NAME

rc, cd, eval, exec, exit, flag, rfork, shift, wait, whatis, ., ~ – command language

SYNOPSIS

`rc` [`-srdiIlxepvV`] [`-c command`] [*file* [*arg* ...]]

DESCRIPTION

Rc is the Plan 9 shell. It executes command lines read from a terminal or a file or, with the `-c` flag, from *rc's* argument list.

Command Lines

A command line is a sequence of commands, separated by ampersands or semicolons (`&` or `;`), terminated by a newline. The commands are executed in sequence from left to right. *Rc* does not wait for a command followed by `&` to finish executing before starting the following command. Whenever a command followed by `&` is executed, its process id is assigned to the *rc* variable `$apid`. Whenever a command *not* followed by `&` exits or is terminated, the *rc* variable `$status` gets the process's wait message (see *wait*(3)); it will be the null string if the command was successful.

A long command line may be continued on subsequent lines by typing a backslash (`\`) followed by a newline. This sequence is treated as though it were a blank. Backslash is not otherwise a special character.

A number-sign (`#`) and any following characters up to (but not including) the next newline are ignored, except in quotation marks.

Simple Commands

A simple command is a sequence of arguments interspersed with I/O redirections. If the first argument is the name of an *rc* function or of one of *rc's* built-in commands, it is executed by *rc*. Otherwise if the name starts with a slash (`/`), it must be the path name of the program to be executed. Names containing no initial slash are searched for in a list of directory names stored in `$path`. The first executable file of the given name found in a directory in `$path` is the program to be executed. To be executable, the user must have execute permission (see *stat*(3)) and the file must be either an executable binary for the current machine's CPU type, or a shell script. Shell scripts begin with a line containing the full path name of a shell (usually `/bin/rc`), prefixed by `#!`.

The first word of a simple command cannot be a keyword unless it is quoted or otherwise disguised. The keywords are

```
for in while if not switch fn ~ ! @
```

Arguments and Variables

A number of constructions may be used where *rc's* syntax requires an argument to appear. In many cases a construction's value will be a list of arguments rather than a single string.

The simplest kind of argument is the unquoted word: a sequence of one or more characters none of which is a blank, tab, newline, or any of the following:

```
# ; & | ^ $ ` ' { } ( ) < >
```

An unquoted word that contains any of the characters `* ? [` is a pattern for matching against file names. The character `*` matches any sequence of characters, `?` matches any single character, and `[`*class*`]` matches any character in the *class*. If the first character of *class* is `~`, the class is complemented. The *class* may also contain pairs of characters separated by `-`, standing for all characters lexically between the two. The character `/` must appear explicitly in a pattern, as must the first character of the path name components `.` and `..`. A pattern is replaced by a list of arguments, one for each path name matched, except that a pattern matching no names is not replaced by the empty list, but rather stands for itself. Pattern matching is done after all other operations. Thus,

```
x=/tmp echo $x^/*.c
```

matches `/tmp/*.c`, rather than matching `/*.c` and then prefixing `/tmp`.

A quoted word is a sequence of characters surrounded by single quotes (`'`). A single quote is represented in a quoted word by a pair of quotes (`''`).

Each of the following is an argument.

`(`*arguments*`)`

The value of a sequence of arguments enclosed in parentheses is a list comprising the members of each element of the sequence. Argument lists have no recursive structure, although their syntax may suggest it. The following are entirely equivalent:

```
echo hi there everybody
((echo) (hi there) everybody)
```

`$`*argument*

`$`*argument*`(`*subscript*`)`

The *argument* after the `$` is the name of a variable whose value is substituted. Multiple levels of indirection are possible, but of questionable utility. Variable values are lists of strings. If *argument* is a number *n*, the value is the *n*th element of `$*`, unless `$*` doesn't have *n* elements, in which case the value is empty. If *argument* is followed by a parenthesized list of subscripts, the value substituted is a list composed of the requested elements (origin 1). The parenthesis must follow the variable name with no spaces. Subscripts can also take the form *m*`-`*n* or *m*`-` to indicate a sequence of elements. Assignments to variables are described below.

`$#`*argument*

The value is the number of elements in the named variable. A variable never assigned a value has zero elements.

$"*argument*

The value is a single string containing the components of the named variable separated by spaces. A variable with zero elements yields the empty string.

`` `{ ``*command*`}`

rc executes the *command* and reads its standard output, splitting it into a list of arguments, using characters in `$ifs` as separators. If `$ifs` is not otherwise set, its value is `' \t\n'`.

`<{`*command*`}`

`>{`*command*`}`

The *command* is executed asynchronously with its standard output or standard input connected to a pipe. The value of the argument is the name of a file referring to the other end of the pipe. This allows the construction of non-linear pipelines. For example, the following runs two commands `old` and `new` and uses `cmp` to compare their outputs

```
cmp <{old} <{new}
```

`<>{`*command*`}`

The *command* is executed asynchronously with its standard input and output each connected to a pipe. The value of the argument is a pair of file names referring to the two other ends of the pipes, in the order corresponding to the symbols `<` and `>` (first the pipe connected to the command's standard output, then the pipe connected to its standard input).

argument`^`*argument*

The `^` operator concatenates its two operands. If the two operands have the same number of components, they are concatenated pairwise. If not, then one operand must have one component, and the other must be non-empty, and concatenation is distributive.

Free Carets

Rc will insert the `^` operator automatically between words that are not separated by white space. Thus

```
cc -$flags $stem.c
```

is equivalent to

```
cc -^$flags $stem^.c
```

I/O Redirections

The sequence `>`*file* redirects the standard output file (file descriptor 1, normally the terminal) to the named *file*; `>>`*file* appends standard output to the file. The standard input file (file descriptor 0, also normally the terminal) may be redirected from a file by the sequence `<`*file*, or from an inline 'here document' by the sequence `<<`*eof-marker*. The contents of a here document are lines of text taken from the command input stream up to a line containing nothing but the *eof-marker*, which may be either a quoted or unquoted word. If *eof-marker* is unquoted, variable names of the form `$`*word* have their values substituted from *rc's* environment. If `$`*word* is followed by a caret (`^`), the caret is deleted. If *eof- marker* is quoted, no substitution occurs.

Redirections may be applied to a file-descriptor other than standard input or output by qualifying the redirection operator with a number in square brackets. For example, the diagnostic output (file descriptor 2) may be redirected by writing `cc junk.c >[2]junk`.

A file descriptor may be redirected to an already open descriptor by writing `>[`*fd0*`=`*fd1*`]` or `<[`*fd0*`=`*fd1*`]`. *Fd1* is a previously opened file descriptor and *fd0* becomes a new copy (in the sense of *dup*(3)) of it. A file descriptor may be closed by writing `>[`*fd0*`=]` or `<[`*fd0*`=]`.

Redirections are executed from left to right. Therefore, `cc junk.c >/dev/null >[2=1]` and `cc junk.c >[2=1] >/dev/null` have different effects: the first puts standard output in `/dev/null` and then puts diagnostic output in the same place, where the second directs diagnostic output to the terminal and sends standard output to `/dev/null`.

Compound Commands

A pair of commands separated by a pipe operator (`|`) is a command. The standard output of the left command is sent through a pipe to the standard input of the right command. The pipe operator may be decorated to use different file descriptors. `|[`*fd*`]` connects the output end of the pipe to file descriptor *fd* rather than 1. `|[`*fd0*`=`*fd1*`]` connects output to *fd1* of the left command and input to *fd0* of the right command.

A pair of commands separated by `&&` or `||` is a command. In either case, the left command is executed and its exit status examined. If the operator is `&&` the right command is executed if the left command's status is null. `||` causes the right command to be executed if the left command's status is non-null.

The exit status of a command may be inverted (non-null is changed to null, null is changed to non-null) by preceding it with a `!`.

The `|` operator has highest precedence, and is left-associative (i.e. binds tighter to the left than the right). `!` has intermediate precedence, and `&&` and `||` have the lowest precedence.

The unary `@` operator, with precedence equal to `!`, causes its operand to be executed in a subshell.

Each of the following is a command.

`if (` *list* `)` *command*

A *list* is a sequence of commands, separated by `&`, `;`, or newline. It is executed and if its exit status is null, the *command* is executed.

`if not` *command*

The immediately preceding command must have been `if(`*list*`)` *command*. If its condition was non-zero, the *command* is executed.

`for(`*name* `in` *arguments*`)` *command*

`for(`*name*`)` *command*

The *command* is executed once for each *argument* with that argument assigned to *name*. If the argument list is omitted, `$*` is used.

`while(`*list*`)` *command*

The *list* is executed repeatedly until its exit status is non-null. Each time it returns null status, the *command* is executed. An empty *list* is taken to give null status.

`switch(`*argument*`){`*list*`}`

The *list* is searched for simple commands beginning with the word `case`. (The search is only at the ‘top level’ of the *list*. That is, `cases` in nested constructs are not found.) *Argument* is matched against each word following `case` using the pattern-matching algorithm described above, except that `/` and the first characters of `.` and `..` need not be matched explicitly. When a match is found, commands in the list are executed up to the next following `case` command (at the top level) or the closing brace.

`{`*list*`}`

Braces serve to alter the grouping of commands implied by operator priorities. The *body* is a sequence of commands separated by `&`, `;`, or newline.

`fn` *name*`{`*list*`}`

`fn` *name*

The first form defines a function with the given *name*. Subsequently, whenever a command whose first argument is *name* is encountered, the current value of the remainder of the command’s argument list will be assigned to `$*`, after saving its current value, and *rc* will execute the *list*. The second form removes *name*’s function definition.

`fn` *note*`{`*list*`}`

`fn` *note*

A function with a special name will be called when *rc* receives a corresponding note; see *notify*(3). The valid note names (and corresponding notes) are `sighup` (`hangup`), `sigint` (`interrupt`), `sigalrm` (`alarm`), and `sigfpe` (floating point trap). By default *rc* exits on receiving any signal, except when run interactively, in which case interrupts and quits normally cause *rc* to stop whatever it’s doing and start reading a new command. The second form causes *rc* to handle a signal in the default manner. *Rc* recognizes an artificial note, `sigexit`, which occurs when *rc* is about to finish executing.

name`=`*argument command*

Any command may be preceded by a sequence of assignments interspersed with redirections. The assignments remain in effect until the end of the command, unless the command is empty (i.e. the assignments stand alone), in which case they are effective until rescinded by later assignments.

Built-in Commands

These commands are executed internally by *rc*, usually because their execution changes or depends on *rc*’s internal state.

`.` *file ...*

Execute commands from *file*. `$*` is set for the duration to the remainder of the argument list following *file*. *File* is searched for using `$path`.

`builtin` *command ...*

Execute *command* as usual except that any function named *command* is ignored in favor of the built-in meaning.

`cd [`*dir*`]`

Change the current directory to *dir*. The default argument is `$home`. *dir* is searched for in each of the directories mentioned in `$cdpath`.

`eval [`*arg ...*`]`

The arguments are concatenated separated by spaces into a single string, read as input to *rc*, and executed.

`exec [`*command ...*`]`

This instance of *rc* replaces itself with the given (non-built-in) *command*.

`flag` *f* `[+-]`

Either set (`+`), clear (`-`), or test (neither `+` nor `-`) the flag *f*, where *f* is a single character, one of the command line flags (see Invocation, below).

`exit [`*status*`]`

Exit with the given exit status. If none is given, the current value of `$status` is used.

`rfork [nNeEsfFm]`

Become a new process group using `rfork(`*flags*`)` where *flags* is composed of the bitwise OR of the `rfork` flags specified by the option letters (see *fork*(2)). If no *flags* are given, they default to `ens`. The *flags* and their meanings are: `n` is `RFNAMEG`; `N` is `RFCNAMEG`; `e` is `RFENVG`; `E` is `RFCENVG`; `s` is `RFNOTEG`; `f` is `RFFDG`; `F` is `RFCFDG`; and `m` is `RFNOMNT`.

`shift [`*n*`]`

Delete the first *n* (default 1) elements of `$*`.

`wait [`*pid*`]`

Wait for the process with the given *pid* to exit. If no *pid* is given, all outstanding processes are waited for.

`whatis` *name ...*

Print the value of each *name* in a form suitable for input to *rc*. The output is an assignment to any variable, the definition of any function, a call to `builtin` for any built-in command, or the completed pathname of any executable file.

`~` *subject pattern ...*

The *subject* is matched against each *pattern* in sequence. If it matches any pattern, `$status` is set to zero. Otherwise, `$status` is set to one. Patterns are the same as for file name matching, except that `/` and the first character of `.` and `..` need not be matched explicitly. The *patterns* are not subjected to file name matching before the `~` command is executed, so they need not be enclosed in quotation marks.

Environment

The *environment* is a list of strings made available to executing binaries by the kernel. *Rc* creates an environment entry for each variable whose value is non-empty, and for each function. The string for a variable entry has the variable's name followed by `=` and its value. If the value has more than one component, these are separated by SOH (001) characters. The string for a function is just the *rc* input that defines the function. The name of a function in the environment is the function name preceded by `fn#`.

When *rc* starts executing it reads variable and function definitions from its environment.

Special Variables

The following variables are set or used by *rc*.

`$*` Set to *rc*'s argument list during initialization. Whenever a `.` command or a function is executed, the current value is saved and `$*` receives the new argument list. The saved value is restored on completion of the `.` or function.

`$apid` Whenever a process is started asynchronously with `&`, `$apid` is set to its process id.

`$home` The default directory for `cd`. Defaults to `$HOME` or else `/`.

`$ifs` The input field separators used in backquote substitutions. If `$ifs` is not set in *rc*'s environment, it is initialized to blank, tab and newline.

`$path` The search path used to find commands and input files for the `.` command. If not set in the environment, it is initialized by parsing the `$PATH` variable (as in *sh*(1)) or by `path=(. /bin)`. The variables `$path` and `$PATH` are maintained together: changes to one will be reflected in the other.

`$pid` Set during initialization to *rc*'s process id.

`$prompt` When *rc* is run interactively, the first component of `$prompt` is printed before reading each command. The second component is printed whenever a newline is typed and more lines are required to complete the command. If not set in the environment, it is initialized by `prompt=('% ' ' ')`.

`$status` Set to the wait message of the last-executed program. (unless started with `&). !` and `~` also change `$status`. Its value is used to control execution in `&&`, `||`, `if` and `while` commands. When *rc* exits at end-of-file of its input or on executing an `exit` command with no argument, `$status` is its exit status.

Invocation

If *rc* is started with no arguments it reads commands from standard input. Otherwise its first non-flag argument is the name of a file from which to read commands (but see `-c` below). Subsequent arguments become the initial value of `$*`. *Rc* accepts the following command-line flags.

`-c` *string* Commands are read from *string*.

`-s` Print out exit status after any command where the status is non-null.

`-e` Exit if `$status` is non-null after executing a simple command.

`-i` If `-i` is present, or *rc* is given no arguments and its standard input is a terminal, it runs interactively. Commands are prompted for using `$prompt`.

`-I` Makes sure *rc* is not run interactively.

`-l` If `-l` is given or the first character of argument zero is `-`, *rc* reads commands from `$home/lib/profile`, if it exists, before reading its normal input.

`-p` A no-op.

`-d` A no-op.

`-v` Echo input on file descriptor 2 as it is read.

`-x` Print each simple command before executing it.

`-r` Print debugging information (internal form of commands as they are executed).

SOURCE

`/usr/local/plan9/src/cmd/rc`

SEE ALSO

Tom Duff, "Rc – The Plan 9 Shell".

BUGS

There should be a way to match patterns against whole lists rather than just single strings.

Using `~` to check the value of `$status` changes `$status`.

Functions that use here documents don't work.

The `<{`*command*`}` syntax depends on the underlying operating system providing a file descriptor device tree at `/dev/fd`.

Some FreeBSD installations does not provide file descriptors greater than 2 in `/dev/fd`. To fix this, add

```
  /fdescfs     /dev/fd     fdescfs     rw     0     0
```

to `/etc/fstab`, and then `mount /dev/fd`. (Adding the line to `fstab` ensures causes FreeBSD to mount the file system automatically at boot time.)

Some systems require `/usr/local/plan9/bin/rc` to be listed in `/etc/shells` before it can be used as a login shell.

Space Glenda