Rc — The Plan 9 Shell

Tom Duff
td@plan9.bell-labs.com

ABSTRACT

Rc is a command interpreter for Plan 9 that provides similar facilities to UNIX's Bourne shell, with some small additions and less idiosyncratic syntax. This paper uses numerous examples to describe *rc*'s features, and contrasts *rc* with the Bourne shell, a model that many readers will be familiar with.

1. Introduction

Rc is similar in spirit but different in detail from UNIX's Bourne shell. This paper describes *rc*'s principal features with many small examples and a few larger ones. It assumes familiarity with the Bourne shell.

2. Simple commands

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

```
date
cat /lib/news/build
who >user.names
who >>user.names
wc <file
echo [a-f]*.c
who | wc
who; date
vc *.c &
mk && v.out /*/bin/fb/*
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. Patterns

An unquoted argument that contains any of the characters `*` `?` `[` is a pattern to be matched against file names. A `*` character matches any sequence of characters, `?` matches any single character, and `[`*class*`]` matches any character in the `class`, unless the first character of *class* is `~`, in which case 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 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; rather it stands for itself.

5. Variables

UNIX's Bourne shell offers string-valued variables. *Rc* provides variables whose values are lists of arguments — that is, arrays of strings. This is the principal difference between *rc* and traditional UNIX command interpreters. Variables may be given values by typing, for example:

```
path=(. /bin)
user=td
font=/lib/font/bit/pelm/ascii.9.font
```

The parentheses indicate that the value assigned to `path` is a list of two strings. The variables `user` and `font` 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
```

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

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

These are equivalent to

```
echo /bin
echo /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 2 for this example.

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 `()`.

Occasionally, it is convenient to treat a variable's value as a single string. The elements of a string are concatenated into a single string, with spaces between the elements, by the `$"` operator. Thus, if we set

```
list=(How now brown cow)
string=$"list
```

then both

```
echo $list
```

and

```
echo $string
```

cause the same output, viz:

```
How now brown cow
```

but

```
echo $#list $#string
```

will output

```
4 1
```

because `$list` has four members, but `$string` has a single member, with three spaces separating its words.

6. 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.

7. 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

```
vc $i^.c
vl -o $1 $i^.v
```

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.

8. 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, an implicit ^ 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.

9. 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 concatenate the ten oldest files in the current directory in temporal order, given the default `ifs` setting of space, tab, and newline.

10. 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`). Some commands are unprepared to deal with input files that turn out not to be seekable. For example `diff` needs to read its input twice.

11. 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 command exit status 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*.

12. 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!'&
```

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

13. 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 the clause "`in` *list*" is missing, "`in $*`" is assumed.

14. Conditional execution — `if`

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

```
for(i in *.c) if(cpp $i >/tmp/$i) vc /tmp/$i
```

runs the C compiler on each C source program that cpp processes without error. 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.

15. Control flow — `while`

Rc's while statement looks like this:

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

This waits until `subr.v` is newer than `subr.c`, presumably because the C compiler finished with it.

If the controlling command is empty, the loop will not terminate. Thus,

```
while() echo y
```

emulates the *yes* command.

16. 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 `.` 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 appends its standard input to the named file. With two, the first is appended to the second. Any other number elicits an error 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
}
```

17. Functions

Functions may be defined by typing

```
fn name { commands }
```

Subsequently, whenever a command named *name* is encountered, the remainder of the command's argument list will assigned to `$*` and *rc* will execute the *commands*. The value of `$*` will be restored on completion. For example:

```
fn g {
    grep $1 *.[hcyl]
}
```

defines `g` *pattern* to look for occurrences of *pattern* in all program source files in the current directory.

Function definitions are deleted by writing

`fn` *name*

with no function body.

18. Command execution

Rc does one of several things to execute a simple command. If the command name is the name of a function defined using `fn`, the function is executed. Otherwise, if it is the name of a built-in command, the built-in is executed directly by *rc*. Otherwise, directories mentioned in the variable `$path` are searched until an executable file is found. Extensive use of the `$path` variable is discouraged in Plan 9. Instead, use the default `(. /bin)` and bind what you need into `/bin`.

19. Built-in commands

Several commands are executed internally by *rc* because they are difficult to implement otherwise.

`. [-i]` *file ...*
: Execute commands from *file*. `$*` is set for the duration to the reminder of the argument list following *file*. `$path` is used to search for *file*. Option `-i` indicates interactive input — a prompt (found in `$prompt`) is printed before each command is read.

`builtin` *command ...*
: Execute *command* as usual except that any function named *command* is ignored. For example,

```
fn cd{
    builtin cd $* && pwd
}
```

defines a replacement for the `cd` built-in (see below) that announces the full name of the new directory.

`cd [`*dir*`]`
: Change the current directory to *dir*. The default argument is `$home`. `$cdpath` is a list of places in which to search for *dir*.

`eval [`*arg ...*`]`
: The arguments are concatenated (separated by spaces) into a string, read as input to *rc*, and executed. For example,

```
x='$y'
y=Doody
eval echo Howdy, $x
```

would echo

```
Howdy, Doody
```

since the arguments of `eval` would be

```
echo Howdy, $y
```

after substituting for `$x`.

`exec [`*command ...*`]`
: *Rc* replaces itself with the given *command*. This is like a *goto* — *rc* does not wait for the command to exit, and does not return to read any more commands.

`exit [`*status*`]`
: *Rc* exits immediately with the given status. If none is given, the current value of `$status` is used.

`flag` *f* `[+-]`
: This command manipulates and tests the command line flags (described below).

```
flag f +
```

sets flag *f*.

```
flag f -
```

clears flag *f*.

```
flag f
```

tests flag *f*, setting `$status` appropriately. Thus

```
if(flag x) flag v +
```

sets the `-v` flag if the `-x` flag is already set.

`rfork [nNeEsfF]`
: This uses the Plan 9 *rfork* system entry to put *rc* into a new process group with the following attributes:

Flag	Name	Function
`n`	RFNAMEG	Make a copy of the parent's name space
`N`	RFCNAMEG	Start with a new, empty name space
`e`	RFENVG	Make a copy of the parent's environment
`E`	RFCENVG	Start with a new, empty environment
`s`	RFNOTEG	Make a new note group
`f`	RFFDG	Make a copy of the parent's file descriptor space
`F`	RFCFDG	Make a new, empty file descriptor space

Section *fork*(2) of the Programmer's Manual describes these attributes in more detail.

`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 a variable, the definition of a function, a call to `builtin` for a built-in command, or the path name of a binary program. For example,

```
whatis path g cd who
```

might print

```
path=(. /bin)
fn g {gre -e $1 *.[hycl]}
builtin cd
/bin/who
```

~ *subject pattern ...*

The *subject* is matched against each *pattern* in turn. On a match, `$status` is set to true. Otherwise, it is set to `'no match'`. Patterns are the same as for filename matching. The *patterns* are not subjected to filename replacement before the ~ command is executed, so they need not be enclosed in quotation marks, unless of course, a literal match for `*` `[` or `?` is required. For example

```
~ $1 ?
```

matches any single character, whereas

```
~ $1 '?'
```

only matches a literal question mark.

20. Advanced I/O Redirection

Rc allows redirection of file descriptors other than 0 and 1 (standard input and output) by specifying the file descriptor in square brackets `[ ]` after the < or >. For example,

```
vc junk.c >[2]junk.diag
```

saves the compiler's diagnostics from standard error in `junk.diag`.

File descriptors may be replaced by a copy, in the sense of *dup*(2), of an already-open file by typing, for example

```
vc junk.c >[2=1]
```

This replaces file descriptor 2 with a copy of file descriptor 1. It is more useful in conjunction with other redirections, like this

```
vc junk.c >junk.out >[2=1]
```

Redirections are evaluated from left to right, so this redirects file descriptor 1 to `junk.out`, then points file descriptor 2 at the same file. By contrast,

```
vc junk.c >[2=1] >junk.out
```

redirects file descriptor 2 to a copy of file descriptor 1 (presumably the terminal), and then directs file descriptor 1 to a file. In the first case, standard and diagnostic output will be intermixed in `junk.out`. In the second, diagnostic output will appear on the terminal, and standard output will be sent to the file.

File descriptors may be closed by using the duplication notation with an empty right-hand side. For example,

```
vc junk.c >[2=]
```

will discard diagnostics from the compilation.

Arbitrary file descriptors may be sent through a pipe by typing, for example,

```
vc junk.c |[2] grep -v '^$'
```

This deletes blank lines from the C compiler's error output. Note that the output of `grep` still appears on file descriptor 1.

Occasionally you may wish to connect the input side of a pipe to some file descriptor other than zero. The notation

```
cmd1 |[5=19] cmd2
```

creates a pipeline with `cmd1`'s file descriptor 5 connected through a pipe to `cmd2`'s file descriptor 19.

21. Here documents

Rc procedures may include data, called ‘‘here documents’’, to be provided as input to commands, as in this version of the *tel* command

```
for(i) grep $i <<!
...
tor 2T−402 2912
kevin 2C−514 2842
bill 2C−562 7214
...
!
```

A here document is introduced by the redirection symbol <<, followed by an arbitrary EOF marker (`!` in the example). Lines following the command, up to a line containing only the EOF marker are saved in a temporary file that is connected to the command's standard input when it is run.

Rc does variable substitution in here documents. The following command:

```
ed $3 <<EOF
g/$1/s//$2/g
w
EOF
```

changes all occurrences of `$1` to `$2` in file `$3`. To include a literal `$` in a here document, type `$$`. If the name of a variable is followed immediately by `^`, the caret is deleted.

Variable substitution can be entirely suppressed by enclosing the EOF marker following << in quotation marks, as in `<<'EOF'`.

Here documents may be provided on file descriptors other than 0 by typing, for example,

```
cmd <<[4]End
...
End
```

If a here document appears within a compound block, the contents of the document must be after the whole block:

```
for(i in $*){
        mail $i <<EOF
}
words to live by
EOF
```

22. Catching Notes

Rc scripts normally terminate when an interrupt is received from the terminal. A function with the name of a UNIX signal, in lower case, is defined in the usual way, but called when *rc* receives the corresponding note. The *notify*(2) section of the Programmer's Manual discusses notes in some detail. Notes of interest are:

`sighup`
: The note was ‘hangup’. Plan 9 sends this when the terminal has disconnected from *rc*.

`sigint`
The note was 'interrupt', usually sent when the interrupt character (ASCII DEL) is typed on the terminal.

`sigterm`
The note was 'kill', normally sent by *kill*(1).

`sigexit`
An artificial note sent when *rc* is about to exit.

As an example,

```
fn sigint{
        rm /tmp/junk
        exit
}
```

sets a trap for the keyboard interrupt that removes a temporary file before exiting.

Notes will be ignored if the note routine is set to `{}`. Signals revert to their default behavior when their handlers' definitions are deleted.

23. Environment

The environment is a list of name–value pairs made available to executing binaries. On Plan 9, the environment is stored in a file system named `#e`, normally mounted on `/env`. The value of each variable is stored in a separate file, with components terminated by zero bytes. (The file system is maintained entirely in core, so no disk or network access is involved.) The contents of `/env` are shared on a per-process group basis – when a new process group is created it effectively attaches `/env` to a new file system initialized with a copy of the old one. A consequence of this organization is that commands can change environment entries and see the changes reflected in *rc*.

Functions also appear in the environment, named by prefixing `fn#` to their names, like `/env/fn#roff`.

24. Local Variables

It is often useful to set a variable for the duration of a single command. An assignment followed by a command has this effect. For example

```
a=global
a=local echo $a
echo $a
```

will print

```
local
global
```

This works even for compound commands, like

```
f=/fairly/long/file/name {
        { wc $f; spell $f; diff $f.old $f } |
          pr -h 'Facts about '$f | lp -dfn
}
```

25. Examples — *cd, pwd*

Here is a pair of functions that provide enhanced versions of the standard `cd` and `pwd` commands. (Thanks to Rob Pike for these.)

```
ps1='% '        # default prompt
tab='	'        # a tab character
fn cd{
  builtin cd $1 &&
  switch($#*){
  case 0
    dir=$home
    prompt=($ps1 $tab)
  case *
    switch($1)
    case /*
      dir=$1
      prompt=(`{basename `{pwd}}^$ps1 $tab)
    case */* ..*
      dir=()
      prompt=(`{basename `{pwd}}^$ps1 $tab)
    case *
      dir=()
      prompt=($1^$ps1 $tab)
    }
  }
}
fn pwd{
  if(~ $#dir 0)
    dir=`{/bin/pwd}
  echo $dir
}
```

Function `pwd` is a version of the standard `pwd` that caches its value in variable `$dir`, because the genuine `pwd` can be quite slow to execute. (Recent versions of Plan 9 have very fast implementations of `pwd`, reducing the advantage of the `pwd` function.)

Function `cd` calls the `cd` built-in, and checks that it was successful. If so, it sets `$dir` and `$prompt`. The prompt will include the last component of the current directory (except in the home directory, where it will be null), and `$dir` will be reset either to the correct value or to `()`, so that the `pwd` function will work correctly.

26. Examples — *man*

The *man* command prints pages of the Programmer's Manual. It is called, for example, as

```
man 2 sinh
man rc
man -t cat
```

In the first case, the page for *sinh* in section 2 is printed. In the second case, the manual page for *rc* is printed. Since no manual section is specified, all sections are searched for the page, and it is found in section 1. In the third case, the page for *cat* is typeset (the `-t` option).

```
cd /sys/man || {
  echo $0: No manual! >[1=2]
  exit 1
}
NT=n  # default nroff
s='*' # section, default try all
for(i) switch($i){
case -t
  NT=t
case -n
  NT=n
case -*
  echo Usage: $0 '[-nt] [section] page ...' >[1=2]
  exit 1
case [1-9] 10
  s=$i
case *
  eval 'pages='$s/$i
  for(page in $pages){
    if(test -f $page)
      $NT^roff -man $page
    if not
      echo $0: $i not found >[1=2]
  }
}
```

Note the use of `eval` to make a list of candidate manual pages. Without `eval`, the `*` stored in `$s` would not trigger filename matching — it's enclosed in quotation marks, and even if it weren't, it would be expanded when assigned to `$s`. Eval causes its arguments to be re-processed by *rc*'s parser and interpreter, effectively delaying evaluation of the `*` until the assignment to `$pages`.

27. Examples — *holmdel*

The following *rc* script plays the deceptively simple game *holmdel*, in which the players alternately name Bell Labs locations, the winner being the first to mention Holmdel.

This script is worth describing in detail (rather, it would be if it weren't so silly.)

Variable `$t` is an abbreviation for the name of a temporary file. Including `$pid`, initialized by *rc* to its process-id, in the names of temporary files insures that their names won't collide, in case more than one instance of the script is running at a time.

Function `read`'s argument is the name of a variable into which a line gathered from standard input is read. `$ifs` is set to just a newline. Thus `read`'s input is not split apart at spaces, but the terminating newline is deleted.

A handler is set to catch `sigint`, `sigquit`, and `sighup`, and the artificial `sigexit` signal. It just removes the temporary file and exits.

The temporary file is initialized from a here document containing a list of Bell Labs locations, and the main loop starts.

First, the program guesses a location (in `$lab`) using the `fortune` program to pick a random line from the location list. It prints the location, and if it guessed Holmdel, prints a message and exits.

Then it uses the `read` function to get lines from standard input and validity-check them until it gets a legal name. Note that the condition part of a `while` can be a compound command. Only the exit status of the last command in the

```
t=/tmp/holmdel$pid
fn read{
        $1=`{awk '{print;exit}'}
}
ifs='
'       # just a newline
fn sigexit sigint sigquit sighup{
        rm -f $t
        exit
}
cat <<'!' >$t
Allentown
Atlanta
Cedar Crest
Chester
Columbus
Elmhurst
Fullerton
Holmdel
Indian Hill
Merrimack Valley
Morristown
Neptune
Piscataway
Reading
Short Hills
South Plainfield
Summit
Whippany
West Long Branch
!
while(){
   lab=`{fortune $t}
   echo $lab
   if(~ $lab Holmdel){
      echo You lose.
      exit
   }
   while(read lab; ! grep -i -s $lab $t) echo No such location.
   if(~ $lab [hH]olmdel){
      echo You win.
      exit
   }
}
```

sequence is checked.

Again, if the result is Holmdel, it prints a message and exits. Otherwise it goes back to the top of the loop.

28. Design Principles

Rc draws heavily from Steve Bourne's `/bin/sh`. Any successor of the Bourne shell is bound to suffer in comparison. I have tried to fix its best-acknowledged shortcomings and to simplify things wherever possible, usually by omitting inessential features. Only when irresistibly tempted have I introduced novel ideas. Obviously I have tinkered extensively with Bourne's syntax.

The most important principle in *rc*'s design is that it's not a macro processor. Input is never scanned more than once by the lexical and syntactic analysis code (except, of course, by the `eval` command, whose *raison d'être* is to break the rule).

Bourne shell scripts can often be made to run wild by passing them arguments containing spaces. These will be split into multiple arguments using `IFS`, often at inopportune times. In *rc*, values of variables, including command line arguments, are not re-read when substituted into a command. Arguments have presumably been scanned in the parent process, and ought not to be re-read.

Why does Bourne re-scan commands after variable substitution? He needs to be able to store lists of arguments in variables whose values are character strings. If we eliminate re-scanning, we must change the type of variables, so that they can explicitly carry lists of strings.

This introduces some conceptual complications. We need a notation for lists of words. There are two different kinds of concatenation, for strings — `$a^$b`, and lists — `($a $b)`. The difference between `()` and `''` is confusing to novices, although the distinction is arguably sensible — a null argument is not the same as no argument.

Bourne also rescans input when doing command substitution. This is because the text enclosed in back-quotes is not a string, but a command. Properly, it ought to be parsed when the enclosing command is, but this makes it difficult to handle nested command substitutions, like this:

```
size=`wc -l \`ls -t|sed 1q\``
```

The inner back-quotes must be escaped to avoid terminating the outer command. This can get much worse than the above example; the number of `\`'s required is exponential in the nesting depth. *Rc* fixes this by making the backquote a unary operator whose argument is a command, like this:

```
size=`{wc -l `{ls -t|sed 1q}}
```

No escapes are ever required, and the whole thing is parsed in one pass.

For similar reasons *rc* defines signal handlers as though they were functions, instead of associating a string with each signal, as Bourne does, with the attendant possibility of getting a syntax error message in response to typing the interrupt character. Since *rc* parses input when typed, it reports errors when you make them.

For all this trouble, we gain substantial semantic simplifications. There is no need for the distinction between `$*` and `$@`. There is no need for four types of quotation, nor the extremely complicated rules that govern them. In *rc* you use quotation marks when you want a syntax character to appear in an argument, or an argument that is the empty string, and at no other time. `IFS` is no longer used, except in the one case where it was indispensable: converting command output into argument lists during command substitution.

This also avoids an important UNIX security hole. In UNIX, the *system* and *popen* functions call `/bin/sh` to execute a command. It is impossible to use either of these routines with any assurance that the specified command will be executed, even if the caller of *system* or *popen* specifies a full path name for the command. This can be devastating if it occurs in a set-userid program. The problem is that `IFS` is used to split the command into words, so an attacker can just set `IFS=/` in his environment and leave a Trojan horse named `usr` or `bin` in the current working directory before running the privileged program. *Rc* fixes this by never rescanning input for any reason.

Most of the other differences between *rc* and the Bourne shell are not so serious. I eliminated Bourne's peculiar forms of variable substitution, like

```
echo ${a=b} ${c-d} ${e?error}
```

because they are little used, redundant and easily expressed in less abstruse terms. I deleted the builtins `export`, `readonly`, `break`, `continue`, `read`, `return`, `set`, `times` and `unset` because they seem redundant or only marginally useful.

Where Bourne's syntax draws from Algol 68, *rc*'s is based on C or Awk. This is harder to defend. I believe that, for example

```
if(test -f junk) rm junk
```

is better syntax than

```
if test -f junk; then rm junk; fi
```

because it is less cluttered with keywords, it avoids the semicolons that Bourne requires in odd places, and the syntax characters better set off the active parts of the command.

The one bit of large-scale syntax that Bourne unquestionably does better than *rc* is the `if` statement with `else` clause. *Rc*'s `if` has no terminating `fi`-like bracket. As a result, the parser cannot tell whether or not to expect an `else` clause without looking ahead in its input. The problem is that after reading, for example

```
if(test -f junk) echo junk found
```

in interactive mode, *rc* cannot decide whether to execute it immediately and print `$prompt(1)`, or to print `$prompt(2)` and wait for the `else` to be typed. In the Bourne shell, this is not a problem, because the `if` command must end with `fi`, regardless of whether it contains an `else` or not.

Rc's admittedly feeble solution is to declare that the `else` clause is a separate statement, with the semantic proviso that it must immediately follow an `if`, and to call it `if not` rather than `else`, as a reminder that something odd is going on. The only noticeable consequence of this is that the braces are required in the construction

```
for(i){
    if(test -f $i) echo $i found
    if not echo $i not found
}
```

and that *rc* resolves the "dangling else" ambiguity in opposition to most people's expectations.

It is remarkable that in the four most recent editions of the UNIX system programmer's manual the Bourne shell grammar described in the manual page does not admit the command `who|wc`. This is surely an oversight, but it suggests something darker: nobody really knows what the Bourne shell's grammar is. Even examination of the source code is little help. The parser is implemented by recursive descent, but the routines corresponding to the syntactic categories all have a flag argument that subtly changes their operation depending on the context. *Rc*'s parser is implemented using *yacc*, so I can say precisely what the grammar is.

29. Acknowledgements

Rob Pike, Howard Trickey and other Plan 9 users have been insistent, incessant sources of good ideas and criticism. Some examples in this document are plagiarized from [Bourne], as are most of *rc*'s good features.

30. Reference

S. R. Bourne, UNIX Time-Sharing System: The UNIX Shell, Bell System Technical Journal, Volume 57 number 6, July-August 1978