Don’t BASH your head in:
Rx for shell variables.

Steven Lembark
Workhorse Computing
lembark@wrkhors.com
Quick review

BASH is interpreted.

Loops are re-parsed.

Variables can appear anywhere.

Unlike Perl, Python, Ruby, Go, Haskell, Scala, Scheme,
Which separate statements from var’s.
Basic Variables

Assignments to foo:

```plaintext
types='files';
```

literal ‘$files’.
Basic Variables

Assignments to foo:

foo='files';  \textit{literal} ‘files’.

foo="files"; \textit{interpolated value of files}.
Basic Variables

Assignments to foo:

foo='files';  
literal ‘files’.

foo="files";  
interpolated value of files.

foo=$(ls $files);  
command output.
Basic Variables

Assignments to foo:

- `foo='files'`; **literal `$files`**.
- `foo="files"`; **interpolated value of files**.
- `foo=$(ls $files);` **command output**.
- `foo="$(ls $files)"`; **string with listing of files**.
Basic Variables

Assignments to $foo$:

- $foo='\$files';$
  - literal ‘$files’.
- $foo="\$files";$
  - interpolated value of files.
- $foo=$(ls $files);
  - command output.

Most of the work is interpolating:

  echo "Your files are: $(ls $somedir)";
De-mangling variable names

> foo='bar';
> echo “foo$foo”; "foobar"
> echo “$bar_foo”; ""

Oops: Variable “bar_foo” doesn’t exist.
De-mangling variable names

> foo='bar';

> echo "foo$foo";       "foobar"

> echo "$bar_foo";     ""

Isolate ‘foo’ as variable name:

"${foo}_bar"          # "bar_bar"
Variable commands

cmd='/bin/ls';
arg=''-lt';;
Variable commands

cmd='/bin/ls';
arg='--lt';

files=$($cmd $arg $1);    /bin/ls -lt ...
# interpolate each command into the loop
for i in $cmd1 $cmd2 $cmd3
do
  $i $args;
done
Really anywhere!

foo='bar';
foo='bar';
$foo='blort';

Q: What happens?
Really, anywhere!

foo='bar';

$foo='blort';

Q: What happens?

A: Nada.

bash: bar=blort: command not found
Your one chance at success

BASH parses in two phases:

  Lexical substitution & tokenizing.

  Execution.

Variables have to expand on the first pass to be used. "foo=blort" cannot be executed, so it failed.
See what bash is doing with the variables:

```bash
> set -vx;
```
Verbosity & Execution

See what bash is doing with the variables:

```
> set -vx;
echo -ne "\033]0;;/$(basename $PWD) \007"
+++ basename
/sandbox/lembark/writings/RockfordLUG/bash
++ echo -ne '\033]0;;/bash \007'
>```
See what bash is doing with the variables:

```bash
> set -vx;
echo -ne "\033[0;../$(basename $PWD) \007"
+++ basename
/sandbox/lembark/writings/RockfordLUG/bash
++ echo -ne '\033[0;../bash \007'
```

Well... sort of.
“unset” removes variables

> unset PROMPT_COMMAND;
> set -vx;
>
Verbosity & Execution

> unset PROMPT_COMMAND;
> set -vx;
> foo=bar;

what I typed

foo=bar;
what BASH read

+ foo=bar

single ‘+’ is one level deep
Verbosity & Execution

```bash
> unset PROMPT_COMMAND;
> set -vx;
> foo=bar;
foo=bar;
+ foo=bar
```

`what I typed`

```bash
what BASH read
```

`single ‘+’ is one level deep`

```bash
> $foo='blort';
> foo='blort';
+ bar=blort
```

`no second chance to re-parse`
> unset PROMPT_COMMAND;
> set -vx;
> foo=bar;  

foo=bar; what I typed

+ foo=bar  

single ‘+’ is one level deep

> $foo='blort';
> foo='blort';

+ bar=blort  

no second chance to re-parse

bash: bar=blort: command not found
A second chance in life

‘eval’ adds one cycle. Interpolates variables. Passes result to the shell.

‘++’ is two levels deep.

> eval"$foo=blort";
+ eval bar=blort
++ bar=blort
> echo $bar;
+ echo blort
  blort
Or a third chance...

eval "eval ... ";

Work out what is happening:

a=' $HOME/?.* ’;
b=’foo’;
c=eval "eval $a $b"’;
We all remember backticks:

```
a=`ls -al ~`
```
Command execution

We all remember backticks:

\[
a=\texttt{`ls -al ~`'};
\]

Better off forgotten:

No way to nest them for one.

Hard to read for another.
BASH offers a better way:

\[ $( \ldots ) \]

i.e., “interpolate subshell output”.

Output of arbitrary commands:

```bash
files=$(ls ~);
jobs=$( grep 'MHz' /proc/cpuinfo | wc -l );
echo -e "DiskHogz:\n$(du -msx *|sort -rn|head)"
```
Twisting a path with basename

```bash
cmd='/image/bin/extract-hi-res';
dir='../raw';

for i in ../low-res/culled//*;
do
  echo "Input: '\$i'";
  $cmd $dir/$(basename $i .ppm).nef;
done
```

basename locates input for next step.
Twisting a path with basename

Quotes
hilite
whitespace in $1.

Don’t leave home without them...

```
cmd='/image/bin/extract-hi-res';
dir='..//raw';
for i in ../low-res/culled/*;
do
echo “Input: ‘$i’”;
  $cmd $dir/$(basename $i .ppm).nef;
done
```
Being there

A “here script” is “appended from stdin”.
Double-quotish.

> perl -MCPAN -E shell <<CPAN 2>&1 | tee a; upgrade
install Module::FromPerlVer
q
CPAN
Being there

A “here script” is “appended from stdin”.
Double-quotish, into stdin.

> perl -MCPAN -E shell <<CPAN
upgrade
install Module::FromPerlVer
q
CPAN
Being there

Closing tag sends EOF (^D) to command:

```bash
> perl -MCPAN -E shell <<CPAN
> upgrade
> install Module::FromPerlVer
CPAN
```
Being there

module='Module::FromPerlVer';

> perl -MCPAN -E shell <<CPAN 2>&1 | tee a;
upgrade
install $module
CPAN
#!/bin/bash

... 

path="$mysql_d/$tspace";
mkdir -p $path || exit -2;
mysql -U$user -P$pass <<<SQL || exit -3;
create tablespace $tspace using '$path' ... ;
create table big ( ... ) tablespace $tspace; SQL
Being there

mysql -U$user -P$pass <<SQL || exit -3;
create tablespace $tspace
using `$path' ... ;
create table
$(cat $table-1.sql)
tablespace $tspace;
SQL
Slicing with curlies

Remove strings from the head or tail of a string.

${i#glob}  \quad${i%glob}

${i##glob}  \quad${i%%glob}
Slicing with curlies

Slice the head:

${i#glob}$

${i##glob}$

#  is shortest match

## is longest match
Slicing with curly braces

Slice the tail:

```
${i%glob}
${i%%glob}
```

% is shortest match

%% is longest match
Stripping a prefix.

Say you want to prefix ‘/opt/bin’ onto a PATH. But it may already be there. You don’t know if someone else hacked the path.

Q: How can we put ‘/opt/bin’ at the front, once?
Stripping a prefix.

Say you want to prefix ‘/opt/bin’ onto a PATH.
But it may already be there.
You don’t know if someone else hacked the path.

Q: How can we put ‘/opt/bin’ at the front, once?
A: Take it off each time.
Striptease.

‘#’ strips off leading content.

Say we tried this:

```bash
PATH=""/opt/bin:${PATH#/opt/bin:}";
```

OK, I can run it a hundred times.
Path hack striptease.

‘#’ strips off leading content.

Say we tried this:

PATH=’/opt/bin:${PATH#/opt/bin:}’;

OK, I can run it a hundred times.

Until /opt/bin isn’t first:

“~/bin:/opt/bin: …”
Globs save the day

Find everything up to the first match:

`PATH="/opt/bin:${PATH#*/opt/bin:}"`;

```sh
> echo $PATH;
/usr/local/bin:/usr/bin:/bin:/opt/bin:/usr/i486-pc-linux-gnu/gcc-bin/4.1.2
```
Globs save the day

Find everything up to the first match:

PATH=${PATH#*/opt/bin:}";

> echo ${PATH#*/opt/bin:};
+ echo /usr/local/bin:/usr/bin:/bin:/opt/bin:/usr/i486-pc-linux-gnu/gcc-bin/4.1.2
Find everything up to the first match:

```
PATH="/opt/bin:${PATH#*/opt/bin:}"
```

```bash
> echo ${PATH#*/opt/bin:};
+ echo /usr/local/bin:/usr/bin:/bin:/opt/bin:/usr/i486-pc-linux-gnu/gcc-bin/4.1.2
```
Globs save the day

Find everything up to the first match:

```
PATH="/opt/bin:${PATH#*/opt/bin:}";
```

`/usr/i486-pc-linux-gnu/gcc-bin/4.1.2`
Fixing the path

Takes a bit more logic:

Strip /opt/bin out of the path.
Paste it onto the front.

Globs aren’t smart enough.
Fixing the path

Takes a bit more logic:

First break up the path.

```bash
> echo $PATH | tr ':' "\n"
/opt/bin
/usr/local/bin
/usr/bin
/opt/bin
/bin
/usr/i486-pc-linux-gnu/gcc-bin/4.1.2
```
Fixing the path

Takes a bit more logic:

Then remove ‘/opt/bin’.

```bash
> echo $PATH | tr ':' "\n" | grep -v '/opt/bin'
/usr/local/bin
/usr/bin
/bin
/usr/i486-pc-linux-gnu/gcc-bin/4.1.2
```
Fixing the path

Takes a bit more logic:

Recombine them.

```bash
> a=$(echo $PATH | tr ':' '
' | grep -v '/opt/bin' | tr '
' ':');
> echo $a
/usr/local/bin:/usr/bin:/bin:/usr/i486-pc-linux-gnu/gcc-bin/4.1.2::
```
Fixing the path

Takes a bit more logic:

Prefix ‘/opt/bin’.

```bash
> a=$(echo $PATH | tr ':' '
' | grep -v '/opt/bin' | tr '
' ':');

> echo "/opt/bin:$a";
/opt/bin:/usr/local/bin:/usr/bin:/bin:/usr/i486-pc-linux-gnu/gcc-bin/4.1.2::
```
Fixing the path

Takes a bit more logic:

Or, as a one-liner:

```bash
> PATH=="/opt/bin:$(echo $PATH | tr ':' "\n" | grep -v '/opt/bin' | tr -s "\n" ':')"

> echo $PATH
/opt/bin:/usr/local/bin:/usr/bin:/bin:/usr/i486-pc-linux-gnu/gcc-bin/4.1.2:
```
Quick version of basename

Strip off the longest match to ‘/’:

```
${file_path##*/}
```

Relative path within a home directory:

```
${file_path#$HOME}
```

Relative path in a sandbox directory:

```
${file_path##*/${(whoami)}/}
```
Getting some tail

Clean up a directory:  ${path%/}

Sandbox root:  ${file$$(whoami)/*}

Root of home:  ${HOME$$(whoami)*}

Less reliable dirname:  ${file_path%//*}
Default values

Common use is with arguments.

> rm -rf $1/*;

What if $1 is empty?

> rm -rf /*   # might not be what you want
Dealing with falsity

Common issue: Dealing with a NUL value.

Choose a default.
Assign a default.
Fail.
Lacking an argument, pick a value:

```
path=${1:-/var/tmp/input};
path=${1:-$input};
path=${1:-/var/cache/$(whoami)};
```

No effect on $1.
Assign a default value

Empty default assigned a value.

‘$’ interpolation may be nested:

```
"Default: '${default:=/var/tmp/$\{whoami\}}'";
```

“:=” does not work with positional parameters ($1...).
Giving up

Maybe not providing a value is an error.

```bash
rm -rf ${path:?:Path required.}/*
```

Code exits with “Path required.” prompt.
For example

#!/bin/bash

# if $1 has a value DEFAULT_PATH is ignored. 
# empty $1 checks for non-empty default.

path=${1:-${DEFAULT_PATH:?:Empty Default}};

# at this point path is not empty.
The next steps

Special Parameters:

$*, $@, $#    Command line

$?, $$, $!    Execution

Interpolate command line arguments, process control.
BASH interpolates variables in one pass.

${...}$ protect, slice variables

eval multi-pass processing.

<<TAG “here script”

-vx debugging

“Parameter Expansion” in bash(1)