The eval builtin command

Synopsis

```
eval: eval [arg ...]
```

Description

`eval` takes its arguments, concatenates them separated by spaces, and executes the resulting string as Bash code in the current execution environment. `eval` in Bash works in essentially the same way as most other languages that have an `eval` function. Perhaps the easiest way to think about `eval` is that it works in the same way as running `bash -c "bash code…"` from a script, except in the case of `eval`, the given code is executed in the current shell environment rather than a child process.

Examples

In this example, the literal text within the here-document is executed as Bash code exactly as though it were to appear within the script in place of the `eval` command below it.

```
#!/usr/bin/env bash
{ myCode=$(</dev/stdin); } <<\EOF
... arbitrary bash code here ...
EOF

eval "$myCode"
```

Expansion side-effects

Frequently, `eval` is used to cause side-effects by performing a pass of expansion on the code before executing the resulting string. This allows for things that otherwise wouldn't be possible with ordinary Bash syntax. This also, of course, makes `eval` the most powerful command in all of shell programming (and in most other languages for that matter).

This code defines a set of identical functions using the supplied names. `eval` is the only way to achieve this effect.

```
main() {
    local fun='() { echo "$FUNCNAME"; }' x

    for x in {f..n}; do
        eval "${x}${fun}"
    done

    "$@"
}

main "$@"
```

Using printf %q

The `printf %q` format string performs shell escaping on its arguments. This makes `printf %q` the "anti-eval" - with each pass of a string through printf requiring another `eval` to peel off the escaping again.

```
while (( ++n <= 5 )) || ! evalBall="eval $evalBall"; do
    printf -v evalBall 'eval %q' "printf $n;${evalBall-printf '0\n'}"
done
$evalBall
```

The above example is mostly fun and games but illustrates the `printf %q` property.

Higher-order functions

Since all current POSIX-compatible shells lack support for first-class functions (http://en.wikipedia.org/wiki/First-class_function), it can be tempting and sometimes useful to simulate some of their effect using `eval` to evaluate a string containing code.

This example shows partial application (http://en.wikipedia.org/wiki/Partial_application) using `eval`.

```
function partial {
    eval shift 2 \; function "$1" \{ "$2" "$(printf '%q ' "${@:3}")"
'"$@"; }'
}

function repeat {
    [[ $1 == +([0-9]) ]] || return
    typeset n
    while ((n++ < $1)); do
        "${@:2}"
    done
}

partial print3 repeat 3 printf '%s ' # Create a new function named pr
int3
print3 hi                            # Print "hi" 3 times
echo
```

This is very easy to do incorrectly and not usually considered idiomatic of Bash if used extensively. However abstracting eval behind functions that validate their input and/or make clear which input must be controlled carefully by the caller is a good way to use it.

Portability considerations

- Unfortunately, because eval is a **special builtin**, it only gets its own environment in Bash, and only when Bash is not in POSIX mode. In all other shells plus Bash in POSIX mode, the environment of eval will leak out into the surrounding environment. It is possible to work around this limitation by prefixing special builtins with the `command` regular builtin, but current versions of ~~ksh93~~ and zsh don't do this properly (fixed (http://article.gmane.org/gmane.comp.programming.tools.ast.devel/686) in ksh 93v-2012-10-24 alpha). Earlier versions of zsh work (with `setopt POSIX_BUILTINS` – looks like a regression). This works correctly in Bash POSIX mode, Dash, and mksh.
- `eval` is another one of the few Bash builtins with keyword-like conditional parsing of arguments that are in the form of compound assignments.

```
 $ ( eval a=( a b\\ c d ); printf '<%s> ' "${a[@]}"; echo ) # Only wo
rks in Bash.
<a> <b c> <d>
 $ ( x=a; eval "$x"=( a b\\ c d ); printf '<%s> ' "${a[@]}"; echo ) #
Argument is no longer in the form of a valid assignment, therefore or
dinary parsing rules apply.
-bash: syntax error near unexpected token `('
 $ ( x=a; eval "$x"'=( a b\ c d )'; printf '<%s> ' "${a[@]}"; echo )
# Proper quoting then gives us the expected results.
<a> <b c> <d>
```

We don't know why Bash does this. Since parentheses are metacharacters, they must ordinary be quoted or escaped when used as arguments. The first example above is the same error as the second in all non-Bash shells, even those with compound assignment.

In the case of `eval` it isn't recommended to use this behavior, because unlike e.g. declare, the initial expansion is still subject to all expansions including word-splitting and pathname expansion.

```
 $ ( set -x; touch 'x+=(\[[123]\]=*)' 'x+=([3]=yo)'; eval x+=(*); ech
o "${x[@]}" )
+ touch 'x+=(\[[123]\]=*)' 'x+=([3]=yo)'
+ eval 'x+=(\[[123]\]=*)' 'x+=([3]=yo)'
++ x+=(\[[123]\]=*)
++ x+=([3]=yo)
+ echo '[[123]]=*' yo
[[123]]=* yo
```

Other commands known to be affected by compound assignment arguments include: let, declare, typeset, local, export, and readonly. More oddities below show both similarities and differences to commands like declare. The rules for `eval` appear identical to those of let.

See also

- BashFAQ 48 - eval and security issues (http://mywiki.wooledge.org/BashFAQ/048) – **IMPORTANT**
- Another eval article (http://fvue.nl/wiki/Bash:_Why_use_eval_with_variable_expansion%3F)
- Indirection via eval (http://mywiki.wooledge.org/BashFAQ/006#Assigning_indirect.2BAC8-reference_variables)
- More indirection via eval (http://fvue.nl/wiki/Bash:_Passing_variables_by_reference)
- Martin Väth's "push" (https://github.com/vaeth/push) – `printf %q` work-alike for POSIX.
- The "magic alias" hack (http://www.chiark.greenend.org.uk/~sgtatham/aliases.html)

Discussion

Chris F.A. Johnson, 2012/06/03 22:13 ()

"just don't use eval" is not obligatory; it's just plain wrong!

Yes, care must be taken when using eval, but it's no different from many other commands in that respect.

Jan Schampera, 2012/07/01 11:46 ()

I agree. It should not be damned. Shell newbies should just be really warned. If one knows what he's doing, everything's fine

Dan Douglas, 2012/07/26 02:42 ()

I also agree (I wrote the page and that note). eval isn't as bad as some make it out to be - particularly in a language whose most fundamental principle is: "take a string, parse, evaluate, reparse, evaluate again, repeat until you eventually run a command". Warnings are needed for beginners. It takes some experience to know where eval is appropriate. Blah blah, we all know this.

I would also say that the more featureful of a shell you're using, the more infrequently it's appropriate to use eval.