NAME

pattern − build a regular expression pattern

SYNOPSIS

#include "mjsu.h"

CHAR *pattern(USHORT *pat, CHAR delim, CHAR *expr);

DESCRIPTION

pattern() builds an encoded pattern in the buffer starting at *pat*, suitable for use with **amatch()** or **match()** in matching characters.

The pattern is created from the "regular-expression" string *expr*; the creation of the pattern stops with the first unescaped occurrence in *expr* of the character *delim*. To prevent an ill-formed regular-expression from confounding the code, *expr* should be terminated with a NUL character, whether or not a *delim* other than NUL is given.

It is not necessary to know the encoding used, as encoded patterns should not be manipulated directly: their only use is as arguments to **amatch()** and **match()**. It is sufficient to know that the encoded pattern at *pat* will never occupy more than twice as many (**USHORT**) elements as the length of the string *expr*, up to and including the terminating occurrence of *delim*.

The encoded pattern is derived by interpreting the string at *expr* as a series of "regular expressions". A regular expression is a shorthand notation for a set of target strings to be searched for in a buffer. Any of these target strings are said to "match" the regular expression. The following regular expressions are allowed:

An ordinary character is considered a regular expression that matches that character. **8-bit characters are acceptable**.

The character sequences "\b", "\f", "\n", "\r", "\t", "\v", in upper or lower case, each match the single special characters representing backspace (BS), formfeed (FF), newline (NL), return (CR), tab (HT), and vertical tab (VT), respectively.

In addition, an arbitrary 8-bit character may be matched by "*ddd*" where *ddd* is the one to three digit octal value of the character code; this is the safest way to match most non-printing characters (eg: the ASCII "RS" character), and is the only way to match a NUL character, '\0'.

A '?' matches any single character except a newline.

A '^' as the leftmost character of a series of regular expressions constrains the match to begin at the start (left) of the target characters.

A '^' following another regular expression matches zero or more occurrences of that expression. A '^' may thus match a null string, which occurs at the beginning of a line, between pairs of characters, or at the end of the line. A '^' enclosed in "\(" and "\)", or following either a '\' or an initial '^', is taken as a literal '^', however.

A literal '^', as just defined, or a '^' in any position other than the ones just mentioned, matches a '^' from the target string.

A '*' matches zero or more characters, not including newline. It is conceptually identical to the sequence "?^".

A character string enclosed in square brackets "[]" matches a single character that may be *any* of the characters in the bracketed string, but no other. However, if the first character of the string is a '!', then the expression matches any character *except* the ones in the bracketed string and newline. Inside the bracketed string, a range of characters in the character collating sequence may be indicated by the three-character sequence <low-character>, '-', <high-character>. If <low-character> is greater than <high-character>, the expression is ignored. Thus, [a0-9b] is a regular expression that would (assuming ASCII encoding) match a single character which is either a decimal digit or the letter a or b. When matching a literal "-", the "-" must be the first or last character in the bracketed list, or must immediately follow a range, as in [0-9-+]; otherwise it is taken to specify a range of characters.

A regular expression enclosed between the sequences "\(" and "\)" tags this expression in a way detectable by the **amatch()** routine, but otherwise has no effect on the characters the expression matches.

A concatenation of regular expressions matches the concatenation of strings matched by individual regular expressions. In other words, a regular expression composed of a series of "sub-expressions" will match a concatenation of target character strings implied by each of the individual "sub-expressions".

A '$' as the rightmost character after a series of regular expressions constrains the match, if any, to end at the end of the target characters prior to the newline.

Note that arbitrary grouping and alternation are not fully supported by this notation, as the expressions accepted are not the full class of regular expressions beloved of mathematicians.

RETURNS

If no syntax errors were found in *expr*, pattern returns a pointer to the occurrence of *delim* that terminated its scan. Otherwise pattern returns NULL.

EXAMPLE

To put out only those lines that have a 'T' as their seventh character:

```
pattern(patbuf, '/', "^??????T/");
while (fgets(buf, BUFSIZ, pf) && match(buf, strlen(buf), patbuf))
        fputs(buf, stdout);
```

SEE ALSO

amatch(3), **match(3)**, **mjsu(7)**.

The masochists among you may gain pleasure by contemplating the character-set-dependent behaviour of **isalpha(), isdigit(), isprint(), strcoll()** and **setlocale()**, all as defined by ANSI standard X3.159-1989. See BUGS for the reason why.

AVAILABILITY

pattern() is written in C, conforming to ANSI X3.159-1989 (hosted program environment).

BUGS

The precise meaning of regular-expressions using ranges ("[]") or octal character-codes ("*ddd*"), is dependent on the character-set imposed by the execution environment. In the worst case, where the character-set of the execution environment is selectable by the user, the meaning of such regular-expressions could change from one invocation of the program to the next...!

For example, the expression "[a-Z]" matches any alphabetic character of the 7-bit ASCII character-set, but would exclude some accented (but still alphabetic) characters of the ISO-8859-1 character-set, and would have a completely different behaviour with an EBCDIC character-set. And some character-sets do not have *any* "alphabetic" characters at all...

Future revisions of the regular-expression language may allow pattern matching against specified "classes" of characters, tested for with the standard C functions **isalpha()**, **isprint()**, **isdigit()**, etc. This would allow *slightly* more character-set-independent expressions to be constructed, assuming that each of the target execution environments support an appropriate and fully-working "locale" to execute within.

But don't get your hopes up, because all "locale" support so far observed is either inappropriate or only partial, or both. I could tell you an amusing story about Sun's "iso_8859-1" locale and the way it makes proper operation of the standard **strcoll()** function impossible to even specify...

These inconveniences are not limited to **pattern()**: they apply to all known implementations of **grep, ex, vi, sh, csh, ksh, bash** and almost all other programs that do pattern-matching.

Unfortunately, but unavoidably, these inconveniences also apply to some of the specifications of the IEEE POSIX 1003.2 standard...

On a different but apparently similar theme, the term "newline" is technically inaccurate - the term "line-feed" is what is intended. This is only an issue if you are using **pattern()** to process material retrieved from files that are "binary" in the sense described by the C Language Standard (ANSI X3.159-1989). Better to avoid pattern-matching of "binary" data!