Rc — A Shell for Plan 9 and UNIX Systems

Tom Duff

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Rc is a command interpreter for Plan 9. It also runs on a variety of traditional systems, including SunOS and the Tenth Edition. It provides similar facilities to Bourne's */bin/sh*, with some small additions and mostly less idiosyncratic syntax. This paper introduces *rc*'s highlights with numerous examples, and discusses its design and why it varies from Bourne's.

1. Introduction

Plan 9 needs a command-programming language. As porting the Bourne shell to an incompatible new environment seemed a daunting task, I chose to write a new command interpreter, called *rc* because it runs commands. Although tinkering with perfection is a dangerous business, I could hardly resist trying to 'improve' on Bourne's design. Thus *rc* is similar in spirit but different in detail from Bourne's shell.

The bulk of this paper describes *rc*'s principal features with many small examples and a few larger ones. We close with a discussion of the principles guiding *rc*'s design and why it differs from Bourne's design. The descriptive sections include little discussion of the rationale for particular features, as individual details are hard to justify in isolation. The impatient reader may wish to skip to the discussion at the end before skimming the expository parts of the paper.

2. Simple commands

For the simplest uses *rc* has syntax familiar to Bourne-shell users. Thus all of the following behave as expected:

```
date
con alice
who >user.names
who >>user.names
wc <file
echo [a-f]*.c
who | wc
who; date
cc *.c &
cyntax *.c && cc -g -o cmd *.c
rm -r junk || echo rm failed!
```

3. Quotation

An argument that contains a space or one of *rc*'s other syntax characters must be enclosed in apostrophes (').

```
rm 'odd file name'
```

An apostrophe in a quoted argument must be doubled:

```
echo 'How''s your father?'
```

4. Variables

Rc provides variables whose values are lists of arguments. Variables may be given values by typing, for example:

```
path=(. /bin /usr/bin)
user=td
tty=/dev/tty8
```

The parentheses indicate that the value assigned to `path` is a list of three strings. The variables `user` and `tty` are assigned lists containing a single string.

The value of a variable can be substituted into a command by preceding its name with a `$`, like this:

```
echo $path
```

If `path` had been set as above, this would be equivalent to

```
echo . /bin /usr/bin
```

Variables may be subscripted by numbers or lists of numbers, like this:

```
echo $path(2)
echo $path(3 2 1)
```

These are equivalent to

```
echo /bin
echo /usr/bin /bin .
```

There can be no space separating the variable's name from the left parenthesis. Otherwise, the subscript would be considered a separate parenthesized list.

The number of strings in a variable can be determined by the `$#` operator. For example,

```
echo $#path
```

would print the number of entries in `$path`.

The following two assignments are subtly different:

```
empty=()
null=''
```

The first sets `empty` to a list containing no strings. The second sets `null` to a list containing a single string, but the string contains no characters.

Although these may seem like more or less the same thing (in Bourne's shell, they are indistinguishable), they behave differently in almost all circumstances. Among other things

```
echo $#empty
```

prints 0, whereas

```
echo $#null
```

prints 1.

All variables that have never been set have the value `()`.

5. Arguments

When *rc* is reading its input from a file, the file has access to the arguments supplied on *rc*'s command line. The variable `$*` initially has the list of arguments assigned to it. The names `$1`, `$2`, etc. are synonyms for `$*(1)`, `$*(2)`, etc. In addition, `$0` is the name of the file from which *rc*'s input is being read.

6. Concatenation

Rc has a string concatenation operator, the caret ^, to build arguments out of pieces.

```
echo hully^gully
```

is exactly equivalent to

```
echo hullygully
```

Suppose variable `i` contains the name of a command. Then

```
cc -o $i $i^.c
```

might compile the command's source code, leaving the result in the appropriate file.

Concatenation distributes over lists. The following

```
echo (a b c)^(1 2 3)
src=(main subr io)
cc $src^.c
```

are equivalent to

```
echo a1 b2 c3
cc main.c subr.c io.c
```

In detail, the rule is: if both operands of ^ are lists of the same non-zero number of strings, they are concatenated pairwise. Otherwise, if one of the operands is a single string, it is concatenated with each member of the other operand in turn. Any other combination of operands is an error.

7. Free carets

User demand has dictated that *rc* insert carets in certain places, to make the syntax look more like the Bourne shell. For example, this:

```
cc -$flags $stems.c
```

is equivalent to

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

In general, *rc* will insert ^ between two arguments that are not separated by white space. Specifically, whenever one of `$'` follows a quoted or unquoted word, or an unquoted word follows a quoted word with no intervening blanks or tabs, a ^ is inserted between the two. If an unquoted word immediately following a `$` contains a character other than an alphanumeric, underscore or `*`, a ^ is inserted before the first such character.

8. Command substitution

It is often useful to build an argument list from the output of a command. *Rc* allows a command, enclosed in braces and preceded by a left quote, `'{...}`, anywhere that an argument is required. The command is executed and its standard output captured. The characters stored in the variable `ifs` are used to split the output into arguments. For example,

```
cat '{ls -tr|sed 10q}
```

will catenate the ten oldest files in the current directory in temporal order.

9. Pipeline branching

The normal pipeline notation is general enough for almost all cases. Very occasionally it is useful to have pipelines that are not linear. Pipeline topologies more general than trees can require arbitrarily large pipe buffers, or worse, can cause deadlock. *Rc* has syntax for some kinds of non-linear but treelike pipelines. For example,

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

will regression test a new version of a command. < or > followed by a command in braces causes the command to be run with its standard output or input attached to a pipe. The parent command (`cmp` in the example) is started with the other end of the pipe attached to some file descriptor or other, and with an argument that will connect to the pipe when opened (e.g. `/dev/fd/6`.) On systems without `/dev/fd` or something similar (SunOS for example) this feature does not work.

10. Exit status

When a command exits it returns status to the program that executed it. On Plan 9 status is a character string describing an error condition. On normal termination it is empty.

Rc captures commands' exit statuses in the variable `$status`. For a simple command the value of `$status` is just as described above. For a pipeline `$status` is set to the concatenation of the statuses of the pipeline components with `|` characters for separators.

Rc has a several kinds of control flow, many of them conditioned by the status returned from previously executed commands. Any `$status` containing only `0`'s and `|`'s has boolean value *true*. Any other status is *false*.

11. Command grouping

A sequence of commands enclosed in `{}` may be used anywhere a command is required. For example:

```
{sleep 3600;echo 'Time''s up!'}&
```

will wait an hour in the background, then print a message. Without the braces:

```
sleep 3600;echo 'Time''s up!'&
```

this would lock up the terminal for an hour, then print the message in the background!

12. Control flow — `for`

A command may be executed once for each member of a list by typing, for example:

```
for(i in printf scanf putchar) look $i /usr/td/lib/dw.dat
```

This looks for each of the words `printf`, `scanf` and `putchar` in the given file. The general form is

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

or

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

In the first case *command* is executed once for each member of *list* with that member assigned to variable *name*. If `in` *list* is not given, *$** is used.

13. Conditional execution — `if`

Rc also provides a general if-statement. For example:

```
if(cyntax *.c) cc -g -o cmd *.c
```

runs the C compiler whenever `cyntax` finds no problems with `*.c`. An 'if not' statement provides a two-tailed conditional. For example:

```
for(i){
    if(test -f /tmp/$i) echo $i already in /tmp
    if not cp $i /tmp
}
```

This loops over each file in `$*`, copying to `/tmp` those that do not already appear there, and printing a message for those that do.

14. Control flow — `while`

Rc's while statement looks like this:

```
while(newer subr.c subr.o) sleep 5
```

This waits until `subr.o` is newer than `subr.c` (presumably because the C compiler finished with it.)

15. Control flow — `switch`

Rc provides a switch statement to do pattern-matching on arbitrary strings. Its general form is

```
switch(word){
case pattern ...
    commands
case pattern ...
    commands
...
}
```

Rc attempts to match the word against the patterns in each case statement in turn. Patterns are the same as for filename matching, except that `/` and the first characters of `.` and `..` need not be matched explicitly.

If any pattern matches, the commands following that case up to the next case (or the end of the switch) are executed, and execution of the switch is complete. For example,

```
switch($#*){
case 1
    cat >>$1
case 2
    cat >>$2 <$1
case *
    echo 'Usage: append [from] to'
}
```

is an append command. Called with one file argument, it tacks standard input to its end. With two, the first is appended to the second. Any other number elicits a usage message.

The built-in `~` command also matches patterns, and is often more concise than a switch. Its arguments are a string and a list of patterns. It sets `$status` to true if and only if any of the patterns matches the string. The following example processes option arguments for the *man*(1) command:

```
opt=()
while(~ $1 -* [1-9] 10){
    switch($1){
    case [1-9] 10
        sec=$1 secn=$1
    case -f
        c=f s=f
    case -[qwnt]
        cmd=$1
    case -T*
        T=$1
    case -*
        opt=($opt $1)
    }
    shift
}
```