#define COMPILE_STACK_TOP (compile_stack.stack[compile_stack.avail])
```

```
/* Set the bit for character C in a list.  */
#define SET_LIST_BIT(c)                                         ¥
  (b[((unsigned char) (c)) / BYTEWIDTH]                         ¥
    |= 1 << (((unsigned char) c) % BYTEWIDTH))



/* Get the next unsigned number in the uncompiled pattern.  */
#define GET_UNSIGNED_NUMBER(num)                                                ¥
  { if (p != pend)                                                      ¥
      {                                                                 ¥
        PATFETCH (c);                                                   ¥
        while (ISDIGIT (c))                                             ¥
          {                                                             ¥
            if (num < 0)                                                ¥
                num = 0;                                                ¥
            num = num * 10 + c - '0';                                   ¥
            if (p == pend)                                              ¥
                break;                                                  ¥
            PATFETCH (c);                                               ¥
          }                                                             ¥
        }                                                               ¥
    }

#if defined _LIBC || (defined HAVE_WCTYPE_H && defined HAVE_WCHAR_H)
/* The GNU C library provides support for user-defined character classes
    and the functions from ISO C amendement 1.  */
# ifdef CHARCLASS_NAME_MAX
#  define CHAR_CLASS_MAX_LENGTH CHARCLASS_NAME_MAX
# else
/* This shouldn't happen but some implementation might still have this
    problem.  Use a reasonable default value.  */
#  define CHAR_CLASS_MAX_LENGTH 256
# endif
```

```
# define IS_CHAR_CLASS(string) wctype (string)
#else
# define CHAR_CLASS_MAX_LENGTH    6 /* Namely, `xdigit'.    */

# define IS_CHAR_CLASS(string)                                          ¥
     (STREQ (string, "alpha") || STREQ (string, "upper")                ¥
       || STREQ (string, "lower") || STREQ (string, "digit")            ¥
       || STREQ (string, "alnum") || STREQ (string, "xdigit")           ¥
       || STREQ (string, "space") || STREQ (string, "print")            ¥
       || STREQ (string, "punct") || STREQ (string, "graph")            ¥
       || STREQ (string, "cntrl") || STREQ (string, "blank"))
#endif
```

```
#ifndef MATCH_MAY_ALLOCATE

/* If we cannot allocate large objects within re_match_2_internal,
     we make the fail stack and register vectors global.
     The fail stack, we grow to the maximum size when a regexp
     is compiled.
     The register vectors, we adjust in size each time we
     compile a regexp, according to the number of registers it needs.    */

static fail_stack_type fail_stack;

/* Size with which the following vectors are currently allocated.
     That is so we can make them bigger as needed,
     but never make them smaller.    */
static int regs_allocated_size;

static const char **        regstart, **        regend;
static const char ** old_regstart, ** old_regend;
static const char **best_regstart, **best_regend;
static register_info_type *reg_info;
static const char **reg_dummy;
static register_info_type *reg_info_dummy;

/* Make the register vectors big enough for NUM_REGS registers,
     but don't make them smaller.    */

static
regex_grow_registers (num_regs)
        int num_regs;
{
   if (num_regs > regs_allocated_size)
      {
         RETALLOC_IF (regstart,      num_regs, const char *);
```

```
        RETALLOC_IF (regend,        num_regs, const char *);
        RETALLOC_IF (old_regstart, num_regs, const char *);
        RETALLOC_IF (old_regend,            num_regs, const char *);
        RETALLOC_IF (best_regstart, num_regs, const char *);
        RETALLOC_IF (best_regend,           num_regs, const char *);
        RETALLOC_IF (reg_info,      num_regs, register_info_type);
        RETALLOC_IF (reg_dummy,             num_regs, const char *);
        RETALLOC_IF (reg_info_dummy, num_regs, register_info_type);

        regs_allocated_size = num_regs;
      }
}

#endif /* not MATCH_MAY_ALLOCATE */
```

```
static boolean group_in_compile_stack(compile_stack_type
                                      compile_stack,
                                      regnum_t regnum);

/* `regex_compile' compiles PATTERN (of length SIZE) according to SYNTAX.
   Returns one of error codes defined in `regex.h', or zero for success.

   Assumes the `allocated' (and perhaps `buffer') and `translate'
   fields are set in BUFP on entry.

   If it succeeds, results are put in BUFP (if it returns an error, the
   contents of BUFP are undefined):
     `buffer' is the compiled pattern;
     `syntax' is set to SYNTAX;
     `used' is set to the length of the compiled pattern;
     `fastmap_accurate' is zero;
     `re_nsub' is the number of subexpressions in PATTERN;
     `not_bol' and `not_eol' are zero;

   The `fastmap' and `newline_anchor' fields are neither
   examined nor set.    */

/* Return, freeing storage we allocated.    */
#define FREE_STACK_RETURN(value)                ¥
  return (free (compile_stack.stack), value)

static reg_errcode_t
regex_compile (const char *pattern, size_t size, reg_syntax_t syntax, struct re_pattern_buffer *bufp)
{
  /* We fetch characters from PATTERN here.    Even though PATTERN is
     `char *' (i.e., signed), we declare these variables as unsigned, so
     they can be reliably used as array indices.    */
  register unsigned char c, c1;
```

```
/* A random temporary spot in PATTERN.  */
const char *p1;

/* Points to the end of the buffer, where we should append.  */
register unsigned char *b;

/* Keeps track of unclosed groups.  */
compile_stack_type compile_stack;

/* Points to the current (ending) position in the pattern.  */
const char *p = pattern;
const char *pend = pattern + size;

/* How to translate the characters in the pattern.  */
RE_TRANSLATE_TYPE translate = bufp->translate;

/* Address of the count-byte of the most recently inserted `exactn'
   command.  This makes it possible to tell if a new exact-match
   character can be added to that command or if the character requires
   a new `exactn' command.  */
unsigned char *pending_exact = 0;

/* Address of start of the most recently finished expression.
   This tells, e.g., postfix * where to find the start of its
   operand.  Reset at the beginning of groups and alternatives.  */
unsigned char *laststart = 0;

/* Address of beginning of regexp, or inside of last group.  */
unsigned char *begalt;

/* Place in the uncompiled pattern (i.e., the {) to
   which to go back if the interval is invalid.  */
const char *beg_interval;
```

```
  /* Address of the place where a forward jump should go to the end of
     the containing expression.  Each alternative of an `or' -- except the
     last -- ends with a forward jump of this sort.  */
  unsigned char *fixup_alt_jump = 0;

  /* Counts open-groups as they are encountered.  Remembered for the
     matching close-group on the compile stack, so the same register
     number is put in the stop_memory as the start_memory.  */
  regnum_t regnum = 0;

#ifdef DEBUG
  DEBUG_PRINT1 ("¥nCompiling pattern: ");
  if (debug)
    {
      unsigned debug_count;

      for (debug_count = 0; debug_count < size; debug_count++)
        putchar (pattern[debug_count]);
      putchar ('¥n');
    }
#endif /* DEBUG */

  /* Initialize the compile stack.  */
  compile_stack.stack = TALLOC (INIT_COMPILE_STACK_SIZE, compile_stack_elt_t);
  if (compile_stack.stack == NULL)
    return REG_ESPACE;

  compile_stack.size = INIT_COMPILE_STACK_SIZE;
  compile_stack.avail = 0;

  /* Initialize the pattern buffer.  */
  bufp->syntax = syntax;
  bufp->fastmap_accurate = 0;
```

```
  bufp->not_bol = bufp->not_eol = 0;

  /* Set `used' to zero, so that if we return an error, the pattern
     printer (for debugging) will think there's no pattern.  We reset it
     at the end.  */
  bufp->used = 0;

  /* Always count groups, whether or not bufp->no_sub is set.  */
  bufp->re_nsub = 0;

#if !defined (emacs) && !defined (SYNTAX_TABLE)
  /* Initialize the syntax table.  */
   init_syntax_once ();
#endif

  if (bufp->allocated == 0)
    {
      if (bufp->buffer)
        { /* If zero allocated, but buffer is non-null, try to realloc
             enough space.  This loses if buffer's address is bogus, but
             that is the user's responsibility.  */
          RETALLOC (bufp->buffer, INIT_BUF_SIZE, unsigned char);
        }
      else
        { /* Caller did not allocate a buffer.  Do it for them.  */
          bufp->buffer = TALLOC (INIT_BUF_SIZE, unsigned char);
        }
      if (!bufp->buffer) FREE_STACK_RETURN (REG_ESPACE);

      bufp->allocated = INIT_BUF_SIZE;
    }

  begalt = b = bufp->buffer;
```

```
/* Loop through the uncompiled pattern until we're at the end.  */
while (p != pend)
  {
    PATFETCH (c);

    switch (c)
      {
      case '^':
        {
          if (     /* If at start of pattern, it's an operator.  */
                   p == pattern + 1
                   /* If context independent, it's an operator.  */
                || syntax & RE_CONTEXT_INDEP_ANCHORS
                   /* Otherwise, depends on what's come before.  */
                || at_begline_loc_p (pattern, p, syntax))
            BUF_PUSH (begline);
          else
            goto normal_char;
        }
        break;


      case '$':
        {
          if (     /* If at end of pattern, it's an operator.  */
                   p == pend
                   /* If context independent, it's an operator.  */
                || syntax & RE_CONTEXT_INDEP_ANCHORS
                   /* Otherwise, depends on what's next.  */
                || at_endline_loc_p (p, pend, syntax))
             BUF_PUSH (endline);
           else
             goto normal_char;
         }
```

```
      break;


case '+':
case '?':
   if ((syntax & RE_BK_PLUS_QM)
         || (syntax & RE_LIMITED_OPS))
      goto normal_char;
handle_plus:
case '*':
   /* If there is no previous pattern... */
   if (!laststart)
      {
         if (syntax & RE_CONTEXT_INVALID_OPS)
            FREE_STACK_RETURN (REG_BADRPT);
         else if (!(syntax & RE_CONTEXT_INDEP_OPS))
            goto normal_char;
      }

   {
      /* Are we optimizing this jump?    */
      boolean keep_string_p = false;

      /* 1 means zero (many) matches is allowed.    */
      char zero_times_ok = 0, many_times_ok = 0;

      /* If there is a sequence of repetition chars, collapse it
         down to just one (the right one).    We can't combine
         interval operators with these because of, e.g., `a{2}*',
         which should only match an even number of `a's.    */

      for (;;)
         {
            zero_times_ok |= c != '+';
```

```
  many_times_ok |= c != '?';

  if (p == pend)
    break;

  PATFETCH (c);

  if (c == '*'
      || (!(syntax & RE_BK_PLUS_QM) && (c == '+' || c == '?')))
    ;

  else if (syntax & RE_BK_PLUS_QM    &&    c == '¥¥')
    {
      if (p == pend) FREE_STACK_RETURN (REG_EESCAPE);

      PATFETCH (c1);
      if (!(c1 == '+' || c1 == '?'))
        {
          PATUNFETCH;
          PATUNFETCH;
          break;
        }

      c = c1;
    }
  else
    {
      PATUNFETCH;
      break;
    }

  /* If we get here, we found another repeat character.    */
}
```

```
/* Star, etc. applied to an empty pattern is equivalent
     to an empty pattern.    */
if (!laststart)
   break;

/* Now we know whether or not zero matches is allowed
     and also whether or not two or more matches is allowed.    */
if (many_times_ok)
   { /* More than one repetition is allowed, so put in at the
          end a backward relative jump from `b' to before the next
          jump we're going to put in below (which jumps from
          laststart to after this jump).

          But if we are at the `*' in the exact sequence `.*¥n',
          insert an unconditional jump backwards to the .,
          instead of the beginning of the loop.    This way we only
          push a failure point once, instead of every time
          through the loop.    */
       assert (p - 1 > pattern);

       /* Allocate the space for the jump.    */
       GET_BUFFER_SPACE (3);

       /* We know we are not at the first character of the pattern,
          because laststart was nonzero.    And we've already
          incremented `p', by the way, to be the character after
          the `*'.    Do we have to do something analogous here
          for null bytes, because of RE_DOT_NOT_NULL?    */
       if (TRANSLATE (*(p - 2)) == TRANSLATE ('.')
             && zero_times_ok
             && p < pend && TRANSLATE (*p) == TRANSLATE ('¥n')
             && !(syntax & RE_DOT_NEWLINE))
          { /* We have .*¥n.    */
             STORE_JUMP (jump, b, laststart);
```

```
              keep_string_p = true;
            }
          else
            /* Anything else.    */
            STORE_JUMP (maybe_pop_jump, b, laststart - 3);

          /* We've added more stuff to the buffer.    */
          b += 3;
        }

      /* On failure, jump from laststart to b + 3, which will be the
         end of the buffer after this jump is inserted.    */
      GET_BUFFER_SPACE (3);
      INSERT_JUMP (keep_string_p ? on_failure_keep_string_jump
                                 : on_failure_jump,
                   laststart, b + 3);
      pending_exact = 0;
      b += 3;

      if (!zero_times_ok)
        {
          /* At least one repetition is required, so insert a
             `dummy_failure_jump' before the initial
             `on_failure_jump' instruction of the loop. This
             effects a skip over that instruction the first time
             we hit that loop.    */
          GET_BUFFER_SPACE (3);
          INSERT_JUMP (dummy_failure_jump, laststart, laststart + 6);
          b += 3;
        }
      }
    break;
```

```
case '.':
   laststart = b;
   BUF_PUSH (anychar);
   break;



case '[':
   {
      boolean had_char_class = false;

      if (p == pend) FREE_STACK_RETURN (REG_EBRACK);

      /* Ensure that we have enough space to push a charset: the
           opcode, the length count, and the bitset; 34 bytes in all.   */
      GET_BUFFER_SPACE (34);

      laststart = b;

      /* We test `*p == '^' twice, instead of using an if
           statement, so we only need one BUF_PUSH.   */
      BUF_PUSH (*p == '^' ? charset_not : charset);
      if (*p == '^')
        p++;

      /* Remember the first position in the bracket expression.   */
      p1 = p;

      /* Push the number of bytes in the bitmap.   */
      BUF_PUSH ((1 << BYTEWIDTH) / BYTEWIDTH);

      /* Clear the whole map.   */
      bzero (b, (1 << BYTEWIDTH) / BYTEWIDTH);

      /* charset_not matches newline according to a syntax bit.   */
```

```
if ((re_opcode_t) b[-2] == charset_not
      && (syntax & RE_HAT_LISTS_NOT_NEWLINE))
   SET_LIST_BIT ('¥n');

/* Read in characters and ranges, setting map bits.    */
for (;;)
   {
      if (p == pend) FREE_STACK_RETURN (REG_EBRACK);

      PATFETCH (c);

      /* ¥ might escape characters inside [...] and [^...].    */
      if ((syntax & RE_BACKSLASH_ESCAPE_IN_LISTS) && c == '¥¥')
        {
          if (p == pend) FREE_STACK_RETURN (REG_EESCAPE);

          PATFETCH (c1);
          SET_LIST_BIT (c1);
          continue;
        }

      /* Could be the end of the bracket expression.    If it's
          not (i.e., when the bracket expression is `[]' so
          far), the ']' character bit gets set way below.    */
      if (c == ']' && p != p1 + 1)
        break;

      /* Look ahead to see if it's a range when the last thing
          was a character class.    */
      if (had_char_class && c == '-' && *p != ']')
        FREE_STACK_RETURN (REG_ERANGE);

      /* Look ahead to see if it's a range when the last thing
          was a character: if this is a hyphen not at the
```

```
      beginning or the end of a list, then it's the range
      operator.    */
if (c == '-'
      && !(p - 2 >= pattern && p[-2] == '[')
      && !(p - 3 >= pattern && p[-3] == '[' && p[-2] == '^')
      && *p != ']')
   {
      reg_errcode_t ret
        = compile_range (&p, pend, translate, syntax, b);
      if (ret != REG_NOERROR) FREE_STACK_RETURN (ret);
   }

else if (p[0] == '-' && p[1] != ']')
   { /* This handles ranges made up of characters only.    */
      reg_errcode_t ret;

      /* Move past the `-'.    */
      PATFETCH (c1);

      ret = compile_range (&p, pend, translate, syntax, b);
      if (ret != REG_NOERROR) FREE_STACK_RETURN (ret);
   }

/* See if we're at the beginning of a possible character
      class.    */

else if (syntax & RE_CHAR_CLASSES && c == '[' && *p == ':')
   { /* Leave room for the null.    */
      char str[CHAR_CLASS_MAX_LENGTH + 1];

      PATFETCH (c);
      c1 = 0;

      /* If pattern is `[[:'.    */
```

```
                    if (p == pend) FREE_STACK_RETURN (REG_EBRACK);

                    for (;;)
                      {
                        PATFETCH (c);
                        if (c == ':' || c == ']' || p == pend
                              || c1 == CHAR_CLASS_MAX_LENGTH)
                          break;
                        str[c1++] = c;
                      }
                    str[c1] = '¥0';

                    /* If isn't a word bracketed by `[:' and:`]':
                        undo the ending character, the letters, and leave
                        the leading `:' and `[' (but set bits for them).    */
                    if (c == ':' && *p == ']')
                      {
#if defined _LIBC || (defined HAVE_WCTYPE_H && defined HAVE_WCHAR_H)
                        boolean is_lower = STREQ (str, "lower");
                        boolean is_upper = STREQ (str, "upper");
                        wctype_t wt;
                        int ch;

                        wt = wctype (str);
                        if (wt == 0)
                          FREE_STACK_RETURN (REG_ECTYPE);

                        /* Throw away the ] at the end of the character
                            class.    */
                        PATFETCH (c);

                        if (p == pend) FREE_STACK_RETURN (REG_EBRACK);

                        for (ch = 0; ch < 1 << BYTEWIDTH; ++ch)
```

```
      {
        if (iswctype (btowc (ch), wt))
          SET_LIST_BIT (ch);

        if (translate && (is_upper || is_lower)
            && (ISUPPER (ch) || ISLOWER (ch)))
          SET_LIST_BIT (ch);
      }

    had_char_class = true;
#else
    int ch;
    boolean is_alnum = STREQ (str, "alnum");
    boolean is_alpha = STREQ (str, "alpha");
    boolean is_blank = STREQ (str, "blank");
    boolean is_cntrl = STREQ (str, "cntrl");
    boolean is_digit = STREQ (str, "digit");
    boolean is_graph = STREQ (str, "graph");
    boolean is_lower = STREQ (str, "lower");
    boolean is_print = STREQ (str, "print");
    boolean is_punct = STREQ (str, "punct");
    boolean is_space = STREQ (str, "space");
    boolean is_upper = STREQ (str, "upper");
    boolean is_xdigit = STREQ (str, "xdigit");

    if (!IS_CHAR_CLASS (str))
      FREE_STACK_RETURN (REG_ECTYPE);

    /* Throw away the ] at the end of the character
       class.  */
    PATFETCH (c);

    if (p == pend) FREE_STACK_RETURN (REG_EBRACK);
```

```
                    for (ch = 0; ch < 1 << BYTEWIDTH; ch++)
                      {
                        /* This was split into 3 if's to
                           avoid an arbitrary limit in some compiler.  */
                        if (    (is_alnum  && ISALNUM (ch))
                            || (is_alpha  && ISALPHA (ch))
                            || (is_blank  && ISBLANK (ch))
                            || (is_cntrl  && ISCNTRL (ch)))
                          SET_LIST_BIT (ch);
                        if (    (is_digit  && ISDIGIT (ch))
                            || (is_graph  && ISGRAPH (ch))
                            || (is_lower  && ISLOWER (ch))
                            || (is_print  && ISPRINT (ch)))
                          SET_LIST_BIT (ch);
                        if (    (is_punct  && ISPUNCT (ch))
                            || (is_space  && ISSPACE (ch))
                            || (is_upper  && ISUPPER (ch))
                            || (is_xdigit && ISXDIGIT (ch)))
                          SET_LIST_BIT (ch);
                        if (    translate && (is_upper || is_lower)
                            && (ISUPPER (ch) || ISLOWER (ch)))
                          SET_LIST_BIT (ch);
                      }
                    had_char_class = true;
#endif  /* libc || wctype.h */
                  }
                else
                  {
                    c1++;
                    while (c1--)
                      PATUNFETCH;
                    SET_LIST_BIT ('[');
                    SET_LIST_BIT (':');
                    had_char_class = false;
```

```
                    }
                }
            else
                {
                    had_char_class = false;
                    SET_LIST_BIT (c);
                }
        }

      /* Discard any (non)matching list bytes that are all 0 at the
         end of the map.  Decrease the map-length byte too.  */
      while ((int) b[-1] > 0 && b[b[-1] - 1] == 0)
        b[-1]--;
      b += b[-1];
    }
    break;



case '(':
    if (syntax & RE_NO_BK_PARENS)
      goto handle_open;
    else
      goto normal_char;



case ')':
    if (syntax & RE_NO_BK_PARENS)
      goto handle_close;
    else
      goto normal_char;



case '¥n':
    if (syntax & RE_NEWLINE_ALT)
```

```
      goto handle_alt;
    else
      goto normal_char;


case '|':
    if (syntax & RE_NO_BK_VBAR)
      goto handle_alt;
    else
      goto normal_char;


case '{':
     if (syntax & RE_INTERVALS && syntax & RE_NO_BK_BRACES)
       goto handle_interval;
     else
       goto normal_char;


case '¥¥':
    if (p == pend) FREE_STACK_RETURN (REG_EESCAPE);

    /* Do not translate the character after the ¥, so that we can
       distinguish, e.g., ¥B from ¥b, even if we normally would
       translate, e.g., B to b.    */
    PATFETCH_RAW (c);

    switch (c)
      {
      case '(':
        if (syntax & RE_NO_BK_PARENS)
          goto normal_backslash;

      handle_open:
```

```
bufp->re_nsub++;
regnum++;

if (COMPILE_STACK_FULL)
  {
    RETALLOC (compile_stack.stack, compile_stack.size << 1,
                  compile_stack_elt_t);
    if (compile_stack.stack == NULL) return REG_ESPACE;

    compile_stack.size <<= 1;
  }

/* These are the values to restore when we hit end of this
    group.  They are all relative offsets, so that if the
    whole pattern moves because of realloc, they will still
    be valid.  */
COMPILE_STACK_TOP.begalt_offset = begalt - bufp->buffer;
COMPILE_STACK_TOP.fixup_alt_jump
  = fixup_alt_jump ? fixup_alt_jump - bufp->buffer + 1 : 0;
COMPILE_STACK_TOP.laststart_offset = b - bufp->buffer;
COMPILE_STACK_TOP.regnum = regnum;

/* We will eventually replace the 0 with the number of
    groups inner to this one.  But do not push a
    start_memory for groups beyond the last one we can
    represent in the compiled pattern.  */
if (regnum <= MAX_REGNUM)
  {
    COMPILE_STACK_TOP.inner_group_offset = b - bufp->buffer + 2;
    BUF_PUSH_3 (start_memory, regnum, 0);
  }

compile_stack.avail++;
```

```
    fixup_alt_jump = 0;
    laststart = 0;
    begalt = b;
    /* If we've reached MAX_REGNUM groups, then this open
       won't actually generate any code, so we'll have to
       clear pending_exact explicitly.    */
    pending_exact = 0;
    break;


case ')':
    if (syntax & RE_NO_BK_PARENS) goto normal_backslash;

    if (COMPILE_STACK_EMPTY)
      if (syntax & RE_UNMATCHED_RIGHT_PAREN_ORD)
        goto normal_backslash;
      else
        FREE_STACK_RETURN (REG_ERPAREN);

handle_close:
    if (fixup_alt_jump)
      { /* Push a dummy failure point at the end of the
           alternative for a possible future
           `pop_failure_jump' to pop.    See comments at
           `push_dummy_failure' in `re_match_2'.    */
        BUF_PUSH (push_dummy_failure);

        /* We allocated space for this jump when we assigned
           to `fixup_alt_jump', in the `handle_alt' case below.    */
        STORE_JUMP (jump_past_alt, fixup_alt_jump, b - 1);
      }

    /* See similar code for backslashed left paren above.    */
    if (COMPILE_STACK_EMPTY)
```

```
    if (syntax & RE_UNMATCHED_RIGHT_PAREN_ORD)
      goto normal_char;
    else
      FREE_STACK_RETURN (REG_ERPAREN);

/* Since we just checked for an empty stack above, this
     ``can't happen".    */
assert (compile_stack.avail != 0);
{
    /* We don't just want to restore into `regnum', because
         later groups should continue to be numbered higher,
         as in `(ab)c(de)' -- the second group is #2.    */
    regnum_t this_group_regnum;

    compile_stack.avail--;
    begalt = bufp->buffer + COMPILE_STACK_TOP.begalt_offset;
    fixup_alt_jump
      = COMPILE_STACK_TOP.fixup_alt_jump
          ? bufp->buffer + COMPILE_STACK_TOP.fixup_alt_jump - 1
          : 0;
    laststart = bufp->buffer + COMPILE_STACK_TOP.laststart_offset;
    this_group_regnum = COMPILE_STACK_TOP.regnum;
    /* If we've reached MAX_REGNUM groups, then this open
         won't actually generate any code, so we'll have to
         clear pending_exact explicitly.    */
    pending_exact = 0;

    /* We're at the end of the group, so now we know how many
         groups were inside this one.    */
    if (this_group_regnum <= MAX_REGNUM)
      {
        unsigned char *inner_group_loc
          = bufp->buffer + COMPILE_STACK_TOP.inner_group_offset;
```

```
            *inner_group_loc = regnum - this_group_regnum;
            BUF_PUSH_3 (stop_memory, this_group_regnum,
                        regnum - this_group_regnum);
          }
        }
      break;


case '|':                                               /* `¥|'.  */
    if (syntax & RE_LIMITED_OPS || syntax & RE_NO_BK_VBAR)
      goto normal_backslash;
handle_alt:
    if (syntax & RE_LIMITED_OPS)
      goto normal_char;

    /* Insert before the previous alternative a jump which
       jumps to this alternative if the former fails.  */
    GET_BUFFER_SPACE (3);
    INSERT_JUMP (on_failure_jump, begalt, b + 6);
    pending_exact = 0;
    b += 3;

    /* The alternative before this one has a jump after it
       which gets executed if it gets matched.  Adjust that
       jump so it will jump to this alternative's analogous
       jump (put in below, which in turn will jump to the next
       (if any) alternative's such jump, etc.).  The last such
       jump jumps to the correct final destination.  A picture:
                    _____ _____
                    |   | |   |
                    |   v |   v
                   a | b   | c

       If we are at `b', then fixup_alt_jump right now points to a
```

```
        three-byte space after `a'.  We'll put in the jump, set
        fixup_alt_jump to right after `b', and leave behind three
        bytes which we'll fill in when we get to after `c'.  */

    if (fixup_alt_jump)
      STORE_JUMP (jump_past_alt, fixup_alt_jump, b);

    /* Mark and leave space for a jump after this alternative,
        to be filled in later either by next alternative or
        when know we're at the end of a series of alternatives.  */
    fixup_alt_jump = b;
    GET_BUFFER_SPACE (3);
    b += 3;

    laststart = 0;
    begalt = b;
    break;



case '{':
    /* If ¥{ is a literal.  */
    if (!(syntax & RE_INTERVALS)
            /* If we're at `¥{' and it's not the open-interval
                operator.  */
        || ((syntax & RE_INTERVALS) && (syntax & RE_NO_BK_BRACES))
        || (p - 2 == pattern  &&  p == pend))
      goto normal_backslash;

handle_interval:
    {
      /* If got here, then the syntax allows intervals.  */

      /* At least (most) this many matches must be made.  */
      int lower_bound = -1, upper_bound = -1;
```

```
beg_interval = p - 1;

if (p == pend)
  {
    if (syntax & RE_NO_BK_BRACES)
      goto unfetch_interval;
    else
      FREE_STACK_RETURN (REG_EBRACE);
  }

GET_UNSIGNED_NUMBER (lower_bound);

if (c == ',')
  {
    GET_UNSIGNED_NUMBER (upper_bound);
    if (upper_bound < 0) upper_bound = RE_DUP_MAX;
  }
else
  /* Interval such as `{1}' => match exactly once. */
  upper_bound = lower_bound;

if (lower_bound < 0 || upper_bound > RE_DUP_MAX
    || lower_bound > upper_bound)
  {
    if (syntax & RE_NO_BK_BRACES)
      goto unfetch_interval;
    else
      FREE_STACK_RETURN (REG_BADBR);
  }

if (!(syntax & RE_NO_BK_BRACES))
  {
    if (c != '¥¥') FREE_STACK_RETURN (REG_EBRACE);
```

```
      PATFETCH (c);
    }

if (c != '}')
    {
      if (syntax & RE_NO_BK_BRACES)
        goto unfetch_interval;
      else
        FREE_STACK_RETURN (REG_BADBR);
    }

/* We just parsed a valid interval.    */

/* If it's invalid to have no preceding re.    */
if (!laststart)
    {
      if (syntax & RE_CONTEXT_INVALID_OPS)
        FREE_STACK_RETURN (REG_BADRPT);
      else if (syntax & RE_CONTEXT_INDEP_OPS)
        laststart = b;
      else
        goto unfetch_interval;
    }

/* If the upper bound is zero, don't want to succeed at
     all; jump from `laststart' to `b + 3', which will be
     the end of the buffer after we insert the jump.    */
  if (upper_bound == 0)
    {
      GET_BUFFER_SPACE (3);
      INSERT_JUMP (jump, laststart, b + 3);
      b += 3;
    }
```

```
/* Otherwise, we have a nontrivial interval.    When
     we're all done, the pattern will look like:
        set_number_at <jump count> <upper bound>
        set_number_at <succeed_n count> <lower bound>
        succeed_n <after jump addr> <succeed_n count>
        <body of loop>
        jump_n <succeed_n addr> <jump count>
     (The upper bound and `jump_n' are omitted if
     `upper_bound' is 1, though.)    */
else
   { /* If the upper bound is > 1, we need to insert
            more at the end of the loop.    */
       unsigned nbytes = 10 + (upper_bound > 1) * 10;

       GET_BUFFER_SPACE (nbytes);

       /* Initialize lower bound of the `succeed_n', even
            though it will be set during matching by its
            attendant `set_number_at' (inserted next),
            because `re_compile_fastmap' needs to know.
            Jump to the `jump_n' we might insert below.    */
       INSERT_JUMP2 (succeed_n, laststart,
                          b + 5 + (upper_bound > 1) * 5,
                          lower_bound);
       b += 5;

       /* Code to initialize the lower bound.    Insert
            before the `succeed_n'.    The `5' is the last two
            bytes of this `set_number_at', plus 3 bytes of
            the following `succeed_n'.    */
       insert_op2 (set_number_at, laststart, 5, lower_bound, b);
       b += 5;
```

```
            if (upper_bound > 1)
              { /* More than one repetition is allowed, so
                   append a backward jump to the `succeed_n'
                   that starts this interval.

                   When we've reached this during matching,
                   we'll have matched the interval once, so
                   jump back only `upper_bound - 1' times.    */
                STORE_JUMP2 (jump_n, b, laststart + 5,
                             upper_bound - 1);
                b += 5;

                /* The location we want to set is the second
                   parameter of the `jump_n'; that is `b-2' as
                   an absolute address.    `laststart' will be
                   the `set_number_at' we're about to insert;
                   `laststart+3' the number to set, the source
                   for the relative address.    But we are
                   inserting into the middle of the pattern --
                   so everything is getting moved up by 5.
                   Conclusion: (b - 2) - (laststart + 3) + 5,
                   i.e., b - laststart.

                   We insert this at the beginning of the loop
                   so that if we fail during matching, we'll
                   reinitialize the bounds.    */
                insert_op2 (set_number_at, laststart, b - laststart,
                            upper_bound - 1, b);
                b += 5;
              }
          }
        pending_exact = 0;
        beg_interval = NULL;
    }
```

```
            break;

          unfetch_interval:
            /* If an invalid interval, match the characters as literals.  */
              assert (beg_interval);
              p = beg_interval;
              beg_interval = NULL;

              /* normal_char and normal_backslash need `c'.  */
              PATFETCH (c);

              if (!(syntax & RE_NO_BK_BRACES))
                {
                  if (p > pattern  &&  p[-1] == '¥¥')
                    goto normal_backslash;
                }
              goto normal_char;

#ifdef emacs
          /* There is no way to specify the before_dot and after_dot
              operators.  rms says this is ok.  --karl  */
          case '=':
            BUF_PUSH (at_dot);
            break;

          case 's':
            laststart = b;
            PATFETCH (c);
            BUF_PUSH_2 (syntaxspec, syntax_spec_code[c]);
            break;

          case 'S':
            laststart = b;
            PATFETCH (c);
```

```
                BUF_PUSH_2 (notsyntaxspec, syntax_spec_code[c]);
                break;
#endif /* emacs */


              case 'w':
                if (re_syntax_options & RE_NO_GNU_OPS)
                  goto normal_char;
                laststart = b;
                BUF_PUSH (wordchar);
                break;


              case 'W':
                if (re_syntax_options & RE_NO_GNU_OPS)
                  goto normal_char;
                laststart = b;
                BUF_PUSH (notwordchar);
                break;


              case '<':
                if (re_syntax_options & RE_NO_GNU_OPS)
                  goto normal_char;
                BUF_PUSH (wordbeg);
                break;

              case '>':
                if (re_syntax_options & RE_NO_GNU_OPS)
                  goto normal_char;
                BUF_PUSH (wordend);
                break;

              case 'b':
```

```
    if (re_syntax_options & RE_NO_GNU_OPS)
      goto normal_char;
    BUF_PUSH (wordbound);
    break;

  case 'B':
    if (re_syntax_options & RE_NO_GNU_OPS)
      goto normal_char;
    BUF_PUSH (notwordbound);
    break;

  case '`':
    if (re_syntax_options & RE_NO_GNU_OPS)
      goto normal_char;
    BUF_PUSH (begbuf);
    break;

  case '¥':
    if (re_syntax_options & RE_NO_GNU_OPS)
      goto normal_char;
    BUF_PUSH (endbuf);
    break;

  case '1': case '2': case '3': case '4': case '5':
  case '6': case '7': case '8': case '9':
    if (syntax & RE_NO_BK_REFS)
      goto normal_char;

    c1 = c - '0';

    if (c1 > regnum)
      FREE_STACK_RETURN (REG_ESUBREG);

    /* Can't back reference to a subexpression if inside of it.  */
```

```
          if (group_in_compile_stack (compile_stack, (regnum_t) c1))
            goto normal_char;

          laststart = b;
          BUF_PUSH_2 (duplicate, c1);
          break;


        case '+':
        case '?':
          if (syntax & RE_BK_PLUS_QM)
            goto handle_plus;
          else
            goto normal_backslash;

        default:
        normal_backslash:
          /* You might think it would be useful for ¥ to mean
               not to translate; but if we don't translate it
               it will never match anything.  */
          c = TRANSLATE (c);
          goto normal_char;
        }
      break;


default:
/* Expects the character in `c'.  */
normal_char:
          /* If no exactn currently being built.  */
      if (!pending_exact

          /* If last exactn not at current position.  */
          || pending_exact + *pending_exact + 1 != b
```

```
              /* We have only one byte following the exactn for the count.  */
              || *pending_exact == (1 << BYTEWIDTH) - 1

              /* If followed by a repetition operator.  */
              || *p == '*' || *p == '^'
              || ((syntax & RE_BK_PLUS_QM)
                  ? *p == '¥¥' && (p[1] == '+' || p[1] == '?')
                  : (*p == '+' || *p == '?'))
              || ((syntax & RE_INTERVALS)
                  && ((syntax & RE_NO_BK_BRACES)
                      ? *p == '{'
                      : (p[0] == '¥¥' && p[1] == '{')))
            {
              /* Start building a new exactn.  */

              laststart = b;

              BUF_PUSH_2 (exactn, 0);
              pending_exact = b - 1;
            }

          BUF_PUSH (c);
          (*pending_exact)++;
          break;
        } /* switch (c) */
      } /* while p != pend */


/* Through the pattern now.  */

if (fixup_alt_jump)
  STORE_JUMP (jump_past_alt, fixup_alt_jump, b);
```

```
  if (!COMPILE_STACK_EMPTY)
    FREE_STACK_RETURN (REG_EPAREN);

  /* If we don't want backtracking, force success
      the first time we reach the end of the compiled pattern.    */
  if (syntax & RE_NO_POSIX_BACKTRACKING)
    BUF_PUSH (succeed);

  free (compile_stack.stack);

  /* We have succeeded; set the length of the buffer.    */
  bufp->used = b - bufp->buffer;

#ifdef DEBUG
  if (debug)
    {
      DEBUG_PRINT1 ("¥nCompiled pattern: ¥n");
      print_compiled_pattern (bufp);
    }
#endif /* DEBUG */

#ifndef MATCH_MAY_ALLOCATE
  /* Initialize the failure stack to the largest possible stack.    This
      isn't necessary unless we're trying to avoid calling alloca in
      the search and match routines.    */
  {
    int num_regs = bufp->re_nsub + 1;

    /* Since DOUBLE_FAIL_STACK refuses to double only if the current size
        is strictly greater than re_max_failures, the largest possible stack
        is 2 * re_max_failures failure points.    */
    if (fail_stack.size < (2 * re_max_failures * MAX_FAILURE_ITEMS))
      {
        fail_stack.size = (2 * re_max_failures * MAX_FAILURE_ITEMS);
```

```
#ifdef emacs
            if (! fail_stack.stack)
              fail_stack.stack
                = (fail_stack_elt_t *) xmalloc (fail_stack.size
                                                * sizeof (fail_stack_elt_t));
            else
              fail_stack.stack
                = (fail_stack_elt_t *) xrealloc (fail_stack.stack,
                                                 (fail_stack.size
                                                  * sizeof (fail_stack_elt_t)));
#else /* not emacs */
            if (! fail_stack.stack)
              fail_stack.stack
                = (fail_stack_elt_t *) malloc (fail_stack.size
                                               * sizeof (fail_stack_elt_t));
            else
              fail_stack.stack
                = (fail_stack_elt_t *) realloc (fail_stack.stack,
                                                (fail_stack.size
                                                 * sizeof (fail_stack_elt_t)));
#endif /* not emacs */
          }

        regex_grow_registers (num_regs);
      }
#endif /* not MATCH_MAY_ALLOCATE */

      return REG_NOERROR;
} /* regex_compile */
```

```
/* Subroutines for `regex_compile'.  */

/* Store OP at LOC followed by two-byte integer parameter ARG.  */

static void
store_op1 (re_opcode_t op, unsigned char *loc, int arg)
{
  *loc = (unsigned char) op;
  STORE_NUMBER (loc + 1, arg);
}


/* Like `store_op1', but for two two-byte parameters ARG1 and ARG2.  */

static void
store_op2 (re_opcode_t op, unsigned char *loc, int arg1, int arg2)
{
  *loc = (unsigned char) op;
  STORE_NUMBER (loc + 1, arg1);
  STORE_NUMBER (loc + 3, arg2);
}


/* Copy the bytes from LOC to END to open up three bytes of space at LOC
   for OP followed by two-byte integer parameter ARG.  */

static void
insert_op1 (re_opcode_t op, unsigned char *loc, int arg, unsigned char *end)
{
  register unsigned char *pfrom = end;
  register unsigned char *pto = end + 3;

  while (pfrom != loc)
```

```
      *--pto = *--pfrom;

  store_op1 (op, loc, arg);
}


/* Like `insert_op1', but for two two-byte parameters ARG1 and ARG2.  */

static void
insert_op2 (re_opcode_t op, unsigned char *loc, int arg1, int arg2, unsigned char *end)
{
  register unsigned char *pfrom = end;
  register unsigned char *pto = end + 5;

  while (pfrom != loc)
    *--pto = *--pfrom;

  store_op2 (op, loc, arg1, arg2);
}


/* P points to just after a ^ in PATTERN.  Return true if that ^ comes
   after an alternative or a begin-subexpression.  We assume there is at
   least one character before the ^.  */

static boolean
at_begline_loc_p (const char *pattern,const char *p, reg_syntax_t syntax)
{
  const char *prev = p - 2;
  boolean prev_prev_backslash = prev > pattern && prev[-1] == '¥¥';

  return
       /* After a subexpression?  */
       (*prev == '(' && (syntax & RE_NO_BK_PARENS || prev_prev_backslash))
```

```
          /* After an alternative?  */
      || (*prev == '|' && (syntax & RE_NO_BK_VBAR || prev_prev_backslash));
}


/* The dual of at_begline_loc_p.  This one is for $.  We assume there is
    at least one character after the $, i.e., `P < PEND'.  */

static boolean
at_endline_loc_p (const char *p, const char *pend, reg_syntax_t syntax)
{
  const char *next = p;
  boolean next_backslash = *next == '¥¥';
  const char *next_next = p + 1 < pend ? p + 1 : 0;

  return
          /* Before a subexpression?  */
          (syntax & RE_NO_BK_PARENS ? *next == ')'
           : next_backslash && next_next && *next_next == ')')
          /* Before an alternative?  */
      || (syntax & RE_NO_BK_VBAR ? *next == '|'
          : next_backslash && next_next && *next_next == '|');
}


/* Returns true if REGNUM is in one of COMPILE_STACK's elements and
    false if it's not.  */

static boolean
group_in_compile_stack(compile_stack_type compile_stack, regnum_t regnum)
{
  int this_element;

  for (this_element = compile_stack.avail - 1;
```

```
           this_element >= 0;
           this_element--)
        if (compile_stack.stack[this_element].regnum == regnum)
          return true;

    return false;
}


/* Read the ending character of a range (in a bracket expression) from the
     uncompiled pattern *P_PTR (which ends at PEND).  We assume the
     starting character is in `P[-2]'.  (`P[-1]' is the character `-'.)
     Then we set the translation of all bits between the starting and
     ending characters (inclusive) in the compiled pattern B.

     Return an error code.

     We use these short variable names so we can use the same macros as
     `regex_compile' itself.  */

static reg_errcode_t
compile_range (const char **p_ptr, const char *pend, RE_TRANSLATE_TYPE translate, reg_syntax_t
syntax, unsigned char *b)
{
    unsigned this_char;

    const char *p = *p_ptr;
    unsigned int range_start, range_end;

    if (p == pend)
      return REG_ERANGE;

    /* Even though the pattern is a signed `char *', we need to fetch
       with unsigned char *'s; if the high bit of the pattern character
```

```
        is set, the range endpoints will be negative if we fetch using a
        signed char *.

        We also want to fetch the endpoints without translating them; the
        appropriate translation is done in the bit-setting loop below.  */
      /* The SVR4 compiler on the 3B2 had trouble with unsigned const char *.  */
      range_start = ((const unsigned char *) p)[-2];
      range_end   = ((const unsigned char *) p)[0];

      /* Have to increment the pointer into the pattern string, so the
         caller isn't still at the ending character.  */
      (*p_ptr)++;

      /* If the start is after the end, the range is empty.  */
      if (range_start > range_end)
        return syntax & RE_NO_EMPTY_RANGES ? REG_ERANGE : REG_NOERROR;

      /* Here we see why `this_char' has to be larger than an `unsigned
         char' -- the range is inclusive, so if `range_end' == 0xff
         (assuming 8-bit characters), we would otherwise go into an infinite
         loop, since all characters <= 0xff.  */
      for (this_char = range_start; this_char <= range_end; this_char++)
        {
          SET_LIST_BIT (TRANSLATE (this_char));
        }

      return REG_NOERROR;
    }
```

```
/* re_compile_fastmap computes a ``fastmap'' for the compiled pattern in
     BUFP.  A fastmap records which of the (1 << BYTEWIDTH) possible
     characters can start a string that matches the pattern.  This fastmap
     is used by re_search to skip quickly over impossible starting points.

     The caller must supply the address of a (1 << BYTEWIDTH)-byte data
     area as BUFP->fastmap.

     We set the `fastmap', `fastmap_accurate', and `can_be_null' fields in
     the pattern buffer.

     Returns 0 if we succeed, -2 if an internal error.   */

int
re_compile_fastmap (struct re_pattern_buffer *bufp)
{
  int j, k;
#ifdef MATCH_MAY_ALLOCATE
  fail_stack_type fail_stack;
#endif
#ifndef REGEX_MALLOC
  char *destination;
#endif
  /* We don't push any register information onto the failure stack.  */
  unsigned num_regs = 0;

  register char *fastmap = bufp->fastmap;
  unsigned char *pattern = bufp->buffer;
  unsigned char *p = pattern;
  register unsigned char *pend = pattern + bufp->used;

#ifdef REL_ALLOC
  /* This holds the pointer to the failure stack, when
```

```
        it is allocated relocatably.    */
    fail_stack_elt_t *failure_stack_ptr;
#endif

    /* Assume that each path through the pattern can be null until
        proven otherwise.    We set this false at the bottom of switch
        statement, to which we get only if a particular path doesn't
        match the empty string.    */
    boolean path_can_be_null = true;

    /* We aren't doing a `succeed_n' to begin with.    */
    boolean succeed_n_p = false;

    assert (fastmap != NULL && p != NULL);

    INIT_FAIL_STACK ();
    bzero (fastmap, 1 << BYTEWIDTH);    /* Assume nothing's valid.    */
    bufp->fastmap_accurate = 1;             /* It will be when we're done.    */
    bufp->can_be_null = 0;

    while (1)
      {
        if (p == pend || *p == succeed)
          {
            /* We have reached the (effective) end of pattern.    */
            if (!FAIL_STACK_EMPTY ())
              {
                bufp->can_be_null |= path_can_be_null;

                /* Reset for next path.    */
                path_can_be_null = true;

                p = fail_stack.stack[--fail_stack.avail].pointer;
```

```
            continue;
          }
        else
          break;
    }

/* We should never be about to go beyond the end of the pattern.    */
assert (p < pend);

switch (SWITCH_ENUM_CAST ((re_opcode_t) *p++))
    {

    /* I guess the idea here is to simply not bother with a fastmap
         if a backreference is used, since it's too hard to figure out
         the fastmap for the corresponding group.    Setting
         `can_be_null' stops `re_search_2' from using the fastmap, so
         that is all we do.    */
    case duplicate:
      bufp->can_be_null = 1;
      goto done;


/* Following are the cases which match a character.    These end
     with `break'.    */

    case exactn:
      fastmap[p[1]] = 1;
      break;


    case charset:
      for (j = *p++ * BYTEWIDTH - 1; j >= 0; j--)
        if (p[j / BYTEWIDTH] & (1 << (j % BYTEWIDTH)))
          fastmap[j] = 1;
```

```
    break;


case charset_not:
    /* Chars beyond end of map must be allowed.    */
    for (j = *p * BYTEWIDTH; j < (1 << BYTEWIDTH); j++)
      fastmap[j] = 1;

    for (j = *p++ * BYTEWIDTH - 1; j >= 0; j--)
      if (!(p[j / BYTEWIDTH] & (1 << (j % BYTEWIDTH))))
        fastmap[j] = 1;
    break;


case wordchar:
    for (j = 0; j < (1 << BYTEWIDTH); j++)
      if (SYNTAX (j) == Sword)
        fastmap[j] = 1;
    break;


case notwordchar:
    for (j = 0; j < (1 << BYTEWIDTH); j++)
      if (SYNTAX (j) != Sword)
        fastmap[j] = 1;
    break;


case anychar:
    {
      int fastmap_newline = fastmap['¥n'];

      /* `.' matches anything ...    */
      for (j = 0; j < (1 << BYTEWIDTH); j++)
```

```
                fastmap[j] = 1;

              /* ... except perhaps newline.    */
              if (!(bufp->syntax & RE_DOT_NEWLINE))
                fastmap['¥n'] = fastmap_newline;

              /* Return if we have already set `can_be_null'; if we have,
                 then the fastmap is irrelevant.    Something's wrong here.    */
              else if (bufp->can_be_null)
                goto done;

              /* Otherwise, have to check alternative paths.    */
              break;
            }

#ifdef emacs
          case syntaxspec:
            k = *p++;
            for (j = 0; j < (1 << BYTEWIDTH); j++)
              if (SYNTAX (j) == (enum syntaxcode) k)
                fastmap[j] = 1;
            break;


          case notsyntaxspec:
            k = *p++;
            for (j = 0; j < (1 << BYTEWIDTH); j++)
              if (SYNTAX (j) != (enum syntaxcode) k)
                fastmap[j] = 1;
            break;


        /* All cases after this match the empty string.    These end with
           `continue'.    */
```

```
            case before_dot:
            case at_dot:
            case after_dot:
              continue;
#endif /* emacs */


            case no_op:
            case begline:
            case endline:
            case begbuf:
            case endbuf:
            case wordbound:
            case notwordbound:
            case wordbeg:
            case wordend:
            case push_dummy_failure:
              continue;


            case jump_n:
            case pop_failure_jump:
            case maybe_pop_jump:
            case jump:
            case jump_past_alt:
            case dummy_failure_jump:
              EXTRACT_NUMBER_AND_INCR (j, p);
              p += j;
              if (j > 0)
                continue;

              /* Jump backward implies we just went through the body of a
```

```
        loop and matched nothing.  Opcode jumped to should be
        `on_failure_jump' or `succeed_n'.  Just treat it like an
        ordinary jump.  For a * loop, it has pushed its failure
        point already; if so, discard that as redundant.  */
    if ((re_opcode_t) *p != on_failure_jump
          && (re_opcode_t) *p != succeed_n)
        continue;

    p++;
    EXTRACT_NUMBER_AND_INCR (j, p);
    p += j;

    /* If what's on the stack is where we are now, pop it.  */
    if (!FAIL_STACK_EMPTY ()
          && fail_stack.stack[fail_stack.avail - 1].pointer == p)
        fail_stack.avail--;

    continue;


case on_failure_jump:
case on_failure_keep_string_jump:
handle_on_failure_jump:
    EXTRACT_NUMBER_AND_INCR (j, p);

    /* For some patterns, e.g., `(a?)?', `p+j' here points to the
        end of the pattern.  We don't want to push such a point,
        since when we restore it above, entering the switch will
        increment `p' past the end of the pattern.  We don't need
        to push such a point since we obviously won't find any more
        fastmap entries beyond `pend'.  Such a pattern can match
        the null string, though.  */
    if (p + j < pend)
      {
```

```
        if (!PUSH_PATTERN_OP (p + j, fail_stack))
          {
            RESET_FAIL_STACK ();
            return -2;
          }
      }
    else
      bufp->can_be_null = 1;

    if (succeed_n_p)
      {
        EXTRACT_NUMBER_AND_INCR (k, p);      /* Skip the n.   */
        succeed_n_p = false;
      }

    continue;


  case succeed_n:
    /* Get to the number of times to succeed.   */
    p += 2;

    /* Increment p past the n for when k != 0.   */
    EXTRACT_NUMBER_AND_INCR (k, p);
    if (k == 0)
      {
        p -= 4;
        succeed_n_p = true;   /* Spaghetti code alert.   */
        goto handle_on_failure_jump;
      }
    continue;


  case set_number_at:
```

```
              p += 4;
              continue;


            case start_memory:
            case stop_memory:
              p += 2;
              continue;


            default:
              abort (); /* We have listed all the cases.  */
            } /* switch *p++ */

          /* Getting here means we have found the possible starting
              characters for one path of the pattern -- and that the empty
              string does not match.  We need not follow this path further.
              Instead, look at the next alternative (remembered on the
              stack), or quit if no more.  The test at the top of the loop
              does these things.  */
          path_can_be_null = false;
          p = pend;
        } /* while p */

      /* Set `can_be_null' for the last path (also the first path, if the
          pattern is empty).  */
      bufp->can_be_null |= path_can_be_null;

    done:
      RESET_FAIL_STACK ();
      return 0;
    } /* re_compile_fastmap */
```

```
/* Set REGS to hold NUM_REGS registers, storing them in STARTS and
    ENDS.  Subsequent matches using PATTERN_BUFFER and REGS will use
    this memory for recording register information.  STARTS and ENDS
    must be allocated using the malloc library routine, and must each
    be at least NUM_REGS * sizeof (regoff_t) bytes long.

    If NUM_REGS == 0, then subsequent matches should allocate their own
    register data.

    Unless this function is called, the first search or match using
    PATTERN_BUFFER will allocate its own register data, without
    freeing the old data.  */

void
re_set_registers (struct re_pattern_buffer *bufp, struct re_registers *regs, unsigned num_regs, regoff_t
*starts, regoff_t *ends)
{
  if (num_regs)
    {
      bufp->regs_allocated = REGS_REALLOCATE;
      regs->num_regs = num_regs;
      regs->start = starts;
      regs->end = ends;
    }
  else
    {
      bufp->regs_allocated = REGS_UNALLOCATED;
      regs->num_regs = 0;
      regs->start = regs->end = (regoff_t *) 0;
    }
}
```

```
/* Searching routines.   */

/* Like re_search_2, below, but only one string is specified, and
     doesn't let you say where to stop matching. */

int
re_search (struct re_pattern_buffer *bufp, const char *string, int size, int startpos, int range, struct
re_registers *regs)
{
   return re_search_2 (bufp, NULL, 0, string, size, startpos, range,
                                  regs, size);
}


/* Using the compiled pattern in BUFP->buffer, first tries to match the
     virtual concatenation of STRING1 and STRING2, starting first at index
     STARTPOS, then at STARTPOS + 1, and so on.

     STRING1 and STRING2 have length SIZE1 and SIZE2, respectively.

     RANGE is how far to scan while trying to match.    RANGE = 0 means try
     only at STARTPOS; in general, the last start tried is STARTPOS +
     RANGE.

     In REGS, return the indices of the virtual concatenation of STRING1
     and STRING2 that matched the entire BUFP->buffer and its contained
     subexpressions.

     Do not consider matching one past the index STOP in the virtual
     concatenation of STRING1 and STRING2.

     We return either the position in the strings at which the match was
     found, -1 if no match, or -2 if error (such as failure
```

```
     stack overflow).    */

int
re_search_2 (struct re_pattern_buffer *bufp, const char *string1, int size1, const char *string2,
         int size2, int startpos, int range, struct re_registers *regs, int stop)
{
   int val;
   register char *fastmap = bufp->fastmap;
   register RE_TRANSLATE_TYPE translate = bufp->translate;
   int total_size = size1 + size2;
   int endpos = startpos + range;

   /* Check for out-of-range STARTPOS.    */
   if (startpos < 0 || startpos > total_size)
      return -1;

   /* Fix up RANGE if it might eventually take us outside
         the virtual concatenation of STRING1 and STRING2.
         Make sure we won't move STARTPOS below 0 or above TOTAL_SIZE.    */
   if (endpos < 0)
      range = 0 - startpos;
   else if (endpos > total_size)
      range = total_size - startpos;

   /* If the search isn't to be a backwards one, don't waste time in a
         search for a pattern that must be anchored.    */
   if (bufp->used > 0 && (re_opcode_t) bufp->buffer[0] == begbuf && range > 0)
      {
         if (startpos > 0)
            return -1;
         else
            range = 1;
      }
```

```
#ifdef emacs
   /* In a forward search for something that starts with ¥=.
        don't keep searching past point.    */
   if (bufp->used > 0 && (re_opcode_t) bufp->buffer[0] == at_dot && range > 0)
      {
         range = PT - startpos;
         if (range <= 0)
            return -1;
      }
#endif /* emacs */

   /* Update the fastmap now if not correct already.    */
   if (fastmap && !bufp->fastmap_accurate)
      if (re_compile_fastmap (bufp) == -2)
         return -2;

   /* Loop through the string, looking for a place to start matching.    */
   for (;;)
      {
         /* If a fastmap is supplied, skip quickly over characters that
              cannot be the start of a match.    If the pattern can match the
              null string, however, we don't need to skip characters; we want
              the first null string.    */
         if (fastmap && startpos < total_size && !bufp->can_be_null)
            {
               if (range > 0)    /* Searching forwards.    */
                  {
                     register const char *d;
                     register int lim = 0;
                     int irange = range;

                     if (startpos < size1 && startpos + range >= size1)
                        lim = range - (size1 - startpos);
```

```
            d = (startpos >= size1 ? string2 - size1 : string1) + startpos;

            /* Written out as an if-else to avoid testing `translate'
               inside the loop.  */
            if (translate)
              while (range > lim
                     && !fastmap[(unsigned char)
                                 translate[(unsigned char) *d++]])
                range--;
            else
              while (range > lim && !fastmap[(unsigned char) *d++])
                range--;

            startpos += irange - range;
          }
        else                                    /* Searching backwards.  */
          {
            register char c = (size1 == 0 || startpos >= size1
                               ? string2[startpos - size1]
                               : string1[startpos]);

            if (!fastmap[(unsigned char) TRANSLATE (c)])
              goto advance;
          }
      }

    /* If can't match the null string, and that's all we have left, fail.  */
    if (range >= 0 && startpos == total_size && fastmap
        && !bufp->can_be_null)
      return -1;

    val = re_match_2_internal (bufp, string1, size1, string2, size2,
                               startpos, regs, stop);
#ifndef REGEX_MALLOC
```

```
#ifdef C_ALLOCA
        alloca (0);
#endif
#endif

        if (val >= 0)
          return startpos;

        if (val == -2)
          return -2;

      advance:
        if (!range)
          break;
        else if (range > 0)
          {
            range--;
            startpos++;
          }
        else
          {
            range++;
            startpos--;
          }
      }
   return -1;
} /* re_search_2 */
```

```
/* This converts PTR, a pointer into one of the search strings `string1'
   and `string2' into an offset from the beginning of that string.  */
#define POINTER_TO_OFFSET(ptr)                          ¥
  (FIRST_STRING_P (ptr)                                 ¥
   ? ((regoff_t) ((ptr) - string1))             ¥
   : ((regoff_t) ((ptr) - string2 + size1)))

/* Macros for dealing with the split strings in re_match_2.  */

#define MATCHING_IN_FIRST_STRING  (dend == end_match_1)

/* Call before fetching a character with *d.  This switches over to
   string2 if necessary.  */
#define PREFETCH()                                                      ¥
  while (d == dend)                                                     ¥
    {                                                                   ¥
      /* End of string2 => fail.  */                                    ¥
      if (dend == end_match_2)                                          ¥
        goto fail;                                                      ¥
      /* End of string1 => advance to string2.  */                      ¥
      d = string2;                                                      ¥
      dend = end_match_2;                                               ¥
    }


/* Test if at very beginning or at very end of the virtual concatenation
   of `string1' and `string2'.  If only one string, it's `string2'.  */
#define AT_STRINGS_BEG(d) ((d) == (size1 ? string1 : string2) || !size2)
#define AT_STRINGS_END(d) ((d) == end2)


/* Test if D points to a character which is word-constituent.  We have
   two special cases to check for: if past the end of string1, look at
```

```
    the first character in string2; and if before the beginning of
    string2, look at the last character in string1.    */
#define WORDCHAR_P(d)                                                    ¥
  (SYNTAX ((d) == end1 ? *string2                                        ¥
                : (d) == string2 - 1 ? *(end1 - 1) : *(d))               ¥
    == Sword)

/* Disabled due to a compiler bug -- see comment at case wordbound */
#if 0
/* Test if the character before D and the one at D differ with respect
    to being word-constituent.    */
#define AT_WORD_BOUNDARY(d)                                              ¥
  (AT_STRINGS_BEG (d) || AT_STRINGS_END (d)                              ¥
    || WORDCHAR_P (d - 1) != WORDCHAR_P (d))
#endif

/* Free everything we malloc.    */
#ifdef MATCH_MAY_ALLOCATE
#define FREE_VAR(var) if (var) REGEX_FREE (var); var = NULL
#define FREE_VARIABLES()                                                 ¥
  do {                                                                   ¥
    REGEX_FREE_STACK (fail_stack.stack);                                 ¥
    FREE_VAR (regstart);                                                 ¥
    FREE_VAR (regend);                                                   ¥
    FREE_VAR (old_regstart);                                             ¥
    FREE_VAR (old_regend);                                               ¥
    FREE_VAR (best_regstart);                                            ¥
    FREE_VAR (best_regend);                                              ¥
    FREE_VAR (reg_info);                                                 ¥
    FREE_VAR (reg_dummy);                                                ¥
    FREE_VAR (reg_info_dummy);                                           ¥
  } while (0)
#else
#define FREE_VARIABLES() ((void)0) /* Do nothing!    But inhibit gcc warning.    */
```

```
#endif /* not MATCH_MAY_ALLOCATE */

/* These values must meet several constraints.  They must not be valid
   register values; since we have a limit of 255 registers (because
   we use only one byte in the pattern for the register number), we can
   use numbers larger than 255.  They must differ by 1, because of
   NUM_FAILURE_ITEMS above.  And the value for the lowest register must
   be larger than the value for the highest register, so we do not try
   to actually save any registers when none are active.  */
#define NO_HIGHEST_ACTIVE_REG (1 << BYTEWIDTH)
#define NO_LOWEST_ACTIVE_REG (NO_HIGHEST_ACTIVE_REG + 1)
```

```
/* Matching routines.    */

#ifndef emacs     /* Emacs never uses this.    */
/* re_match is like re_match_2 except it takes only a single string.    */

int
re_match (struct re_pattern_buffer *bufp,
        const char *string,
        int size, int pos,
        struct re_registers *regs)
{
   int result = re_match_2_internal (bufp, NULL, 0, string, size,
                                                 pos, regs, size);
#ifndef REGEX_MALLOC
#ifdef C_ALLOCA
   alloca (0);
#endif
#endif
   return result;
}
#endif /* not emacs */

static boolean group_match_null_string_p(unsigned char **p,
                                                    unsigned char *end,
                                                register_info_type *reg_info);
static boolean alt_match_null_string_p(unsigned char *p,
                                                   unsigned char *end,
                                                register_info_type *reg_info);
static boolean common_op_match_null_string_p(unsigned char **p,
                                                        unsigned char *end,
                                                register_info_type *reg_info);
static int bcmp_translate(const char *s1, const char *s2,
                                                 int len, char *translate);
```

```
/* re_match_2 matches the compiled pattern in BUFP against the
     the (virtual) concatenation of STRING1 and STRING2 (of length SIZE1
     and SIZE2, respectively).    We start matching at POS, and stop
     matching at STOP.

     If REGS is non-null and the `no_sub' field of BUFP is nonzero, we
     store offsets for the substring each group matched in REGS.    See the
     documentation for exactly how many groups we fill.

     We return -1 if no match, -2 if an internal error (such as the
     failure stack overflowing).    Otherwise, we return the length of the
     matched substring.    */

int
re_match_2 (struct re_pattern_buffer *bufp,
        const char *string1, int size1, const char *string2,
        int size2, int pos, struct re_registers *regs, int stop)
{
   int result = re_match_2_internal (bufp, string1, size1, string2, size2,
                                                 pos, regs, stop);
#ifndef REGEX_MALLOC
#ifdef C_ALLOCA
   alloca (0);
#endif
#endif
   return result;
}

/* This is a separate function so that we can force an alloca cleanup
     afterwards.    */
static int
re_match_2_internal (struct re_pattern_buffer *bufp,
        const char *string1, int size1, const char *string2,
```

```
    int size2, int pos, struct re_registers *regs, int stop)
{
  /* General temporaries.  */
  int mcnt;
  unsigned char *p1;

  /* Just past the end of the corresponding string.  */
  const char *end1, *end2;

  /* Pointers into string1 and string2, just past the last characters in
     each to consider matching.  */
  const char *end_match_1, *end_match_2;

  /* Where we are in the data, and the end of the current string.  */
  const char *d, *dend;

  /* Where we are in the pattern, and the end of the pattern.  */
  unsigned char *p = bufp->buffer;
  register unsigned char *pend = p + bufp->used;

  /* Mark the opcode just after a start_memory, so we can test for an
     empty subpattern when we get to the stop_memory.  */
  unsigned char *just_past_start_mem = 0;

  /* We use this to map every character in the string.  */
  RE_TRANSLATE_TYPE translate = bufp->translate;

  /* Failure point stack.  Each place that can handle a failure further
     down the line pushes a failure point on this stack.  It consists of
     restart, regend, and reg_info for all registers corresponding to
     the subexpressions we're currently inside, plus the number of such
     registers, and, finally, two char *'s.  The first char * is where
     to resume scanning the pattern; the second one is where to resume
     scanning the strings.  If the latter is zero, the failure point is
```

```
        a ``dummy''; if a failure happens and the failure point is a dummy,
        it gets discarded and the next next one is tried.  */
#ifdef MATCH_MAY_ALLOCATE /* otherwise, this is global.  */
  fail_stack_type fail_stack;
#endif
#ifdef DEBUG
  static unsigned failure_id = 0;
  unsigned nfailure_points_pushed = 0, nfailure_points_popped = 0;
#endif

#ifdef REL_ALLOC
  /* This holds the pointer to the failure stack, when
        it is allocated relocatably.  */
  fail_stack_elt_t *failure_stack_ptr;
#endif

  /* We fill all the registers internally, independent of what we
        return, for use in backreferences.  The number here includes
        an element for register zero.  */
  size_t num_regs = bufp->re_nsub + 1;

  /* The currently active registers.  */
  active_reg_t lowest_active_reg = NO_LOWEST_ACTIVE_REG;
  active_reg_t highest_active_reg = NO_HIGHEST_ACTIVE_REG;

  /* Information on the contents of registers. These are pointers into
        the input strings; they record just what was matched (on this
        attempt) by a subexpression part of the pattern, that is, the
        regnum-th regstart pointer points to where in the pattern we began
        matching and the regnum-th regend points to right after where we
        stopped matching the regnum-th subexpression.  (The zeroth register
        keeps track of what the whole pattern matches.)  */
#ifdef MATCH_MAY_ALLOCATE /* otherwise, these are global.  */
  const char **regstart, **regend;
```

```
#endif

    /* If a group that's operated upon by a repetition operator fails to
       match anything, then the register for its start will need to be
       restored because it will have been set to wherever in the string we
       are when we last see its open-group operator.  Similarly for a
       register's end.  */
#ifdef MATCH_MAY_ALLOCATE /* otherwise, these are global.  */
    const char **old_regstart, **old_regend;
#endif

    /* The is_active field of reg_info helps us keep track of which (possibly
       nested) subexpressions we are currently in. The matched_something
       field of reg_info[reg_num] helps us tell whether or not we have
       matched any of the pattern so far this time through the reg_num-th
       subexpression.  These two fields get reset each time through any
       loop their register is in.  */
#ifdef MATCH_MAY_ALLOCATE /* otherwise, this is global.  */
    register_info_type *reg_info;
#endif

    /* The following record the register info as found in the above
       variables when we find a match better than any we've seen before.
       This happens as we backtrack through the failure points, which in
       turn happens only if we have not yet matched the entire string. */
    unsigned best_regs_set = false;
#ifdef MATCH_MAY_ALLOCATE /* otherwise, these are global.  */
    const char **best_regstart, **best_regend;
#endif

    /* Logically, this is `best_regend[0]'.  But we don't want to have to
       allocate space for that if we're not allocating space for anything
       else (see below).  Also, we never need info about register 0 for
       any of the other register vectors, and it seems rather a kludge to
```

```
      treat `best_regend' differently than the rest.  So we keep track of
      the end of the best match so far in a separate variable.  We
      initialize this to NULL so that when we backtrack the first time
      and need to test it, it's not garbage.  */
  const char *match_end = NULL;

  /* This helps SET_REGS_MATCHED avoid doing redundant work.  */
  int set_regs_matched_done = 0;

  /* Used when we pop values we don't care about.  */
#ifdef MATCH_MAY_ALLOCATE /* otherwise, these are global.  */
  const char **reg_dummy;
  register_info_type *reg_info_dummy;
#endif

#ifdef DEBUG
  /* Counts the total number of registers pushed.  */
  unsigned num_regs_pushed = 0;
#endif

  DEBUG_PRINT1 ("¥n¥nEntering re_match_2.¥n");

  INIT_FAIL_STACK ();

#ifdef MATCH_MAY_ALLOCATE
  /* Do not bother to initialize all the register variables if there are
      no groups in the pattern, as it takes a fair amount of time.  If
      there are groups, we include space for register 0 (the whole
      pattern), even though we never use it, since it simplifies the
      array indexing.  We should fix this.  */
  if (bufp->re_nsub)
    {
      regstart = REGEX_TALLOC (num_regs, const char *);
      regend = REGEX_TALLOC (num_regs, const char *);
```

```
        old_regstart = REGEX_TALLOC (num_regs, const char *);
        old_regend = REGEX_TALLOC (num_regs, const char *);
        best_regstart = REGEX_TALLOC (num_regs, const char *);
        best_regend = REGEX_TALLOC (num_regs, const char *);
        reg_info = REGEX_TALLOC (num_regs, register_info_type);
        reg_dummy = REGEX_TALLOC (num_regs, const char *);
        reg_info_dummy = REGEX_TALLOC (num_regs, register_info_type);

        if (!(regstart && regend && old_regstart && old_regend && reg_info
                && best_regstart && best_regend && reg_dummy && reg_info_dummy))
          {
            FREE_VARIABLES ();
            return -2;
          }
      }
    else
      {
        /* We must initialize all our variables to NULL, so that
           `FREE_VARIABLES' doesn't try to free them.  */
        regstart = regend = old_regstart = old_regend = best_regstart
          = best_regend = reg_dummy = NULL;
        reg_info = reg_info_dummy = (register_info_type *) NULL;
      }
#endif /* MATCH_MAY_ALLOCATE */

    /* The starting position is bogus.  */
    if (pos < 0 || pos > size1 + size2)
      {
        FREE_VARIABLES ();
        return -1;
      }

    /* Initialize subexpression text positions to -1 to mark ones that no
       start_memory/stop_memory has been seen for. Also initialize the
```

```
     register information struct.    */
for (mcnt = 1; (unsigned) mcnt < num_regs; mcnt++)
   {
      regstart[mcnt] = regend[mcnt]
        = old_regstart[mcnt] = old_regend[mcnt] = REG_UNSET_VALUE;

      REG_MATCH_NULL_STRING_P (reg_info[mcnt]) = MATCH_NULL_UNSET_VALUE;
      IS_ACTIVE (reg_info[mcnt]) = 0;
      MATCHED_SOMETHING (reg_info[mcnt]) = 0;
      EVER_MATCHED_SOMETHING (reg_info[mcnt]) = 0;
   }

/* We move `string1' into `string2' if the latter's empty -- but not if
     `string1' is null.    */
if (size2 == 0 && string1 != NULL)
   {
      string2 = string1;
      size2 = size1;
      string1 = 0;
      size1 = 0;
   }
end1 = string1 + size1;
end2 = string2 + size2;

/* Compute where to stop matching, within the two strings.    */
if (stop <= size1)
   {
      end_match_1 = string1 + stop;
      end_match_2 = string2;
   }
else
   {
      end_match_1 = end1;
      end_match_2 = string2 + stop - size1;
```

```
    }

  /* `p' scans through the pattern as `d' scans through the data.
     `dend' is the end of the input string that `d' points within.  `d'
     is advanced into the following input string whenever necessary, but
     this happens before fetching; therefore, at the beginning of the
     loop, `d' can be pointing at the end of a string, but it cannot
     equal `string2'.  */
  if (size1 > 0 && pos <= size1)
    {
      d = string1 + pos;
      dend = end_match_1;
    }
  else
    {
      d = string2 + pos - size1;
      dend = end_match_2;
    }

  DEBUG_PRINT1 ("The compiled pattern is:¥n");
  DEBUG_PRINT_COMPILED_PATTERN (bufp, p, pend);
  DEBUG_PRINT1 ("The string to match is: `");
  DEBUG_PRINT_DOUBLE_STRING (d, string1, size1, string2, size2);
  DEBUG_PRINT1 ("'¥n");

  /* This loops over pattern commands.  It exits by returning from the
     function if the match is complete, or it drops through if the match
     fails at this starting point in the input data.  */
  for (;;)
    {
#ifdef _LIBC
      DEBUG_PRINT2 ("¥n%p: ", p);
#else
      DEBUG_PRINT2 ("¥n0x%x: ", p);
```

```
#endif
            boolean same_str_p;

        if (p == pend)
          { /* End of pattern means we might have succeeded.    */
            DEBUG_PRINT1 ("end of pattern ... ");

            /* If we haven't matched the entire string, and we want the
                longest match, try backtracking.    */
            if (d != end_match_2)
              {
                /* 1 if this match ends in the same string (string1 or string2)
                    as the best previous match.    */
                same_str_p = (FIRST_STRING_P (match_end)
                                        == MATCHING_IN_FIRST_STRING);
                /* 1 if this match is the best seen so far.    */
                boolean best_match_p;

                /* AIX compiler got confused when this was combined
                    with the previous declaration.    */
                if (same_str_p)
                  best_match_p = d > match_end;
                else
                  best_match_p = !MATCHING_IN_FIRST_STRING;

                DEBUG_PRINT1 ("backtracking.¥n");

                if (!FAIL_STACK_EMPTY ())
                  { /* More failure points to try.    */

                    /* If exceeds best match so far, save it.    */
                    if (!best_regs_set || best_match_p)
                      {
                        best_regs_set = true;
```

```
            match_end = d;

            DEBUG_PRINT1 ("¥nSAVING match as best so far.¥n");

            for (mcnt = 1; (unsigned) mcnt < num_regs; mcnt++)
              {
                best_regstart[mcnt] = regstart[mcnt];
                best_regend[mcnt] = regend[mcnt];
              }
          }
        goto fail;
      }

  /* If no failure points, don't restore garbage.    And if
     last match is real best match, don't restore second
     best one. */
  else if (best_regs_set && !best_match_p)
    {
    restore_best_regs:
      /* Restore best match.    It may happen that `dend ==
           end_match_1' while the restored d is in string2.
           For example, the pattern `x.*y.*z' against the
           strings `x-' and `y-z-', if the two strings are
           not consecutive in memory.    */
      DEBUG_PRINT1 ("Restoring best registers.¥n");

      d = match_end;
      dend = ((d >= string1 && d <= end1)
                ? end_match_1 : end_match_2);

      for (mcnt = 1; (unsigned) mcnt < num_regs; mcnt++)
        {
          regstart[mcnt] = best_regstart[mcnt];
          regend[mcnt] = best_regend[mcnt];
```

```
                }
            }
        } /* d != end_match_2 */

succeed_label:
    DEBUG_PRINT1 ("Accepting match.¥n");

    /* If caller wants register contents data back, do it.  */
    if (regs && !bufp->no_sub)
      {
        /* Have the register data arrays been allocated?  */
        if (bufp->regs_allocated == REGS_UNALLOCATED)
          { /* No.  So allocate them with malloc.  We need one
               extra element beyond `num_regs' for the `-1' marker
               GNU code uses.  */
            regs->num_regs = MAX (RE_NREGS, num_regs + 1);
            regs->start = TALLOC (regs->num_regs, regoff_t);
            regs->end = TALLOC (regs->num_regs, regoff_t);
            if (regs->start == NULL || regs->end == NULL)
              {
                FREE_VARIABLES ();
                return -2;
              }
            bufp->regs_allocated = REGS_REALLOCATE;
          }
        else if (bufp->regs_allocated == REGS_REALLOCATE)
          { /* Yes.  If we need more elements than were already
               allocated, reallocate them.  If we need fewer, just
               leave it alone.  */
            if (regs->num_regs < num_regs + 1)
              {
                regs->num_regs = num_regs + 1;
                RETALLOC (regs->start, regs->num_regs, regoff_t);
                RETALLOC (regs->end, regs->num_regs, regoff_t);
```

```
            if (regs->start == NULL || regs->end == NULL)
              {
                FREE_VARIABLES ();
                return -2;
              }
          }
      }
    else
      {
        /* These braces fend off a "empty body in an else-statement"
           warning under GCC when assert expands to nothing.  */
        assert (bufp->regs_allocated == REGS_FIXED);
      }

    /* Convert the pointer data in `regstart' and `regend' to
       indices.  Register zero has to be set differently,
       since we haven't kept track of any info for it.  */
    if (regs->num_regs > 0)
      {
        regs->start[0] = pos;
        regs->end[0] = (MATCHING_IN_FIRST_STRING
                        ? ((regoff_t) (d - string1))
                        : ((regoff_t) (d - string2 + size1)));
      }

    /* Go through the first `min (num_regs, regs->num_regs)'
       registers, since that is all we initialized.  */
    for (mcnt = 1; (unsigned) mcnt < MIN (num_regs, regs->num_regs);
         mcnt++)
      {
        if (REG_UNSET (regstart[mcnt]) || REG_UNSET (regend[mcnt]))
          regs->start[mcnt] = regs->end[mcnt] = -1;
        else
          {
```

```
                    regs->start[mcnt]
                      = (regoff_t) POINTER_TO_OFFSET (regstart[mcnt]);
                    regs->end[mcnt]
                      = (regoff_t) POINTER_TO_OFFSET (regend[mcnt]);
                  }
              }

            /* If the regs structure we return has more elements than
                were in the pattern, set the extra elements to -1.    If
                we (re)allocated the registers, this is the case,
                because we always allocate enough to have at least one
                -1 at the end.    */
            for (mcnt = num_regs; (unsigned) mcnt < regs->num_regs; mcnt++)
              regs->start[mcnt] = regs->end[mcnt] = -1;
          } /* regs && !bufp->no_sub */

        DEBUG_PRINT4 ("%u failure points pushed, %u popped (%u remain).¥n",
                      nfailure_points_pushed, nfailure_points_popped,
                      nfailure_points_pushed - nfailure_points_popped);
        DEBUG_PRINT2 ("%u registers pushed.¥n", num_regs_pushed);

        mcnt = d - pos - (MATCHING_IN_FIRST_STRING
                          ? string1
                          : string2 - size1);

        DEBUG_PRINT2 ("Returning %d from re_match_2.¥n", mcnt);

        FREE_VARIABLES ();
        return mcnt;
      }

    /* Otherwise match next pattern command.    */
    switch (SWITCH_ENUM_CAST ((re_opcode_t) *p++))
      {
```

```
/* Ignore these.    Used to ignore the n of succeed_n's which
     currently have n == 0.    */
case no_op:
   DEBUG_PRINT1 ("EXECUTING no_op.¥n");
   break;

case succeed:
   DEBUG_PRINT1 ("EXECUTING succeed.¥n");
   goto succeed_label;

/* Match the next n pattern characters exactly.    The following
     byte in the pattern defines n, and the n bytes after that
     are the characters to match.    */
case exactn:
   mcnt = *p++;
   DEBUG_PRINT2 ("EXECUTING exactn %d.¥n", mcnt);

   /* This is written out as an if-else so we don't waste time
        testing `translate' inside the loop.    */
   if (translate)
     {
        do
          {
             PREFETCH ();
             if ((unsigned char) translate[(unsigned char) *d++]
                  != (unsigned char) *p++)
               goto fail;
          }
        while (--mcnt);
     }
   else
     {
        do
          {
```

```
              PREFETCH ();
              if (*d++ != (char) *p++) goto fail;
            }
          while (--mcnt);
        }
      SET_REGS_MATCHED ();
      break;


/* Match any character except possibly a newline or a null.  */
case anychar:
      DEBUG_PRINT1 ("EXECUTING anychar.¥n");

      PREFETCH ();

      if ((!(bufp->syntax & RE_DOT_NEWLINE) && TRANSLATE (*d) == '¥n')
          || (bufp->syntax & RE_DOT_NOT_NULL && TRANSLATE (*d) == '¥000'))
        goto fail;

      SET_REGS_MATCHED ();
      DEBUG_PRINT2 ("  Matched `%d'.¥n", *d);
      d++;
      break;


case charset:
case charset_not:
      {
        register unsigned char c;
        boolean not = (re_opcode_t) *(p - 1) == charset_not;

        DEBUG_PRINT2 ("EXECUTING charset%s.¥n", not ? "_not" : "");

        PREFETCH ();
```

```
      c = TRANSLATE (*d); /* The character to match.    */

      /* Cast to `unsigned' instead of `unsigned char' in case the
         bit list is a full 32 bytes long.    */
      if (c < (unsigned) (*p * BYTEWIDTH)
          && p[1 + c / BYTEWIDTH] & (1 << (c % BYTEWIDTH)))
        not = !not;

      p += 1 + *p;

      if (!not) goto fail;

      SET_REGS_MATCHED ();
      d++;
      break;
    }


/* The beginning of a group is represented by start_memory.
     The arguments are the register number in the next byte, and the
     number of groups inner to this one in the next.    The text
     matched within the group is recorded (in the internal
     registers data structure) under the register number.    */
case start_memory:
    DEBUG_PRINT3 ("EXECUTING start_memory %d (%d):¥n", *p, p[1]);

    /* Find out if this group can match the empty string.    */
    p1 = p;                         /* To send to group_match_null_string_p.    */

    if (REG_MATCH_NULL_STRING_P (reg_info[*p]) == MATCH_NULL_UNSET_VALUE)
      REG_MATCH_NULL_STRING_P (reg_info[*p])
        = group_match_null_string_p (&p1, pend, reg_info);

    /* Save the position in the string where we were the last time
```

```
    we were at this open-group operator in case the group is
    operated upon by a repetition operator, e.g., with `(a*)*b'
    against `ab'; then we want to ignore where we are now in
    the string in case this attempt to match fails.    */
old_regstart[*p] = REG_MATCH_NULL_STRING_P (reg_info[*p])
                         ? REG_UNSET (regstart[*p]) ? d : regstart[*p]
                         : regstart[*p];
DEBUG_PRINT2 ("    old_regstart: %d¥n",
                  POINTER_TO_OFFSET (old_regstart[*p]));

regstart[*p] = d;
DEBUG_PRINT2 ("    regstart: %d¥n", POINTER_TO_OFFSET (regstart[*p]));

IS_ACTIVE (reg_info[*p]) = 1;
MATCHED_SOMETHING (reg_info[*p]) = 0;

/* Clear this whenever we change the register activity status.    */
set_regs_matched_done = 0;

/* This is the new highest active register.    */
highest_active_reg = *p;

/* If nothing was active before, this is the new lowest active
    register.    */
if (lowest_active_reg == NO_LOWEST_ACTIVE_REG)
   lowest_active_reg = *p;

/* Move past the register number and inner group count.    */
p += 2;
just_past_start_mem = p;

break;
```

```
/* The stop_memory opcode represents the end of a group.  Its
     arguments are the same as start_memory's: the register
     number, and the number of inner groups.  */
case stop_memory:
  DEBUG_PRINT3 ("EXECUTING stop_memory %d (%d):¥n", *p, p[1]);

  /* We need to save the string position the last time we were at
       this close-group operator in case the group is operated
       upon by a repetition operator, e.g., with `((a*)*(b*)*)*'
       against `aba'; then we want to ignore where we are now in
       the string in case this attempt to match fails.  */
  old_regend[*p] = REG_MATCH_NULL_STRING_P (reg_info[*p])
                           ? REG_UNSET (regend[*p]) ? d : regend[*p]
                           : regend[*p];
  DEBUG_PRINT2 ("          old_regend: %d¥n",
                        POINTER_TO_OFFSET (old_regend[*p]));

  regend[*p] = d;
  DEBUG_PRINT2 ("          regend: %d¥n", POINTER_TO_OFFSET (regend[*p]));

  /* This register isn't active anymore.  */
  IS_ACTIVE (reg_info[*p]) = 0;

  /* Clear this whenever we change the register activity status.  */
  set_regs_matched_done = 0;

  /* If this was the only register active, nothing is active
       anymore.  */
  if (lowest_active_reg == highest_active_reg)
    {
      lowest_active_reg = NO_LOWEST_ACTIVE_REG;
      highest_active_reg = NO_HIGHEST_ACTIVE_REG;
    }
  else
```

```
  { /* We must scan for the new highest active register, since
       it isn't necessarily one less than now: consider
       (a(b)c(d(e)f)g).  When group 3 ends, after the f), the
       new highest active register is 1.  */
    unsigned char r = *p - 1;
    while (r > 0 && !IS_ACTIVE (reg_info[r]))
      r--;

    /* If we end up at register zero, that means that we saved
       the registers as the result of an `on_failure_jump', not
       a `start_memory', and we jumped to past the innermost
       `stop_memory'.  For example, in ((.)*) we save
       registers 1 and 2 as a result of the *, but when we pop
       back to the second ), we are at the stop_memory 1.
       Thus, nothing is active.  */
    if (r == 0)
      {
        lowest_active_reg = NO_LOWEST_ACTIVE_REG;
        highest_active_reg = NO_HIGHEST_ACTIVE_REG;
      }
    else
      highest_active_reg = r;
  }

/* If just failed to match something this time around with a
    group that's operated on by a repetition operator, try to
    force exit from the ``loop'', and restore the register
    information for this group that we had before trying this
    last match.  */
if ((!MATCHED_SOMETHING (reg_info[*p])
       || just_past_start_mem == p - 1)
    && (p + 2) < pend)
  {
    boolean is_a_jump_n = false;
```

```
p1 = p + 2;
mcnt = 0;
switch ((re_opcode_t) *p1++)
  {
    case jump_n:
      is_a_jump_n = true;
    case pop_failure_jump:
    case maybe_pop_jump:
    case jump:
    case dummy_failure_jump:
      EXTRACT_NUMBER_AND_INCR (mcnt, p1);
      if (is_a_jump_n)
        p1 += 2;
      break;

    default:
      /* do nothing */ ;
  }
p1 += mcnt;

/* If the next operation is a jump backwards in the pattern
    to an on_failure_jump right before the start_memory
    corresponding to this stop_memory, exit from the loop
    by forcing a failure after pushing on the stack the
    on_failure_jump's jump in the pattern, and d.  */
if (mcnt < 0 && (re_opcode_t) *p1 == on_failure_jump
    && (re_opcode_t) p1[3] == start_memory && p1[4] == *p)
  {
    /* If this group ever matched anything, then restore
        what its registers were before trying this last
        failed match, e.g., with `(a*)*b' against `ab' for
        regstart[1], and, e.g., with `((a*)*(b*)*)*'
        against `aba' for regend[3].
```

```
            Also restore the registers for inner groups for,
            e.g., `((a*)(b*))*' against `aba' (register 3 would
            otherwise get trashed).    */

          if (EVER_MATCHED_SOMETHING (reg_info[*p]))
            {
              unsigned r;

              EVER_MATCHED_SOMETHING (reg_info[*p]) = 0;

              /* Restore this and inner groups' (if any) registers.    */
              for (r = *p; r < (unsigned) *p + (unsigned) *(p + 1);
                   r++)
                {
                  regstart[r] = old_regstart[r];

                  /* xx why this test?    */
                  if (old_regend[r] >= regstart[r])
                    regend[r] = old_regend[r];
                }
            }
          p1++;
          EXTRACT_NUMBER_AND_INCR (mcnt, p1);
          PUSH_FAILURE_POINT (p1 + mcnt, d, -2);

          goto fail;
        }
    }

/* Move past the register number and the inner group count.    */
p += 2;
break;
```

```
/* ¥<digit> has been turned into a `duplicate' command which is
     followed by the numeric value of <digit> as the register number.    */
case duplicate:
   {
      register const char *d2, *dend2;
      int regno = *p++;     /* Get which register to match against.    */
      DEBUG_PRINT2 ("EXECUTING duplicate %d.¥n", regno);

      /* Can't back reference a group which we've never matched.    */
      if (REG_UNSET (regstart[regno]) || REG_UNSET (regend[regno]))
         goto fail;

      /* Where in input to try to start matching.    */
      d2 = regstart[regno];

      /* Where to stop matching; if both the place to start and
           the place to stop matching are in the same string, then
           set to the place to stop, otherwise, for now have to use
           the end of the first string.    */

      dend2 = ((FIRST_STRING_P (regstart[regno])
                  == FIRST_STRING_P (regend[regno]))
                ? regend[regno] : end_match_1);
      for (;;)
        {
          /* If necessary, advance to next segment in register
               contents.    */
          while (d2 == dend2)
            {
              if (dend2 == end_match_2) break;
              if (dend2 == regend[regno]) break;

              /* End of string1 => advance to string2. */
```

```
              d2 = string2;
              dend2 = regend[regno];
            }
          /* At end of register contents => success */
          if (d2 == dend2) break;

          /* If necessary, advance to next segment in data.    */
          PREFETCH ();

          /* How many characters left in this segment to match.    */
          mcnt = dend - d;

          /* Want how many consecutive characters we can match in
               one shot, so, if necessary, adjust the count.    */
          if (mcnt > dend2 - d2)
            mcnt = dend2 - d2;

          /* Compare that many; failure if mismatch, else move
               past them.    */
          if (translate
                ? bcmp_translate (d, d2, mcnt, translate)
                : bcmp (d, d2, mcnt))
            goto fail;
          d += mcnt, d2 += mcnt;

          /* Do this because we've match some characters.    */
          SET_REGS_MATCHED ();
        }
    }
    break;


/* begline matches the empty string at the beginning of the string
     (unless `not_bol' is set in `bufp'), and, if
```

```
     `newline_anchor' is set, after newlines.    */
case begline:
   DEBUG_PRINT1 ("EXECUTING begline.¥n");

   if (AT_STRINGS_BEG (d))
     {
       if (!bufp->not_bol) break;
     }
   else if (d[-1] == '¥n' && bufp->newline_anchor)
     {
       break;
     }
   /* In all other cases, we fail.    */
   goto fail;


/* endline is the dual of begline.    */
case endline:
   DEBUG_PRINT1 ("EXECUTING endline.¥n");

   if (AT_STRINGS_END (d))
     {
       if (!bufp->not_eol) break;
     }

   /* We have to ``prefetch'' the next character.    */
   else if ((d == end1 ? *string2 : *d) == '¥n'
            && bufp->newline_anchor)
     {
       break;
     }
   goto fail;
```

```
/* Match at the very beginning of the data.  */
case begbuf:
  DEBUG_PRINT1 ("EXECUTING begbuf.¥n");
  if (AT_STRINGS_BEG (d))
    break;
  goto fail;


/* Match at the very end of the data.  */
case endbuf:
  DEBUG_PRINT1 ("EXECUTING endbuf.¥n");
  if (AT_STRINGS_END (d))
    break;
  goto fail;


/* on_failure_keep_string_jump is used to optimize `.*¥n'.  It
     pushes NULL as the value for the string on the stack.  Then
     `pop_failure_point' will keep the current value for the
     string, instead of restoring it.  To see why, consider
     matching `foo¥nbar' against `.*¥n'.  The .* matches the foo;
     then the . fails against the ¥n.  But the next thing we want
     to do is match the ¥n against the ¥n; if we restored the
     string value, we would be back at the foo.

     Because this is used only in specific cases, we don't need to
     check all the things that `on_failure_jump' does, to make
     sure the right things get saved on the stack.  Hence we don't
     share its code.  The only reason to push anything on the
     stack at all is that otherwise we would have to change
     `anychar's code to do something besides goto fail in this
     case; that seems worse than this.  */
case on_failure_keep_string_jump:
  DEBUG_PRINT1 ("EXECUTING on_failure_keep_string_jump");
```

```
            EXTRACT_NUMBER_AND_INCR (mcnt, p);
#ifdef _LIBC
            DEBUG_PRINT3 (" %d (to %p):¥n", mcnt, p + mcnt);
#else
            DEBUG_PRINT3 (" %d (to 0x%x):¥n", mcnt, p + mcnt);
#endif

            PUSH_FAILURE_POINT (p + mcnt, NULL, -2);
            break;


          /* Uses of on_failure_jump:

             Each alternative starts with an on_failure_jump that points
             to the beginning of the next alternative.  Each alternative
             except the last ends with a jump that in effect jumps past
             the rest of the alternatives.  (They really jump to the
             ending jump of the following alternative, because tensioning
             these jumps is a hassle.)

             Repeats start with an on_failure_jump that points past both
             the repetition text and either the following jump or
             pop_failure_jump back to this on_failure_jump.  */
          case on_failure_jump:
          on_failure:
            DEBUG_PRINT1 ("EXECUTING on_failure_jump");

            EXTRACT_NUMBER_AND_INCR (mcnt, p);
#ifdef _LIBC
            DEBUG_PRINT3 (" %d (to %p)", mcnt, p + mcnt);
#else
            DEBUG_PRINT3 (" %d (to 0x%x)", mcnt, p + mcnt);
#endif
```

```
/* If this on_failure_jump comes right before a group (i.e.,
     the original * applied to a group), save the information
     for that group and all inner ones, so that if we fail back
     to this point, the group's information will be correct.
     For example, in ¥(a*¥)*¥1, we need the preceding group,
     and in ¥(zz¥(a*¥)b*¥)¥2, we need the inner group.    */

/* We can't use `p' to check ahead because we push
     a failure point to `p + mcnt' after we do this.    */
p1 = p;

/* We need to skip no_op's before we look for the
     start_memory in case this on_failure_jump is happening as
     the result of a completed succeed_n, as in ¥(a¥)¥{1,3¥}b¥1
     against aba.    */
while (p1 < pend && (re_opcode_t) *p1 == no_op)
   p1++;

if (p1 < pend && (re_opcode_t) *p1 == start_memory)
   {
      /* We have a new highest active register now.    This will
           get reset at the start_memory we are about to get to,
           but we will have saved all the registers relevant to
           this repetition op, as described above.    */
      highest_active_reg = *(p1 + 1) + *(p1 + 2);
      if (lowest_active_reg == NO_LOWEST_ACTIVE_REG)
         lowest_active_reg = *(p1 + 1);
   }

DEBUG_PRINT1 (":¥n");
PUSH_FAILURE_POINT (p + mcnt, d, -2);
break;
```

```
/* A smart repeat ends with `maybe_pop_jump'.
     We change it to either `pop_failure_jump' or `jump'.    */
case maybe_pop_jump:
   EXTRACT_NUMBER_AND_INCR (mcnt, p);
   DEBUG_PRINT2 ("EXECUTING maybe_pop_jump %d.¥n", mcnt);
   {
      register unsigned char *p2 = p;

      /* Compare the beginning of the repeat with what in the
           pattern follows its end. If we can establish that there
           is nothing that they would both match, i.e., that we
           would have to backtrack because of (as in, e.g., `a*a')
           then we can change to pop_failure_jump, because we'll
           never have to backtrack.

           This is not true in the case of alternatives: in
           `(a|ab)*' we do need to backtrack to the `ab' alternative
           (e.g., if the string was `ab').    But instead of trying to
           detect that here, the alternative has put on a dummy
           failure point which is what we will end up popping.    */

      /* Skip over open/close-group commands.
           If what follows this loop is a ...+ construct,
           look at what begins its body, since we will have to
           match at least one of that.    */
      while (1)
        {
          if (p2 + 2 < pend
                && ((re_opcode_t) *p2 == stop_memory
                     || (re_opcode_t) *p2 == start_memory))
            p2 += 3;
          else if (p2 + 6 < pend
                       && (re_opcode_t) *p2 == dummy_failure_jump)
```

```
          p2 += 6;
        else
          break;
      }

p1 = p + mcnt;
/* p1[0] ... p1[2] are the `on_failure_jump' corresponding
     to the `maybe_finalize_jump' of this case.    Examine what
     follows.    */

/* If we're at the end of the pattern, we can change.    */
if (p2 == pend)
    {
      /* Consider what happens when matching ":¥(.*¥)"
           against ":/".    I don't really understand this code
           yet.    */
      p[-3] = (unsigned char) pop_failure_jump;
      DEBUG_PRINT1
        ("    End of pattern: change to `pop_failure_jump'.¥n");
    }

else if ((re_opcode_t) *p2 == exactn
              || (bufp->newline_anchor && (re_opcode_t) *p2 == endline))
    {
      register unsigned char c
        = *p2 == (unsigned char) endline ? '¥n' : p2[2];

      if ((re_opcode_t) p1[3] == exactn && p1[5] != c)
        {
          p[-3] = (unsigned char) pop_failure_jump;
          DEBUG_PRINT3 ("    %c != %c => pop_failure_jump.¥n",
                                c, p1[5]);
        }
```

```
                else if ((re_opcode_t) p1[3] == charset
                         || (re_opcode_t) p1[3] == charset_not)
                  {
                    int not = (re_opcode_t) p1[3] == charset_not;

                    if (c < (unsigned char) (p1[4] * BYTEWIDTH)
                        && p1[5 + c / BYTEWIDTH] & (1 << (c % BYTEWIDTH)))
                      not = !not;

                    /* `not' is equal to 1 if c would match, which means
                        that we can't change to pop_failure_jump.    */
                    if (!not)
                      {
                        p[-3] = (unsigned char) pop_failure_jump;
                        DEBUG_PRINT1 ("    No match => pop_failure_jump.¥n");
                      }
                  }
              }
            else if ((re_opcode_t) *p2 == charset)
              {
#ifdef DEBUG
                register unsigned char c
                  = *p2 == (unsigned char) endline ? '¥n' : p2[2];
#endif

#if 0
                if ((re_opcode_t) p1[3] == exactn
                    && ! ((int) p2[1] * BYTEWIDTH > (int) p1[5]
                          && (p2[2 + p1[5] / BYTEWIDTH]
                              & (1 << (p1[5] % BYTEWIDTH)))))
#else
                if ((re_opcode_t) p1[3] == exactn
                    && ! ((int) p2[1] * BYTEWIDTH > (int) p1[4]
                          && (p2[2 + p1[4] / BYTEWIDTH]
```

```
                                & (1 << (p1[4] % BYTEWIDTH)))))
#endif
                    {
                      p[-3] = (unsigned char) pop_failure_jump;
                      DEBUG_PRINT3 ("    %c != %c => pop_failure_jump.¥n",
                                    c, p1[5]);
                    }

                  else if ((re_opcode_t) p1[3] == charset_not)
                    {
                      int idx;
                      /* We win if the charset_not inside the loop
                         lists every character listed in the charset after.  */
                      for (idx = 0; idx < (int) p2[1]; idx++)
                        if (! (p2[2 + idx] == 0
                               || (idx < (int) p1[4]
                                   && ((p2[2 + idx] & ~ p1[5 + idx]) == 0))))
                          break;

                      if (idx == p2[1])
                        {
                          p[-3] = (unsigned char) pop_failure_jump;
                          DEBUG_PRINT1 ("    No match => pop_failure_jump.¥n");
                        }
                    }
                  else if ((re_opcode_t) p1[3] == charset)
                    {
                      int idx;
                      /* We win if the charset inside the loop
                         has no overlap with the one after the loop.  */
                      for (idx = 0;
                           idx < (int) p2[1] && idx < (int) p1[4];
                           idx++)
                        if ((p2[2 + idx] & p1[5 + idx]) != 0)
```

```
                    break;


                if (idx == p2[1] || idx == p1[4])
                  {
                    p[-3] = (unsigned char) pop_failure_jump;
                    DEBUG_PRINT1 ("    No match => pop_failure_jump.¥n");
                  }
              }
          }
      }
    p -= 2;             /* Point at relative address again.    */
    if ((re_opcode_t) p[-1] != pop_failure_jump)
      {
        p[-1] = (unsigned char) jump;
        DEBUG_PRINT1 ("    Match => jump.¥n");
        goto unconditional_jump;
      }
/* Note fall through.    */


/* The end of a simple repeat has a pop_failure_jump back to
     its matching on_failure_jump, where the latter will push a
     failure point.    The pop_failure_jump takes off failure
     points put on by this pop_failure_jump's matching
     on_failure_jump; we got through the pattern to here from the
     matching on_failure_jump, so didn't fail.    */
case pop_failure_jump:
    {
      /* We need to pass separate storage for the lowest and
           highest registers, even though we don't care about the
           actual values.    Otherwise, we will restore only one
           register from the stack, since lowest will == highest in
           `pop_failure_point'.    */
      active_reg_t dummy_low_reg, dummy_high_reg;
```

```
              unsigned char *pdummy;
              const char *sdummy;

              DEBUG_PRINT1 ("EXECUTING pop_failure_jump.¥n");
              POP_FAILURE_POINT (sdummy, pdummy,
                                 dummy_low_reg, dummy_high_reg,
                                 reg_dummy, reg_dummy, reg_info_dummy);
            }
            /* Note fall through.  */

          unconditional_jump:
#ifdef _LIBC
            DEBUG_PRINT2 ("¥n%p: ", p);
#else
            DEBUG_PRINT2 ("¥n0x%x: ", p);
#endif
            /* Note fall through.  */

          /* Unconditionally jump (without popping any failure points).  */
          case jump:
            EXTRACT_NUMBER_AND_INCR (mcnt, p);      /* Get the amount to jump.  */
            DEBUG_PRINT2 ("EXECUTING jump %d ", mcnt);
            p += mcnt;                                  /* Do the jump.  */
#ifdef _LIBC
            DEBUG_PRINT2 ("(to %p).¥n", p);
#else
            DEBUG_PRINT2 ("(to 0x%x).¥n", p);
#endif
            break;


          /* We need this opcode so we can detect where alternatives end
             in `group_match_null_string_p' et al.  */
          case jump_past_alt:
```

```
    DEBUG_PRINT1 ("EXECUTING jump_past_alt.¥n");
    goto unconditional_jump;


/* Normally, the on_failure_jump pushes a failure point, which
       then gets popped at pop_failure_jump.    We will end up at
       pop_failure_jump, also, and with a pattern of, say, `a+', we
       are skipping over the on_failure_jump, so we have to push
       something meaningless for pop_failure_jump to pop.    */
case dummy_failure_jump:
    DEBUG_PRINT1 ("EXECUTING dummy_failure_jump.¥n");
    /* It doesn't matter what we push for the string here.    What
         the code at `fail' tests is the value for the pattern.    */
    PUSH_FAILURE_POINT (0, 0, -2);
    goto unconditional_jump;


/* At the end of an alternative, we need to push a dummy failure
       point in case we are followed by a `pop_failure_jump', because
       we don't want the failure point for the alternative to be
       popped.    For example, matching `(a|ab)*' against `aab'
       requires that we match the `ab' alternative.    */
case push_dummy_failure:
    DEBUG_PRINT1 ("EXECUTING push_dummy_failure.¥n");
    /* See comments just above at `dummy_failure_jump' about the
         two zeroes.    */
    PUSH_FAILURE_POINT (0, 0, -2);
    break;

/* Have to succeed matching what follows at least n times.
       After that, handle like `on_failure_jump'.    */
case succeed_n:
    EXTRACT_NUMBER (mcnt, p + 2);
    DEBUG_PRINT2 ("EXECUTING succeed_n %d.¥n", mcnt);
```

```
            assert (mcnt >= 0);
            /* Originally, this is how many times we HAVE to succeed.    */
            if (mcnt > 0)
              {
                  mcnt--;
                  p += 2;
                  STORE_NUMBER_AND_INCR (p, mcnt);
#ifdef _LIBC
                  DEBUG_PRINT3 ("    Setting %p to %d.¥n", p - 2, mcnt);
#else
                  DEBUG_PRINT3 ("    Setting 0x%x to %d.¥n", p - 2, mcnt);
#endif
              }
            else if (mcnt == 0)
              {
#ifdef _LIBC
                DEBUG_PRINT2 ("    Setting two bytes from %p to no_op.¥n", p+2);
#else
                DEBUG_PRINT2 ("    Setting two bytes from 0x%x to no_op.¥n", p+2);
#endif
                p[2] = (unsigned char) no_op;
                p[3] = (unsigned char) no_op;
                goto on_failure;
              }
            break;


          case jump_n:
            EXTRACT_NUMBER (mcnt, p + 2);
            DEBUG_PRINT2 ("EXECUTING jump_n %d.¥n", mcnt);

            /* Originally, this is how many times we CAN jump.    */
            if (mcnt)
              {
```

```
                    mcnt--;
                    STORE_NUMBER (p + 2, mcnt);
#ifdef _LIBC
                    DEBUG_PRINT3 ("    Setting %p to %d.¥n", p + 2, mcnt);
#else
                    DEBUG_PRINT3 ("    Setting 0x%x to %d.¥n", p + 2, mcnt);
#endif
                    goto unconditional_jump;
                  }
                /* If don't have to jump any more, skip over the rest of command.    */
                else
                  p += 4;
                break;

              case set_number_at:
                {
                  DEBUG_PRINT1 ("EXECUTING set_number_at.¥n");

                  EXTRACT_NUMBER_AND_INCR (mcnt, p);
                  p1 = p + mcnt;
                  EXTRACT_NUMBER_AND_INCR (mcnt, p);
#ifdef _LIBC
                  DEBUG_PRINT3 ("    Setting %p to %d.¥n", p1, mcnt);
#else
                  DEBUG_PRINT3 ("    Setting 0x%x to %d.¥n", p1, mcnt);
#endif
                  STORE_NUMBER (p1, mcnt);
                  break;
                }

#if 0
              /* The DEC Alpha C compiler 3.x generates incorrect code for the
                 test    WORDCHAR_P (d - 1) != WORDCHAR_P (d)    in the expansion of
                 AT_WORD_BOUNDARY, so this code is disabled.    Expanding the
```

```
            macro and introducing temporary variables works around the bug.    */

          case wordbound:
            DEBUG_PRINT1 ("EXECUTING wordbound.¥n");
            if (AT_WORD_BOUNDARY (d))
              break;
            goto fail;

          case notwordbound:
            DEBUG_PRINT1 ("EXECUTING notwordbound.¥n");
            if (AT_WORD_BOUNDARY (d))
              goto fail;
            break;
#else
          case wordbound:
          {
            boolean prevchar, thischar;

            DEBUG_PRINT1 ("EXECUTING wordbound.¥n");
            if (AT_STRINGS_BEG (d) || AT_STRINGS_END (d))
              break;

            prevchar = WORDCHAR_P (d - 1);
            thischar = WORDCHAR_P (d);
            if (prevchar != thischar)
              break;
            goto fail;
          }

        case notwordbound:
          {
            boolean prevchar, thischar;

            DEBUG_PRINT1 ("EXECUTING notwordbound.¥n");
```

```
            if (AT_STRINGS_BEG (d) || AT_STRINGS_END (d))
              goto fail;

            prevchar = WORDCHAR_P (d - 1);
            thischar = WORDCHAR_P (d);
            if (prevchar != thischar)
              goto fail;
            break;
          }
#endif

          case wordbeg:
            DEBUG_PRINT1 ("EXECUTING wordbeg.¥n");
            if (WORDCHAR_P (d) && (AT_STRINGS_BEG (d) || !WORDCHAR_P (d - 1)))
              break;
            goto fail;

          case wordend:
            DEBUG_PRINT1 ("EXECUTING wordend.¥n");
            if (!AT_STRINGS_BEG (d) && WORDCHAR_P (d - 1)
                && (!WORDCHAR_P (d) || AT_STRINGS_END (d)))
              break;
            goto fail;

#ifdef emacs
          case before_dot:
            DEBUG_PRINT1 ("EXECUTING before_dot.¥n");
            if (PTR_CHAR_POS ((unsigned char *) d) >= point)
              goto fail;
            break;

          case at_dot:
            DEBUG_PRINT1 ("EXECUTING at_dot.¥n");
            if (PTR_CHAR_POS ((unsigned char *) d) != point)
```

```
        goto fail;
      break;

    case after_dot:
      DEBUG_PRINT1 ("EXECUTING after_dot.¥n");
      if (PTR_CHAR_POS ((unsigned char *) d) <= point)
        goto fail;
      break;

    case syntaxspec:
      DEBUG_PRINT2 ("EXECUTING syntaxspec %d.¥n", mcnt);
      mcnt = *p++;
      goto matchsyntax;

    case wordchar:
      DEBUG_PRINT1 ("EXECUTING Emacs wordchar.¥n");
      mcnt = (int) Sword;
    matchsyntax:
      PREFETCH ();
      /* Can't use *d++ here; SYNTAX may be an unsafe macro.  */
      d++;
      if (SYNTAX (d[-1]) != (enum syntaxcode) mcnt)
        goto fail;
      SET_REGS_MATCHED ();
      break;

    case notsyntaxspec:
      DEBUG_PRINT2 ("EXECUTING notsyntaxspec %d.¥n", mcnt);
      mcnt = *p++;
      goto matchnotsyntax;

    case notwordchar:
      DEBUG_PRINT1 ("EXECUTING Emacs notwordchar.¥n");
      mcnt = (int) Sword;
```

```
          matchnotsyntax:
            PREFETCH ();
            /* Can't use *d++ here; SYNTAX may be an unsafe macro.  */
            d++;
            if (SYNTAX (d[-1]) == (enum syntaxcode) mcnt)
              goto fail;
            SET_REGS_MATCHED ();
            break;


#else /* not emacs */
          case wordchar:
            DEBUG_PRINT1 ("EXECUTING non-Emacs wordchar.¥n");
            PREFETCH ();
            if (!WORDCHAR_P (d))
              goto fail;
            SET_REGS_MATCHED ();
            d++;
            break;


          case notwordchar:
            DEBUG_PRINT1 ("EXECUTING non-Emacs notwordchar.¥n");
            PREFETCH ();
            if (WORDCHAR_P (d))
              goto fail;
            SET_REGS_MATCHED ();
            d++;
            break;
#endif /* not emacs */


          default:
            abort ();
          }
        continue;  /* Successfully executed one pattern command; keep going.  */
```

```
/* We goto here if a matching operation fails. */
fail:
  if (!FAIL_STACK_EMPTY ())
    { /* A restart point is known.  Restore to that state.  */
      DEBUG_PRINT1 ("¥nFAIL:¥n");
      POP_FAILURE_POINT (d, p,
                         lowest_active_reg, highest_active_reg,
                         regstart, regend, reg_info);

      /* If this failure point is a dummy, try the next one.  */
      if (!p)
        goto fail;

      /* If we failed to the end of the pattern, don't examine *p.  */
      assert (p <= pend);
      if (p < pend)
        {
          boolean is_a_jump_n = false;

          /* If failed to a backwards jump that's part of a repetition
             loop, need to pop this failure point and use the next one.  */
          switch ((re_opcode_t) *p)
            {
            case jump_n:
              is_a_jump_n = true;
            case maybe_pop_jump:
            case pop_failure_jump:
            case jump:
              p1 = p + 1;
              EXTRACT_NUMBER_AND_INCR (mcnt, p1);
              p1 += mcnt;

              if ((is_a_jump_n && (re_opcode_t) *p1 == succeed_n)
```

```
                        || (!is_a_jump_n
                            && (re_opcode_t) *p1 == on_failure_jump))
                      goto fail;
                    break;
                  default:
                    /* do nothing */ ;
                  }
              }

            if (d >= string1 && d <= end1)
              dend = end_match_1;
          }
        else
          break;     /* Matching at this starting point really fails.    */
      } /* for (;;) */

    if (best_regs_set)
      goto restore_best_regs;

    FREE_VARIABLES ();

    return -1;                                  /* Failure to match.    */
} /* re_match_2 */
```

```
/* Subroutine definitions for re_match_2.    */


/* We are passed P pointing to a register number after a start_memory.

     Return true if the pattern up to the corresponding stop_memory can
     match the empty string, and false otherwise.

     If we find the matching stop_memory, sets P to point to one past its number.
     Otherwise, sets P to an undefined byte less than or equal to END.

     We don't handle duplicates properly (yet).    */

static boolean
group_match_null_string_p (unsigned char **p, unsigned char *end,
      register_info_type *reg_info)
{
   int mcnt;
   /* Point to after the args to the start_memory.    */
   unsigned char *p1 = *p + 2;

   while (p1 < end)
      {
         /* Skip over opcodes that can match nothing, and return true or
              false, as appropriate, when we get to one that can't, or to the
              matching stop_memory.    */

         switch ((re_opcode_t) *p1)
            {
            /* Could be either a loop or a series of alternatives.    */
            case on_failure_jump:
               p1++;
               EXTRACT_NUMBER_AND_INCR (mcnt, p1);
```

```
/* If the next operation is not a jump backwards in the
     pattern.    */

if (mcnt >= 0)
   {
      /* Go through the on_failure_jumps of the alternatives,
           seeing if any of the alternatives cannot match nothing.
           The last alternative starts with only a jump,
           whereas the rest start with on_failure_jump and end
           with a jump, e.g., here is the pattern for `a|b|c':

           /on_failure_jump/0/6/exactn/1/a/jump_past_alt/0/6
           /on_failure_jump/0/6/exactn/1/b/jump_past_alt/0/3
           /exactn/1/c

           So, we have to first go through the first (n-1)
           alternatives and then deal with the last one separately.    */


      /* Deal with the first (n-1) alternatives, which start
           with an on_failure_jump (see above) that jumps to right
           past a jump_past_alt.    */

      while ((re_opcode_t) p1[mcnt-3] == jump_past_alt)
        {
          /* `mcnt' holds how many bytes long the alternative
               is, including the ending `jump_past_alt' and
               its number.    */

          if (!alt_match_null_string_p (p1, p1 + mcnt - 3,
                                                    reg_info))
            return false;
```

```
            /* Move to right after this alternative, including the
                jump_past_alt.  */
            p1 += mcnt;

            /* Break if it's the beginning of an n-th alternative
                that doesn't begin with an on_failure_jump.  */
            if ((re_opcode_t) *p1 != on_failure_jump)
              break;

            /* Still have to check that it's not an n-th
                alternative that starts with an on_failure_jump.  */
            p1++;
            EXTRACT_NUMBER_AND_INCR (mcnt, p1);
            if ((re_opcode_t) p1[mcnt-3] != jump_past_alt)
              {
                /* Get to the beginning of the n-th alternative.  */
                p1 -= 3;
                break;
              }
          }

        /* Deal with the last alternative: go back and get number
            of the `jump_past_alt' just before it.  `mcnt' contains
            the length of the alternative.  */
        EXTRACT_NUMBER (mcnt, p1 - 2);

        if (!alt_match_null_string_p (p1, p1 + mcnt, reg_info))
          return false;

        p1 += mcnt;           /* Get past the n-th alternative.  */
      } /* if mcnt > 0 */
    break;
```

```
            case stop_memory:
               assert (p1[1] == **p);
               *p = p1 + 2;
               return true;



            default:
               if (!common_op_match_null_string_p (&p1, end, reg_info))
                  return false;
            }
         } /* while p1 < end */

      return false;
   } /* group_match_null_string_p */



   /* Similar to group_match_null_string_p, but doesn't deal with alternatives:
        It expects P to be the first byte of a single alternative and END one
        byte past the last. The alternative can contain groups.    */

   static boolean
   alt_match_null_string_p (unsigned char *p, unsigned char *end,
         register_info_type *reg_info)
   {
      int mcnt;
      unsigned char *p1 = p;

      while (p1 < end)
         {
            /* Skip over opcodes that can match nothing, and break when we get
                 to one that can't.    */

            switch ((re_opcode_t) *p1)
               {
```

```
            /* It's a loop.   */
            case on_failure_jump:
               p1++;
               EXTRACT_NUMBER_AND_INCR (mcnt, p1);
               p1 += mcnt;
               break;

            default:
               if (!common_op_match_null_string_p (&p1, end, reg_info))
                  return false;
            }
      }   /* while p1 < end */

   return true;
} /* alt_match_null_string_p */


/* Deals with the ops common to group_match_null_string_p and
     alt_match_null_string_p.

     Sets P to one after the op and its arguments, if any.   */

static boolean
common_op_match_null_string_p (unsigned char **p, unsigned char *end,
      register_info_type *reg_info)
{
   int mcnt;
   boolean ret;
   int reg_no;
   unsigned char *p1 = *p;

   switch ((re_opcode_t) *p1++)
      {
      case no_op:
```

```
        case begline:
        case endline:
        case begbuf:
        case endbuf:
        case wordbeg:
        case wordend:
        case wordbound:
        case notwordbound:
#ifdef emacs
        case before_dot:
        case at_dot:
        case after_dot:
#endif
          break;


        case start_memory:
          reg_no = *p1;
          assert (reg_no > 0 && reg_no <= MAX_REGNUM);
          ret = group_match_null_string_p (&p1, end, reg_info);

          /* Have to set this here in case we're checking a group which
              contains a group and a back reference to it.    */

          if (REG_MATCH_NULL_STRING_P (reg_info[reg_no]) == MATCH_NULL_UNSET_VALUE)
            REG_MATCH_NULL_STRING_P (reg_info[reg_no]) = ret;

          if (!ret)
            return false;
          break;

        /* If this is an optimized succeed_n for zero times, make the jump.    */
        case jump:
          EXTRACT_NUMBER_AND_INCR (mcnt, p1);
          if (mcnt >= 0)
```

```
          p1 += mcnt;
        else
          return false;
        break;

      case succeed_n:
        /* Get to the number of times to succeed.    */
        p1 += 2;
        EXTRACT_NUMBER_AND_INCR (mcnt, p1);

        if (mcnt == 0)
          {
            p1 -= 4;
            EXTRACT_NUMBER_AND_INCR (mcnt, p1);
            p1 += mcnt;
          }
        else
          return false;
        break;

      case duplicate:
        if (!REG_MATCH_NULL_STRING_P (reg_info[*p1]))
          return false;
        break;

      case set_number_at:
        p1 += 4;

      default:
        /* All other opcodes mean we cannot match the empty string.    */
        return false;
    }

  *p = p1;
```

```
    return true;
} /* common_op_match_null_string_p */



/* Return zero if TRANSLATE[S1] and TRANSLATE[S2] are identical for LEN
     bytes; nonzero otherwise.    */

static int
bcmp_translate (const char *s1, const char *s2,
        register int len, RE_TRANSLATE_TYPE translate)
{
   register const unsigned char *p1 = (const unsigned char *) s1;
   register const unsigned char *p2 = (const unsigned char *) s2;
   while (len)
      {
         if (translate[*p1++] != translate[*p2++]) return 1;
         len--;
      }
   return 0;
}
```

```
/* Entry points for GNU code.  */

/* re_compile_pattern is the GNU regular expression compiler: it
     compiles PATTERN (of length SIZE) and puts the result in BUFP.
     Returns 0 if the pattern was valid, otherwise an error string.

     Assumes the `allocated' (and perhaps `buffer') and `translate' fields
     are set in BUFP on entry.

     We call regex_compile to do the actual compilation.  */

const char *
re_compile_pattern (const char *pattern, size_t length,
        struct re_pattern_buffer *bufp)
{
   reg_errcode_t ret;

   /* GNU code is written to assume at least RE_NREGS registers will be set
        (and at least one extra will be -1).  */
   bufp->regs_allocated = REGS_UNALLOCATED;

   /* And GNU code determines whether or not to get register information
        by passing null for the REGS argument to re_match, etc., not by
        setting no_sub.  */
   bufp->no_sub = 0;

   /* Match anchors at newline.  */
   bufp->newline_anchor = 1;

   ret = regex_compile (pattern, length, re_syntax_options, bufp);

   if (!ret)
     return NULL;
```

```
    return gettext (re_error_msgid[(int) ret]);
}
```

```
/* Entry points compatible with 4.2 BSD regex library.  We don't define
     them unless specifically requested.  */

#if defined (_REGEX_RE_COMP) || defined (_LIBC)

/* BSD has one and only one pattern buffer.  */
static struct re_pattern_buffer re_comp_buf;

char *
#ifdef _LIBC
/* Make these definitions weak in libc, so POSIX programs can redefine
     these names if they don't use our functions, and still use
     regcomp/regexec below without link errors.  */
weak_function
#endif
re_comp (s)
      const char *s;
{
   reg_errcode_t ret;

   if (!s)
      {
         if (!re_comp_buf.buffer)
            return gettext ("No previous regular expression");
         return 0;
      }

   if (!re_comp_buf.buffer)
      {
         re_comp_buf.buffer = (unsigned char *) malloc (200);
         if (re_comp_buf.buffer == NULL)
            return gettext (re_error_msgid[(int) REG_ESPACE]);
         re_comp_buf.allocated = 200;
```

```
      re_comp_buf.fastmap = (char *) malloc (1 << BYTEWIDTH);
      if (re_comp_buf.fastmap == NULL)
        return gettext (re_error_msgid[(int) REG_ESPACE]);
    }

  /* Since `re_exec' always passes NULL for the `regs' argument, we
     don't need to initialize the pattern buffer fields which affect it.  */

  /* Match anchors at newlines.  */
  re_comp_buf.newline_anchor = 1;

  ret = regex_compile (s, strlen (s), re_syntax_options, &re_comp_buf);

  if (!ret)
    return NULL;

  /* Yes, we're discarding `const' here if !HAVE_LIBINTL.  */
  return (char *) gettext (re_error_msgid[(int) ret]);
}


int
#ifdef _LIBC
weak_function
#endif
re_exec (s)
    const char *s;
{
  const int len = strlen (s);
  return
    0 <= re_search (&re_comp_buf, s, len, 0, len, (struct re_registers *) 0);
}
```

```
#endif /* _REGEX_RE_COMP */
```

```
/* POSIX.2 functions.  Don't define these for Emacs.  */

#ifndef emacs

/* regcomp takes a regular expression as a string and compiles it.

    PREG is a regex_t *.  We do not expect any fields to be initialized,
    since POSIX says we shouldn't.  Thus, we set

      `buffer' to the compiled pattern;
      `used' to the length of the compiled pattern;
      `syntax' to RE_SYNTAX_POSIX_EXTENDED if the
        REG_EXTENDED bit in CFLAGS is set; otherwise, to
        RE_SYNTAX_POSIX_BASIC;
      `newline_anchor' to REG_NEWLINE being set in CFLAGS;
      `fastmap' and `fastmap_accurate' to zero;
      `re_nsub' to the number of subexpressions in PATTERN.

    PATTERN is the address of the pattern string.

    CFLAGS is a series of bits which affect compilation.

      If REG_EXTENDED is set, we use POSIX extended syntax; otherwise, we
      use POSIX basic syntax.

      If REG_NEWLINE is set, then . and [^...] don't match newline.
      Also, regexec will try a match beginning after every newline.

      If REG_ICASE is set, then we considers upper- and lowercase
      versions of letters to be equivalent when matching.

      If REG_NOSUB is set, then when PREG is passed to regexec, that
      routine will report only success or failure, and nothing about the
```

```
        registers.

      It returns 0 if it succeeds, nonzero if it doesn't.  (See regex.h for
      the return codes and their meanings.)  */

int
regcomp (regex_t *preg, const char *pattern, int cflags)
{
   reg_errcode_t ret;
   reg_syntax_t syntax
     = (cflags & REG_EXTENDED) ?
        RE_SYNTAX_POSIX_EXTENDED : RE_SYNTAX_POSIX_BASIC;

   /* regex_compile will allocate the space for the compiled pattern.  */
   preg->buffer = 0;
   preg->allocated = 0;
   preg->used = 0;

   /* Don't bother to use a fastmap when searching.  This simplifies the
        REG_NEWLINE case: if we used a fastmap, we'd have to put all the
        characters after newlines into the fastmap.  This way, we just try
        every character.  */
   preg->fastmap = 0;

   if (cflags & REG_ICASE)
     {
        unsigned i;

        preg->translate
          = (RE_TRANSLATE_TYPE) malloc (CHAR_SET_SIZE
                                        * sizeof (*(RE_TRANSLATE_TYPE)0));
        if (preg->translate == NULL)
          return (int) REG_ESPACE;
```

```
        /* Map uppercase characters to corresponding lowercase ones.  */
        for (i = 0; i < CHAR_SET_SIZE; i++)
          preg->translate[i] = ISUPPER (i) ? tolower (i) : i;
      }
    else
      preg->translate = NULL;

    /* If REG_NEWLINE is set, newlines are treated differently.  */
    if (cflags & REG_NEWLINE)
      { /* REG_NEWLINE implies neither . nor [^...] match newline.  */
        syntax &= ~RE_DOT_NEWLINE;
        syntax |= RE_HAT_LISTS_NOT_NEWLINE;
        /* It also changes the matching behavior.  */
        preg->newline_anchor = 1;
      }
    else
      preg->newline_anchor = 0;

    preg->no_sub = !!(cflags & REG_NOSUB);

    /* POSIX says a null character in the pattern terminates it, so we
       can use strlen here in compiling the pattern.  */
    ret = regex_compile (pattern, strlen (pattern), syntax, preg);

    /* POSIX doesn't distinguish between an unmatched open-group and an
       unmatched close-group: both are REG_EPAREN.  */
    if (ret == REG_ERPAREN) ret = REG_EPAREN;

    return (int) ret;
}


/* regexec searches for a given pattern, specified by PREG, in the
   string STRING.
```

```
    If NMATCH is zero or REG_NOSUB was set in the cflags argument to
    `regcomp', we ignore PMATCH.  Otherwise, we assume PMATCH has at
    least NMATCH elements, and we set them to the offsets of the
    corresponding matched substrings.

    EFLAGS specifies `execution flags' which affect matching: if
    REG_NOTBOL is set, then ^ does not match at the beginning of the
    string; if REG_NOTEOL is set, then $ does not match at the end.

    We return 0 if we find a match and REG_NOMATCH if not.  */

int
regexec (const regex_t *preg, const char *string,
      size_t nmatch, regmatch_t pmatch[], int eflags)
{
  int ret;
  struct re_registers regs;
  regex_t private_preg;
  int len = strlen (string);
  boolean want_reg_info = !preg->no_sub && nmatch > 0;

  private_preg = *preg;

  private_preg.not_bol = !!(eflags & REG_NOTBOL);
  private_preg.not_eol = !!(eflags & REG_NOTEOL);

  /* The user has told us exactly how many registers to return
     information about, via `nmatch'.  We have to pass that on to the
     matching routines.  */
  private_preg.regs_allocated = REGS_FIXED;

  if (want_reg_info)
    {
```

```
      regs.num_regs = nmatch;
      regs.start = TALLOC (nmatch, regoff_t);
      regs.end = TALLOC (nmatch, regoff_t);
      if (regs.start == NULL || regs.end == NULL)
        return (int) REG_NOMATCH;
    }

  /* Perform the searching operation.  */
  ret = re_search (&private_preg, string, len,
                   /* start: */ 0, /* range: */ len,
                   want_reg_info ? &regs : (struct re_registers *) 0);

  /* Copy the register information to the POSIX structure.  */
  if (want_reg_info)
    {
      if (ret >= 0)
        {
          unsigned r;

          for (r = 0; r < nmatch; r++)
            {
              pmatch[r].rm_so = regs.start[r];
              pmatch[r].rm_eo = regs.end[r];
            }
        }

      /* If we needed the temporary register info, free the space now.  */
      free (regs.start);
      free (regs.end);
    }

  /* We want zero return to mean success, unlike `re_search'.  */
  return ret >= 0 ? (int) REG_NOERROR : (int) REG_NOMATCH;
}
```

```
/* Returns a message corresponding to an error code, ERRCODE, returned
     from either regcomp or regexec.      We don't use PREG here.    */

size_t
regerror (int errcode, const regex_t *preg, char *errbuf, size_t errbuf_size)
{
   const char *msg;
   size_t msg_size;

   if (errcode < 0
         || errcode >= (int) (sizeof (re_error_msgid)
                                    / sizeof (re_error_msgid[0])))
      /* Only error codes returned by the rest of the code should be passed
         to this routine.    If we are given anything else, or if other regex
         code generates an invalid error code, then the program has a bug.
         Dump core so we can fix it.    */
      abort ();

   msg = gettext (re_error_msgid[errcode]);

   msg_size = strlen (msg) + 1; /* Includes the null.    */

   if (errbuf_size != 0)
     {
       if (msg_size > errbuf_size)
         {
           strncpy (errbuf, msg, errbuf_size - 1);
           errbuf[errbuf_size - 1] = 0;
         }
       else
         strcpy (errbuf, msg);
     }
```

```
    return msg_size;
}


/* Free dynamically allocated space used by PREG.    */

void
regfree (regex_t *preg)
{
    if (preg->buffer != NULL)
        free (preg->buffer);
    preg->buffer = NULL;

    preg->allocated = 0;
    preg->used = 0;

    if (preg->fastmap != NULL)
        free (preg->fastmap);
    preg->fastmap = NULL;
    preg->fastmap_accurate = 0;

    if (preg->translate != NULL)
        free (preg->translate);
    preg->translate = NULL;
}

#endif /* not emacs    */
```