NAME

regexp, compile, step, advance – simple regular expression compile and match routines

SYNOPSIS

#define INIT *declarations*
#define GETC(void) *getc code*
#define PEEKC(void) *peekc code*
#define UNGETC(void) *ungetc code*
#define RETURN(*ptr***)** *return code*
#define ERROR(*val***)** *error code*

#include <regexp.h>

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

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

int advance(char ∗*string,* **char** ∗*expbuf***);**

extern char ∗**loc1,** ∗**loc2,** ∗**locs;**

DESCRIPTION

Regular Expressions (REs) provide a mechanism to select specific strings from a set of character strings. The Simple Regular Expressions described below differ from the Internationalized Regular Expressions described on the **regex**(5) manual page in the following ways:

- only Basic Regular Expressions are supported
- the Internationalization features—character class, equivalence class, and multi-character collation—are not supported.

The functions **step( )**, **advance( )**, and **compile( )** are general purpose regular expression matching routines to be used in programs that perform regular expression matching. These functions are defined by the **<regexp.h>** header.

The functions **step( )** and **advance( )** do pattern matching given a character string and a compiled regular expression as input.

The function **compile( )** takes as input a regular expression as defined below and produces a compiled expression that can be used with **step( )** or **advance( )**.

Basic Regular Expressions

A regular expression specifies a set of character strings. A member of this set of strings is said to be matched by the regular expression. Some characters have special meaning when used in a regular expression; other characters stand for themselves.

The following *one-character RE*s match a *single* character:

1.1 An ordinary character (*not* one of those discussed in 1.2 below) is a one-character RE that matches itself.

1.2 A backslash (****) followed by any special character is a one-character RE that matches the special character itself. The special characters are:

a. **.**, ∗, **[** , and **** (period, asterisk, left square bracket, and backslash, respectively), which are always special, *except* when they appear within square brackets (**[ ]**; see 1.4 below).

b. ˆ (caret or circumflex), which is special at the *beginning* of an *entire* RE (see 4.1 and 4.3 below), or when it immediately follows the left of a pair of square brackets (**[ ]**) (see 1.4 below).

c. **$** (dollar sign), which is special at the **end** of an *entire* RE (see 4.2 below).

d. The character used to bound (that is, delimit) an entire RE, which is special for that RE (for example, see how slash (**/**) is used in the **g** command, below.)

1.3 A period (**.**) is a one-character RE that matches any character except new-line.

1.4 A non-empty string of characters enclosed in square brackets (**[ ]**) is a one-character RE that matches *any one* character in that string. If, however, the first character of the string is a circumflex (ˆ), the one-character RE matches any character *except* new-line and the remaining characters in the string. The ˆ has this special meaning *only* if it occurs first in the string. The minus (–) may be used to indicate a range of consecutive characters; for example, **[0–9]** is equivalent to **[0123456789]**. The – loses this special meaning if it occurs first (after an initial ˆ, if any) or last in the string. The right square bracket (**]**) does not terminate such a string when it is the first character within it (after an initial ˆ, if any); for example, **[ ]a–f]** matches either a right square bracket (**]**) or one of the ASCII letters **a** through **f** inclusive. The four characters listed in 1.2.a above stand for themselves within such a string of characters.

The following rules may be used to construct REs from one-character REs:

2.1 A one-character RE is a RE that matches whatever the one-character RE matches.

2.2 A one-character RE followed by an asterisk (*) is a RE that matches **0** or more occurrences of the one-character RE. If there is any choice, the longest leftmost string that permits a match is chosen.

2.3 A one-character RE followed by \{*m*\}, \{*m,*\}, or \{*m,n*\} is a RE that matches a *range* of occurrences of the one-character RE. The values of *m* and *n* must be non-negative integers less than 256; \{*m*\} matches *exactly m* occurrences; \{*m,*\} matches *at least m* occurrences; \{*m,n*\} matches *any number* of occurrences *between m* and *n* inclusive. Whenever a choice exists, the RE matches as many occurrences as possible.

2.4 The concatenation of REs is a RE that matches the concatenation of the strings matched by each component of the RE.

2.5 A RE enclosed between the character sequences **\(** and **\)** is a RE that matches whatever the unadorned RE matches.

2.6 The expression **** *n* matches the same string of characters as was matched by an expression enclosed between **\(** and **\)** *earlier* in the same RE. Here *n* is a digit; the sub-expression specified is that beginning with the *n*-th occurrence of **\(** counting from the left. For example, the expression **ˆ\(.*\)\1$** matches a line consisting of two repeated appearances of the same string.

A RE may be constrained to match words.

3.1 **\ <** constrains a RE to match the beginning of a string or to follow a character that is not a digit, underscore, or letter. The first character matching the RE must be a digit, underscore, or letter.

3.2 **\ >** constrains a RE to match the end of a string or to precede a character that is not a digit, underscore, or letter.

An *entire RE* may be constrained to match only an initial segment or final segment of a line (or both).

4.1 A circumflex (^) at the beginning of an entire RE constrains that RE to match an *initial* segment of a line.

4.2 A dollar sign (**$**) at the end of an entire RE constrains that RE to match a *final* segment of a line.

4.3 The construction ^*entire RE* **$** constrains the entire RE to match the entire line.

The null RE (for example, **//**) is equivalent to the last RE encountered.

Addressing with REs

Addresses are constructed as follows:

1. The character "**.**" addresses the current line.
2. The character "**$**" addresses the last line of the buffer.
3. A decimal number *n* addresses the *n*-th line of the buffer.
4. ′*x* addresses the line marked with the mark name character *x*, which must be an ASCII lower-case letter (**a**–**z**). Lines are marked with the **k** command described below.
5. A RE enclosed by slashes (**/**) addresses the first line found by searching *forward* from the line *following* the current line toward the end of the buffer and stopping at the first line containing a string matching the RE. If necessary, the search wraps around to the beginning of the buffer and continues up to and including the current line, so that the entire buffer is searched.
6. A RE enclosed in question marks (**?**) addresses the first line found by searching *backward* from the line *preceding* the current line toward the beginning of the buffer and stopping at the first line containing a string matching the RE. If necessary, the search wraps around to the end of the buffer and continues up to and including the current line.
7. An address followed by a plus sign (**+**) or a minus sign (–) followed by a decimal number specifies that address plus (respectively minus) the indicated number of lines. A shorthand for .+5 is .5.
8. If an address begins with **+** or –, the addition or subtraction is taken with respect to the current line; for example, –**5** is understood to mean **.**–**5**.
9. If an address ends with **+** or –, then 1 is added to or subtracted from the address, respectively. As a consequence of this rule and of Rule 8, immediately above, the address – refers to the line preceding the current line. (To maintain compatibility with earlier versions of the editor, the character ^ in addresses is entirely

equivalent to –.) Moreover, trailing **+** and – characters have a cumulative effect, so –– refers to the current line less 2.

10. For convenience, a comma (**,**) stands for the address pair **1,$**, while a semicolon (**;**) stands for the pair **.,$**.

Characters With Special Meaning

Characters that have special meaning except when they appear within square brackets ([ ]) or are preceded by \ are: **.**, *, [, \ . Other special characters, such as **$** have special meaning in more restricted contexts.

The character ^ at the beginning of an expression permits a successful match only immediately after a newline, and the character **$** at the end of an expression requires a trailing newline.

Two characters have special meaning only when used within square brackets. The character – denotes a range, **[** *c*–*c***]**, unless it is just after the open bracket or before the closing bracket, **[** –*c***]** or **[** *c*–**]** in which case it has no special meaning. When used within brackets, the character ^ has the meaning *complement of* if it immediately follows the open bracket (example: **[**^*c***]**); elsewhere between brackets (example: **[***c*^**]**) it stands for the ordinary character ^.

The special meaning of the \ operator can be escaped only by preceding it with another \ , for example \\ .

Macros

Programs must have the following five macros declared before the **#include <regexp.h>** statement. These macros are used by the **compile( )** routine. The macros **GETC**, **PEEKC**, and **UNGETC** operate on the regular expression given as input to **compile( )**.

GETC This macro returns the value of the next character (byte) in the regular expression pattern. Successive calls to **GETC** should return successive characters of the regular expression.

PEEKC This macro returns the next character (byte) in the regular expression. Immediately successive calls to **PEEKC** should return the same character, which should also be the next character returned by **GETC**.

UNGETC This macro causes the argument **c** to be returned by the next call to **GETC** and **PEEKC**. No more than one character of pushback is ever needed and this character is guaranteed to be the last character read by **GETC**. The return value of the macro **UNGETC(c)** is always ignored.

RETURN(*ptr***)** This macro is used on normal exit of the **compile( )** routine. The value of the argument *ptr* 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 macro is the abnormal return from the **compile( )** routine. The argument *val* is an error number (see ERRORS below for meanings). This call should never return.

compile( )

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 than the highest address where 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. This character is usually a **/**.

Each program that includes the **<regexp.h>** header file must have a **#define** statement for **INIT**. It is used for dependent declarations and initializations. Most often it is used to set a register variable to point to the beginning of the regular expression so that this register variable can be used in the declarations for **GETC**, **PEEKC**, and **UNGETC**. Otherwise it can be used to declare external variables that might be used by **GETC**, **PEEKC** and **UNGETC**. (See **EXAMPLES** below.)

step( ), advance( )

The first parameter to the **step( )** and **advance( )** functions 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 to the function **compile( )**.

The function **step( )** returns non-zero if some substring of *string* matches the regular expression in *expbuf* and **0** if there is no match. If there is a match, two external character pointers are set as a side effect to the call to **step( )**. The variable **loc1** points to the first character that matched the regular expression; the variable **loc2** points to the character after the last character that matches the regular expression. Thus if the regular expression matches the entire input string, **loc1** will point to the first character of *string* and **loc2** will point to the null at the end of *string*.

The function **advance( )** returns non-zero if the initial substring of *string* matches the regular expression in *expbuf*. If there is a match, an external character pointer, **loc2**, is set as a side effect. The variable **loc2** points to the next character in *string* after the last character that matched.

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.

The external variables **circf**, **sed**, and **nbra** are reserved.

EXAMPLES

The following is an example of how the regular expression macros and calls might be defined by an application program:

```
#define INIT        register char *sp = instring;
#define GETC        (*sp++)
#define PEEKC       (*sp)
#define UNGETC(c)   (−−sp)
#define RETURN(*c)  return;
#define ERROR(c)    regerr

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

DIAGNOSTICS

The function **compile( )** uses the macro **RETURN** on success and the macro **ERROR** on failure (see above). The functions **step( )** and **advance( )** return non-zero on a successful match and zero if there is no match. Errors are:

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.

SEE ALSO

regex(5)