NAME

regex – match patterns against a string

SYNOPSIS

regex [**–e**] [**–v "***string***"**] [*pattern template*] ... *pattern* [*template*]

DESCRIPTION

The **regex** command takes a string from *the standard input*, and a list of *pattern* / *template* pairs, and runs **regex( )** to compare the string against each *pattern* until there is a match. When a match occurs, **regex** writes the corresponding *template* to *the standard output* and returns **TRUE**. The last (or only) *pattern* does not need a template. If that is the pattern that matches the string, the function simply returns **TRUE**. If no match is found, **regex** returns **FALSE**.

The argument *pattern* is a regular expression of the form described in **regex ( ).** In most cases *pattern* should be enclosed in single quotes to turn off special meanings of characters. Note that only the final *pattern* in the list may lack a *template .*

The argument *template* may contain the strings **$m0** through **$m9 ,** which will be expanded to the part of *pattern* enclosed in **(. . .)$0** through **(. . .)$9** constructs (see examples below). Note that if you use this feature, you must be sure to enclose *template* in single quotes so that FMLI does not expand **$m0** through **$m9** at parse time. This feature gives **regex** much of the power of **cut**(1), **paste**(1), and **grep**(1), and some of the capabilities of **sed**(1). If there is no *template,* the default is **$m0$m1$m2$m3$m4$m5$m6$m7$m8$m9 .**

OPTIONS

–e Evaluate the corresponding template and write the result to *the standard output.*

–v "*string*" Use *string* instead of *the standard input* to match against patterns.

EXAMPLES

To cut the 4th through 8th letters out of a string (this example will output **strin** and return **TRUE**):

```
`regex –v "my string is nice" ′^.{3}(.{5})$0′ ′$m0′`
```

In a form, to validate input to field 5 as an integer:

```
valid=`regex –v "$F5" ′^[0-9]+$′`
```

In a form, to translate an environment variable which contains one of the numbers **1**, **2**, **3**, **4**, **5** to the letters **a**, **b**, **c**, **d**, **e**:

```
value=`regex –v "$VAR1" 1 a 2 b 3 c 4 d 5 e ′.*′ ′Error′`
```

Note the use of the pattern **′.*′** to mean "anything else".

In the example below, all three lines constitute a single backquoted expression. This expression, by itself, could be put in a menu definition file. Since backquoted expressions are expanded as they are parsed, and output from a backquoted expression (the **cat** command, in this example) becomes part of the definition file being parsed, this expression would read **/etc/passwd** and make a dynamic menu of all the login ids on the system.

```
`cat /etc/passwd | regex ′^([^:]*)$0.*$′ ′
name=$m0
action=`message "$m0 is a user"`′`
```

DIAGNOSTICS

If none of the patterns match, **regex** returns **FALSE**, otherwise **TRUE**.

NOTES

Patterns and templates must often be enclosed in single quotes to turn off the special meanings of characters. Especially if you use the **$m0** through **$m9** variables in the template, since FMLI will expand the variables (usually to "") before **regex** even sees them.

Single characters in character classes (inside **[ ]**) must be listed before character ranges, otherwise they will not be recognized. For example, **[a-zA-Z_/]** will not find underscores (_) or slashes (**/**), but **[_/a-zA-Z]** will.

The regular expressions accepted by **regcmp** differ slightly from other utilities (that is, **sed**, **grep**, **awk**, **ed**, etc.).

regex with the −**e** option forces subsequent commands to be ignored. In other words if a backquoted statement appears as follows:

`regex -e ...; command1; command2`

command1 and *command2* would never be executed. However, dividing the expression into two:

`regex -e ...``command1; command2`

would yield the desired result.

SEE ALSO

awk(1), **cut**(1), **grep**(1), **paste**(1), **sed**(1), **regcmp**(3G)