

**NAME**

macros – description of macro facility

**DESCRIPTION**

The following commands are associated with the macro facility.

`lsdef` – List the names and sizes of all macros.

`def m s` – Define or redefine `m` to have the definition `s`, where `s` is a string. Assignment is made immediately.

`rdef m s` – Define or redefine `m` to have the definition `s`, where `s` is a string. Assignment is made when the enclosing statement block is executed.

`undef m1 [m2 ... ]` – Remove macro definitions.

`prdef [m1 ... ]` – Print all (or selected) macro definitions.

Within a macro, the symbols `$1`, `$2`, ... are replaced with the arguments with which the macro is invoked. Also,

- `$0` is replaced with the macro name,
- `$*` is replaced with all the arguments,
- `$#` is replaced with the number of arguments,
- `\$` is a literal `$`.

A macro argument is a string of characters separated by spaces. Use quotes to include more than one string in a single argument. Use `\"` or `\'`, in arguments to pass literal quotes to the macro. You can continue supplying arguments on subsequent lines by putting a backslash at the end of the previous line.

When a macro is defined without arguments, only the macro name is replaced with the definition.

When a macro is defined with arguments and is encountered by the program, all characters on that line up to a `;`, a `{` or the end of the line are eaten up, whether or not the macro uses them as arguments. When numbered arguments are referred to in the macro definition, but are missing when the macro is invoked, they are replaced with zeros. If `$*` is used in the definition and there are no arguments, no characters are substituted.

There is a limit to the length of a macro definition and to the number of arguments allowed. You will be informed if you exceed these limits. Each macro definition consumes approximately 20 bytes of dynamic memory. The macro definitions themselves are stored in a file.

**BUGS AND CAVEATS**

Beware of unwanted side effects when referencing the same argument more than once. For example,

```
def test 'a = $1; b = 2 * $1'
```

invoked as `test i++`, would be replaced with `a = i++; b = 2 * i++`, with the result that `i` is incremented twice, even though that action is not apparent to the user. The previous definition also would cause problems if invoked as `test 2+3`, as that would be replaced with `a = 2+3; b = 2 * 2+3`. The latter expression evaluates to 7, not 10, as might have been intended by the user. Use of parenthesis to surround arguments used in arithmetic expressions in macro definitions will avoid such problems, as in `b = 2 * ($1)`.

**EXAMPLES**

```
def u 'unix ( "$*" )'
```

Notice the double quotes convert the arguments into a single string, as required by the syntax of the `unix()` built-in command. When invoked with no arguments, the macro evaluates to

```
unix("").
def ct ' {
  local c
  if ((c = $1) == 0) tcount(1)
  else if (c < 0) mcount(-c)
  else tcount(c)
  waitcount; getcounts
  print S[0], S[1], S[2]
} '
```

Notice that the argument \$1 is referred to only once, by using the local variable `c`. When invoked with no arguments, the first line evaluates to `if ((c = 0) == 0)`.

```
def hscan '
  hklscan $1 $2 K K L L $3 $4
'
```

**SEE ALSO**

```
def undef lsdef
```