SHELISP — A Unix shell interface for Common Lisp

Alexandru Dan Corlan, MD, PhD

Bucharest, Romania

SHELISP — Unix shell interface with Common Lisp

`d/shelisp`

Version 3.2, January 15, 2011

Address for Correspondence
Dr. Alexandru Corlan,
`alexandru@corlan.net, http://dan.corlan.net`

This manual documents SHELISP, a very short and simple program, written in CommonLisp, that extends the CommonLisp syntax with constructs similar to unix shells, by invoking such shells.

Shelisp comes from the contraction of 'shell' and 'lisp' and should be pronounced accordingly (as if written shelllisp).

1 Quick guide to shelisp

This guide assumes you are on a Debian Woody GNU/Linux distribution, run the default shell (`bash`) and have `cmulisp` installed, and preferably also `emacs`. This is my setup and, while many others should work, this is the only one I tested on so far.

To run shelisp, say at the command prompt:

```
lisp -load shelisp.lisp
```

This should start CMU Common Lisp and provide the prompt, `*`. A more convenient form could be to start emacs, and issue the command `M-x cmulisp` that will start an 'inferior lisp' mode with cmu; then, say:

```
(load "shelisp.lisp")
```

1.1 The bang (!) escape to shell

Now you can say (the '*' is already put there by cmulisp):

```
* !ls
```

And it will execute the shell `ls` command (by running a `bash` instance and passing the command to it.

Of course, you are actually in Lisp. You can try this:

```
* (defun factorial (x) (if (zerop x) 1 (* x (factorial (1- x)))))
FACTORIAL
* (factorial 33)
8683317618811886495518194401280000000
```

So, if you enter "!" the rest of the line (until the first end of line that is not escaped with a "\") is interpreted as a bash command and the result is printed on the standard output.

Now try:

```
* !echo ?(+ 2 3) zuzu
5zuzu
```

The '?' is the 'lisp escape'. It is followed by an s-expression which is read, executed and printed (with `princ`) and the printed result replaces the '?' and the expression in the shell command. It can be any Lisp expression.

```
* !echo ?(+ 2/3 2/11) " <- this is a fraction"
28/33 <- this is a fraction
* !echo ?(factorial 100) " <- this is a beegnum"
933262154439441526816992388562667004907159682643816\
214685929638952175999932299156089414631\
976156518286253697920827223758251185210916864000\
00000000000000000000 <- this is a beegnum
```

Provided that you already entered the factorial definition above.

You may escape the '?' with a '\' to have it transfered to the shell command. for example:

```
* !echo \?\(+ 2 3\)
?(+ 2 3)
```

1.2 Embedded bash scripts

Anything written between square brackets is interpreted as a shell script. What the script prints on the standard output, however, is not displayed, but collected in a string and returned as a result of the bracketed expression.

For example:

```
* [echo hi there!]
"hi there!
"
```

One thing that you can't ordinarily do in bash:

```
* (dotimes (i 7) (princ [echo ?i ]) )
0
1
2
3
4
5
6
```

You can now say:

```
* (defun count-to (x) (dotimes (i x) (princ [echo ?i ]) ))
COUNT-TO
* (COUNT-TO 3)
0
1
2
```

Or, for example:[1]

[1] cmulisp ilisp interprets | as start of symbol and does not deliver a buffer to lisp until you enter a new | ; this is pure cmulisp "smartness"; you could finish the shell line with #|, for example, say:

```
...  (princ [echo ?i | sed 's/\(.\)/\1-\1-\1/' #| ]) ))
```

```
* (defun c-c-count-to (x)
        (dotimes (i x)
            (princ [echo ?i | sed 's/\(.\)/\1-\1-\1/' ]) ))
C-C-COUNT-TO
* (c-c-count-to 3)
0-0-0
1-1-1
2-2-2
```

1.3 Switching to shell mode (double-bang, !!)

If you enter a double bang (!!) then the prompter is changed to $ and you can issue unescaped shell command until you start a line with '!!' again—then you revert to 'lisp mode'.

Constructs with '?' are honored and are read and evaluated immediately by Lisp. Results of commands are printed immediately after being issued.

For example:

```
* !!
$ ls
Makefile
shelisp.lisp
shelisp_mn.aux
shelisp_mn.log
shelisp_mn.pdf
shelisp_mn.tex
shelisp_sc.aux
shelisp_sc.log
shelisp_sc.pdf
shelisp_sc.tex
shelisp.tex
spec.txt
$ # ?(setq bb 33.34)

$ echo ?bb " is " ?(sqrt bb) " squared."
33.34 is  5.77408 squared.
$ echo ?bb " is " ?(sqrt bb) " squared." >somefile.txt
$ cat somefile.txt
33.34 is  5.77408 squared.
$ echo "I am almost sure that " `cat somefile.txt`
I am almost sure that  33.34 is 5.77408 squared.
$ !!
$

NIL
*
```

Notice how purely Lisp commands, such as variable assignement (bindings), can be escaped with '#' characters as bash comments.

1.4 Run scripts as Lisp calls

The function `script` takes as argument a string and executes it as a bash script, returning the standard output of the script as a string.

```
* (script "ls")
"
Makefile
shelisp.lisp
shelisp_mn.aux
shelisp_mn.log
shelisp_mn.pdf
shelisp_mn.tex
shelisp_sc.aux
shelisp_sc.log
shelisp_sc.pdf
shelisp_sc.tex
shelisp.tex
spec.txt
"
```

1.5 Templates

A template is a string introduced with `#[` and ended with `]#`. It is treated like an usual string, however `?`-preceded lisp expressions are evaluated and their result printed inside the string.

For example:

```
(defvar *title* "Title of an empty page")

...

(prin1 #[Content-type: text/html

<html>
<head><title> ?*title* </title></head>
<body></body>
</html>
]#)
```

Will print to `*standard-output*`:

```
Content-type: text/html

<html>
<head><title>Title of an empty page</title></head>
<body></body>
</html>
```

1.6 Storable templates

One problem with templates is that we might desire to run them at a later time, in a different context. For example, we might want to define a variable with a generic web-page template and then generate actual web pages at later times, with various contents.

We use `#{` and `}#`for this purpose. In the example below notice that each time the value of variable A is evaluated, the BB in the evaluation context is used.

```
*(setf bb 9)
9
* (setf a #{ plus: ?bb :sulp }#)
(MIXED-TEMPLATE " plus: " BB " :sulp ")
* (setf bb 10)
10
* (eval a)
" plus: 10 :sulp "
* (setf bb 22)
22
* (eval a)
" plus: 22 :sulp "
* (defun calc-a (bb) (eval a))
CALC-A
* (calc-a 88)
" plus: 88 :sulp "
* (eval a)
" plus: 22 :sulp "
```

1.7 The `sh` macro

This macro aims to provide the use of shell commands with the same syntax as lisp forms, presumably for cases when arguments to commands are supposed to represent substantial computations in Common Lisp. It allows calling shell scripts with more lisp-like syntax.

Example call:

```
(sh ls -a)
```

Issues a `ls -a` shell call and returns a string with the standard output.

On the other hand:

```
(sh printf "%6.2f" (sqrt 45))
```

Will print the square root of 45 with two digits in a string.

The first argument after `sh`, the command name is not evaluated. Moreover, it is converted to lowcase letters, as the lisp reader, by default, interns symbols in all upcase (printf is interned as PRINTF). Also, symbol arguments starting with '-' are converted to lower case and not interpreted.

All other arguments are interpreted. You can use strings for command names and any other expression for switches and arguments.

If not sure, use strings. For example, to say `ls -A` enter:

```
(sh ls "-A")
```

If the result of an expression is a symbol, keep in mind that it will be upper cased by the reader by default. For example:

```
(defvar zz '(a b c))
(sh echo (cadr zz))
```

Will return a string starting with B followed by a newline.

Thus, the macro provides the shortest and most strightforward syntax for common commands and styles.

2 Backward compatibility

The shelisp interface is frequently normally in collections of scripts mixing a number of languages with minimal provision for maintenance. These scripts are employed for example in website generation.

New versions should always be drop-in replacements for existing ones. Sometimes, variants of the functions and macros and also of the syntax are proposed that do not respect the initial specification or not exactly.

We choose to leave the existing code untouched except for very serious bug fixes. Thus, programs using a version of shelisp should always work untouched with future versions.

New variants will have a numeric code apended (for example, `script3` for a new version of `script`).

3 External dependencies

Currently shelisp does not depend on or require any other specific module.

4 Technical details

Expressions preceded with '?' in the embedded shell scripts (with the '[]' syntax) are evaluated in the context where they appear, at 'eval' time; expressions in the 'bang' context (with the '!' or the '!!' syntax) are evaluated at read time, in the context of the last top level form before the form containing the bangs. This is because the bangs are intended for shell-only commands, normally given at the top level (command line) with immediate results. The embedded scripts are supposed to become part of functions or more complex forms, are parsed at read time and prepared to be executed at runtime.

In the 'bang' forms only simple shell commands can be issued as the reader does not detect the circumstances when a construct (such as ca 'case') occupies more than one line. In the embedded form or with the `script` command, any script can be executed.