NAME

regexp − regular expression compile and match routines

SYNOPSIS

#define INIT <declarations>
#define GETC() <getc code>
#define PEEKC() <peekc code>
#define UNGETC(c) <ungetc code>
#define RETURN(pointer) <return code>
#define ERROR(val) <error code>

#include <regexp.h>

char *compile(instring, expbuf, endbuf, eof)
char *instring, *expbuf, *endbuf;

int step(string, expbuf)
char *string, *expbuf;

DESCRIPTION

This page describes general purpose regular expression matching routines in the form of *ed*(1), defined in **/usr/include/regexp.h**. Programs such as *ed*(1), *sed*(1), *grep*(1), *bs*(1), *expr*(1), etc., which perform regular expression matching use this source file. In this way, only this file need be changed to maintain regular expression compatibility.

The interface to this file is unpleasantly complex. Programs that include this file must have the following five macros declared before the "#include <regexp.h>" statement. These macros are used by the *compile* routine.

GETC()	Return the value of the next character in the regular expression pattern. Successive calls to GETC() should return successive characters of the regular expression.
PEEKC()	Return the next character in the regular expression. Successive calls to PEEKC() should return the same character (which should also be the next character returned by GETC()).
UNGETC(*c*)	Cause the argument *c* to be returned by the next call to GETC() (and PEEKC()). No more that one character of pushback is ever needed and this character is guaranteed to be the last character read by GETC(). The value of the macro UNGETC(*c*) is always ignored.
RETURN(*pointer*)	This macro is used on normal exit of the *compile* routine. The value of the argument *pointer* is a pointer to the character after the last character of the compiled regular expression. This is useful to programs which have memory allocation to manage.
ERROR(*val*)	This is the abnormal return from the *compile* routine. The argument *val* is an error number (see table below for meanings). This call should never return.

ERROR	*MEANING*
11	Range endpoint too large.
16	Bad number.
25	"\digit" out of range.
36	Illegal or missing delimiter.
41	No remembered search string.
42	\(\) imbalance.
43	Too many \(.
44	More than 2 numbers given in \{ \}.
45	} expected after \.
46	First number exceeds second in \{ \}.
49	[ ] imbalance.
50	Regular expression overflow.

The syntax of the *compile* routine is as follows:

```
compile(instring, expbuf, endbuf, eof)
```

The first parameter "instring" is never used explicitly by the *compile* routine but is useful for programs that pass down different pointers to input characters. It is sometimes used in the INIT declaration (see below). Programs which call functions to input characters or have characters in an external array can pass down a value of ((char *) 0) for this parameter.

The next parameter "expbuf" is a character pointer. It points to the place where the compiled regular expression will be placed.

The parameter "endbuf" is one more that the highest address that the compiled regular expression may be placed. If the compiled expression cannot fit in (endbuf−expbuf) bytes, a call to ERROR(50) is made.

The parameter "eof" is the character which marks the end of the regular expression. For example, in *ed*(1), this character is usually a /.

Each programs that includes this file must have a **#define** statement for INIT. This definition will be placed right after the declaration for the function *compile* and the opening curly brace ({). It is used for dependent declarations and initializations. Most often it is used to set a register variable to point the beginning of the regular expression so that this register variable can be used in the declarations for **GETC()**, **PEE()** and **UNGETC()**. Otherwise it can be used to declare external variables that might be used by **GETC()**, **PEEKC()** and **UNGETC()**. See the example below of the declarations taken from *grep*(1).

There are other functions in this file which perform actual regular expression matching, one of which is the function *step*. The call to *step* is as follows:

```
step(string, expbuf)
```

The first parameter to *step* is a pointer to a string of characters to be checked for a match. This string should be null terminated.

The second parameter "expbuf" is the compiled regular expression which was obtained by a call of the function *compile*.

The function *step* returns one, if the given string matches the regular expression, and zero if the expressions do not match. If there is a match, two external character pointers are set as a side effect to the call to *step*. The variable set in *step* is "loc1". This is a pointer to the first character that matched the regular expression. The variable "loc2", which is set by the function *advance*, points the character after the last character that matches the regular expression. Thus if the regular expression matches the entire line, loc1 will point to the first character of "string" and "loc2" will point to the null at the end of "string".

Step uses the external variable "circf" which is set by *compile* if the regular expression begins with ^. If this is set then *step* will only try to match the regular expression to the beginning of the string. If more than one regular expression is to be compiled before the the first is executed the value of "circf" should be saved for each compiled expression and "circf" should be set to that saved value before each call to *step*.

The function *advance* is called from *step* with the same arguments as *step*. The purpose of *step* is to step through the "string" argument and call *advance* until *advance* returns a one indicating a match or until the end of "string" is reached. If one wants to constrain "string" to the beginning of the line in all cases, *step* need not be called, simply call *advance*.

When *advance* encounters a * or \{ \} sequence in the regular expression it will advance its pointer to the string to be matched as far as possible and will recursively call itself trying to match the rest of the string to the rest of the regular expression. As long as there is no match, *advance* will back up along the string until it finds a match or reaches the point in the string that initially matched the * or \{ \}. It is sometimes desirable to stop this backing up before the initial point in the string is reached. If the external character pointer "locs" is equal to the point in the string at sometime during the backing up process, *advance* will break out of the loop that backs up and will return zero. This is used be *ed*(1) and *sed*(1) for substitutions done globally (not just the first occurrence, but the whole line) so, for example, expressions like "s/y*//g" do not loop forever.

The routines *ecmp* and *getrange* are trivial and are called by the routines previously mentioned.

EXAMPLES

The following is an example of how the regular expression macros and calls look from *grep*(1):

```
#define INIT         register char *sp = instring;   /* First arg points to RE string */
#define GETC()       (*sp++)
#define PEEKC()      (*sp)
#define UNGETC(c)    (--sp)
#define RETURN(c)    return;
#define ERROR(c)     regerr()

#include <regexp.h>
...
               compile(*argv, expbuf, &expbuf[ESIZE], '\0');
...
               if(step(linebuf, expbuf))
                              succeed();
```

FILES

/usr/include/regexp.h

SEE ALSO

ed(1), grep(1), sed(1).

BUGS

The handling of "circf" is kludgy.
The routine *ecmp* is equivalent to the Standard I/O routine *strncmp* and should be replaced by that routine.
The actual code is probably easier to understand than this manual page.