Tools Talk

Unix Shells:

Expansion, Globbing, Pipelines, and much more

Greg J. Badros
University of Washington
Computer Science & Engineering
October 8, 1997

http://www.cs.washington.edu/homes/gjb/ToolsTalks/Shell.html

Overview

- What is a shell?
- Shell startup
- Variables
- Arguments and quoting
- Expansions
- Globbing
- Redirection and Pipelines
- Control Structures
- Job control, completion control, line editor keybindings
- Scripting and Functions

What is a shell?

- Common and important interaction method
- Interface to OS, especially to `exec`
  - manages environment variables
  - creates argument lists
  - opens file descriptors and pipelines
  - controls processes
- Shell does more than you might think
  - lots of substitutions, expansions
- Shell does less than you might think
  - not `ls`, `ps`, `env` other simple commands

Simple Example

```
cat $MDIR/notes.* ~/letters.* > Notes
```

- Variable substitution
- Filename (tilde) expansion
- Generates list of filenames (globbing)
- Pre-opens fd 1 (stdout) for I/O redirection
- Context of the current working directory
- What `cat` might get:
  - argc = 4
  - argv[0] = "cat"
  - argv[1] = "/home/gjb/Mail/notes.1"
  - argv[2] = "/home/gjb/Mail/notes.2"
  - argv[3] = "/home/gjb/letters.foo"
  - and `stdout` goes to `Notes`

Special cases

```
cat $MDIR/notes.* ~/letters.* > Notes
```

- $MDIR not defined?
- $HOME not defined?
- no files matching `/home/gjb/Mail/notes.*` ?
- no files matching `/home/gjb/letters.*` ?
- directory exists named `/home/gjb/letters.bak/`
- already a file names `Notes`

Some answers

```
cat $MDIR/notes.* ~/letters.* > Notes
```

- $MDIR not defined?
- $HOME not defined?
  - Error in csh, empty expansion in sh
- no files matching `/home/gjb/Mail/notes.*` ? `/home/gjb/letters.*` ?
  - Error if any don't match
  - or error if none match
  - or remove from argument list
  - or leave * unchanged
- directory exists named `/home/gjb/letters.bak/`
  - cat will give a warning
- already a file names `Notes`
  - Clobber? Could be dangerous
  - No clobber? Could be annoying

Better answers

- Man pages
- Info pages
- Zsh: `http://www.peak.org/zsh/`
- `http://www.tac.nyc.ny.us/mirrors/tcsh-book/` esp. `csh-whynot`

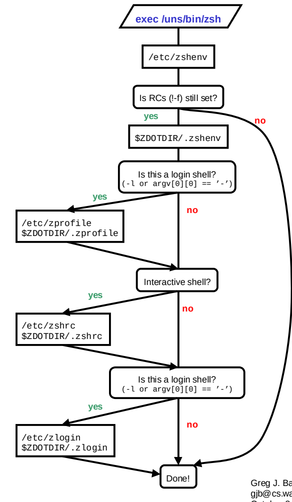
Choosing your shell

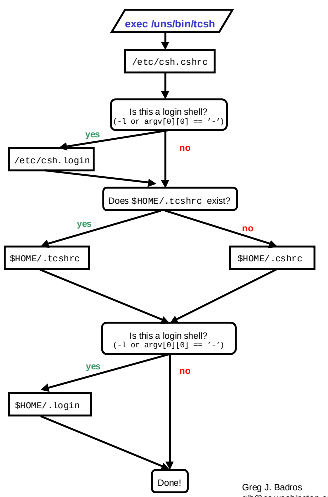
`chsh`, `passwd -s/e`, or `yppasswd -s`

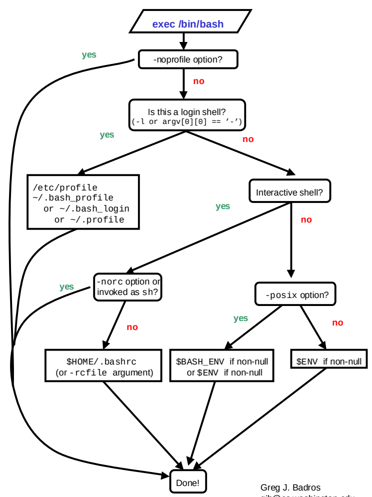
shell must be listed in `/etc/shells`

Zsh is not in /etc/shells, now what?

- Use csh, with `.cshrc` generated from your `~/.env` (see `make-cshrc-from-env`)
- Csh gets used for rsh/ssh, when only a simple environment is needed (faster startup)
- Or just `exec /uns/bin/zsh` from first line of `~/.cshrc`
- Be sure `~/.tcshrc` does not exist
- Source your `.env` from your .xsession and from your `.zshenv`







Variables

- Environment variables
  - e.g., `$PATH`, `$HOME`
  - available to child processes
  - only a string; lists simulated with `:`
  - list them with `env` command
  - means of setting options for programs
  - exist separately from the shell
    - e.g.,window manager can exec processes
- Regular shell variables
  - e.g., `$i`, `$PROCMAILDIR`
  - only available in shell
  - may have more complex structure
- Special variables
  - $?, $$, $#, `$PWD` "set" by shell
  - `$PROMPT` "used" by shell
  - `$path` is an array, e.g., `$path[4]`, automatically mirrors `$PATH`

Sh variables

`foo=bar` # just shell variable
`export FOO` # now make env. var
`export FOO2=baz` (not bare sh)

Csh variables

```
set foo=bar
setenv foo BAZ
```

What is $foo?

Csh variables hide environment variables

echo $foo ⇒ bar
env | grep foo ⇒ foo=BAZ

Note: no $ prefix when setting, it is a notation for replacing the variable name with its value

Other shell state

- options (from command line)
  - zsh has named options, too
- aliases
  - `alias ll 'ls -l'` (csh)
  - `alias print 'pr \!* | lpr'` (csh)
  - `alias mroe='more'` (sh)
  - \<command> avoids alias expansion
- functions
  - `title() { echo "^[]0;$*^G";}`
- completion control
  - zsh and tcsh, not csh or sh
- line-editing bindings
  - bash just uses readline library
  - tcsh and zsh have `bindkey` built-in

Passing argument lists

argc = 2
argv[1] = "`foo bar baz`"

is a lot different from

argc = 4
argv[1] = "`foo`"
argv[2] = "`bar`"
argv[3] = "`baz`"

Shells need to be able to create arbitrary argument lists. Quoting lets us control exactly what the arguments are.

Shell Quoting

- Shells split on unquoted whitespace in input
- Handy program to display arguments:

```
/* args.c */
#include <stdio.h>

int main(int argc, char **argv)
{
  int i;
  printf("argc = %d\n",argc);
  for (i=0; i < argc; i++) {
    printf("argv[%d] = \"%s\"\n",
           i, argv[i]);
  }
}
% foo=test   # or set foo=test
% args $foo bar baz
        ^     ^    ^
```

argc = 4
argv[0] = "args"
argv[1] = "test"
argv[2] = "bar"
argv[3] = "baz"

Shell Quoting \ vs "" vs ''

`\` escapes (suppresses shell meaning)
`'...'` escapes all enclosed chars
"..." escapes all except $, \, ``

```
% foo=Test
% args $foo\ bar\ baz
% args "$foo bar baz"
```

argc = 2; argv[0] = "args"; argv[1] = "Test bar baz"

```
% args '$foo bar baz'
```

argc = 2; argv[0] = "args"; argv[1] = "$foo bar baz"

$ is a special character, too

```
% args \$foo\ bar\ baz'
```

argc = 2; argv[0] = "args"; argv[1] = "$foo bar baz"

Shell special characters

whitespace [space, tab, newline]

$	Variable interpolation/ substitution
&, &&	Asynch exec, and
|, ||	Pipe, or
{ }	Grouping
( )	Subshell
< >	Redirection
#	Comment (not always when interactive)
[[ ]]	Zsh conditionals
;	Sequencing
!	History, not
*, ?	Pattern matching
~	Filename expansion

...more

Combining Quoting Types

Remember,
- Shells split on unquoted whitespace in input
- Single quotes are not special in double quotes
- Double quotes are not special in single quotes

```
% args "home is '$HOME'"
% args "home is "\'"$HOME"\'
argv[1] = home is '/homes/gws/gjb'
% args "home is "'$HOME'
argv[1] = home is $HOME
% args '$foo's value is $foo'
Error -- unmatched '
% args '$foo\'s value is $foo'
Error -- \ has no special meaning in ' '
% args '$foo'"'s value is $foo"
argv[1] = $foo's value is Test
```


What about whitespace in variable expansions?

```
% foo="this is a test"
% args $foo
argv[1] = this; argv[2] = is; ...
(except zsh, unless setopt shwordsplit)

% args "$foo"
argv[1] = this is a test

% args ${=foo}                    #zsh
% foo_array=(this is a test) #zsh
% args $foo_array                #zsh
argv[1] = this; argv[2] = is; ...

But
% args "$foo_array"              #zsh
argv[1] = this is a test
```


`find` Example

```
% find . -name "*~" -exec rm {} \;
```

What does find get for arguments?

```
% args . -name "*~" -exec rm {} \;
argv[1] = .
argv[2] = -name
argv[3] = *~
argv[4] = -exec
argv[5] = rm
argv[6] = {}          # ok because no space
argv[7] = ;
```

`*~` is subject to filename generation
`;` is the shell sequencing operator

```
% args . -name *~ -exec rm {} ;
argv[3] = bar~
argv[4] = baz~
....
argv[argc-1] = {}    # ; got removed
```


Variable expansions

`$foo` can be written as `${foo}`
Safer in:
`setenv PATH /uns/bin:${PATH}:.`
- trailing colons are modifiers in zsh, (t)csh
- zsh ignores bad modifiers, (t)csh complains

(T)csh/Zsh Modifiers

leave :head
leave :tail
:remove ext
leave :ext
:substitute (e.g., `$foo:s/.c/.o/`)

But csh only allows one

```
% foo=/home/gjb/args.c
% echo $foo:t:r ⋗ args.c:r #csh
% echo $foo:t:r ⋗ args
```

Use braces! Prefer `${foo:t:r}`

Sh's variable expansions

```
% foo=test.bar.baz
echo ${foo%.*}       ⋗ test.bar
echo ${foo%%.*}      ⋗ test
echo ${foo#*.}       ⋗ bar.baz
echo ${foo##*.}      ⋗ baz
```

Note: `.` is a literal; not regexp meta-character

% and # cause non-greedy match
%% and ## take the longest match

Can use

* = Multiple characters
? = single character
[..] = any of characters ..
[^..] = anything but characters ..

in patterns.

Other expansions

(All available in Zsh)

Order is:

1) history (interactive only)
   e.g., `!$` is last argument from prior line
   `^cat^more` replace cat with more
2) alias; e.g., `alias ll='ls -l'`
3) process substitution, variable expansion, command substitution, arithmetic expansion, brace expansion (same time, left to right)
4) Removal of unquoted `\`, `'`, `"`
5) filename expansion (could be after alias substitution if setopt sh_file_expansion)
   e.g., ~ ⋗ `$HOME`
6) filename generation

History Expansion

<event>:<word>[:<modifier>]+

Event specifiers:

`!!`	previous command
`!`*n*	command n (use history)
`!-`*n*	relative command n
`!#`	current command line
`!`<str>	command starting with
`!?`<str>	command containing

Word specifiers:

k	argument *k* (0 for command name)
^/$	first/last argument
x-*y*	arguments *x* through *y*
x-	arguments *x* through $-1
*x**	arguments *x* through $
-*y*	arguments 0 through *y*

```
% mv long_filename !#$.old
% ^update^commit # like !!:s/commit/update/
```

Often, prefer interactive line editing.

Command Substitution

`‘`*pipeline*`‘` ⋗ stdout from pipeline
(Zsh and Bash allow `$(`*pipeline*`)`)

```
% cat ‘echo foo bar‘ ⋗ cat foo bar
% cat $(echo foo bar)⋗ cat foo bar
```

Advantage of `$(..)` is that they can be nested

stdout is split into arguments
at whitespace:

```
% args ‘echo This is a test‘
```
argc = 5
```
% args "‘echo This is a test‘"
```
argc = 2

Common Files Example

```
% ls -d ‘cd /tmp/gjb; /bin/ls‘ 2>/dev/null
```

Lists filenames that are common between the current directory and `/tmp/gjb`

`-d` is a switch to ls telling it not to list contents of directories names in the argument list

Use `/bin/ls` to defeat aliasing of `ls`
(`-F` option would append file type marks to the listing that would get in the way)

`2>/dev/null` redirects stderr (fd 2) to standard device which throws away its input

Brace Expansion

```
% echo args.{c,h,o}
```
args.c args.h args.o

```
% echo args.[cho]
```
args.c args.h
(Note args.o didn't exist)

```
% echo file{1..3}
```
file1 file2 file3

```
% head ~{gjb,billg,rms}/.profile
```
Could give file not found

Use a glob, too, to filter out non-existing files

```
% head ~{gjb,billg,rms}/.[p]rofile
% mv long_filename{,.old}
% echo ${^manpath}/**/*(.)
```


Arithmetic Expansion

(and evaluation)

`$[ .. ]` replaces .. with its evaluation as an expression (Zsh)

Other shells, use expr command, e.g.,

```
% set i=‘expr $i + 1‘  # csh
```

`let` built-in evals args and returns 0 if last expression is nonzero

```
% let i++  # Zsh
```

`((..))` == `let ".."`

```
% if (( $# > 9 )); then
%     echo "Too many args"
% fi
```

Numbers: <base>#<value
e.g., 16#ff == 255_{10}
or [16]ff (deprecated)

Filename Expansion

(a.k.a. tilde expansion)

~	⋗	`$HOME`
~<username>	⋗	home directory
~+	⋗	`$PWD`
~-	⋗	`$OLDPWD`
~<variable>	⋗	`$VARIABLE`
~1	⋗	Top of the directory stack

Zsh-only:

=<cmdname>	⋗	`‘which <cmdname>‘` (text of aliases is used)

Directory Stack

Use `dirs` to view it;
`pushd` and `popd` to manipulate it.

Related Zsh options:
`auto_pushd, pushd_silent, pushd_to_home`

Also: two argument form of `cd/pushd`

```
%cd gjb billg #cd ‘echo ~+ | sed s/gjb/billg‘
```

Filename Generation (Glob)

Unquoted *, ?, [..]

```
% file *.[ch]  # character class

% for i in ?*; do
%     test -f $i && cp $i ~/dots
% done
```

Ignore `*.o` files (requires Zsh setopt extendedglob)

```
% egrep zsh *~*.o
```

`**/*` All non-dot files under current directory

All files? Need a Zsh advanced glob modifier

Other shells: use something like:

```
  find . -print0 | xargs -0 <cmd>
```


Zsh Glob Modifiers

`*(.)`	Only regular files
`*(/)`	Only directories
`*(D)`	Include dot files
`*(g{grad_cs})`	Group grad_cs
`*(u{gjb})`	Owned by gjb
`*(u{^root})`	Not owned by root
`*(mh-2)`	Changed in last 2 hours

Combinations

`**/*(Du{gjb},g{uns}mh-2)`
All files owned by gjb, or
group uns and modified in last two hours

`**/*(Lk+500:h)`
Directories with files bigger than 500kb

`output.lo[g]{1..10000}`
(or use setopt numeric_glob_sort)

Redirection

>	Redirect stdout (use>> to append)
<	Redirect stdin
2>	Redirect stderr (fd 2) (2>> to append)
>&	Redirect stdout + stderr

(Also can append `!` to force overwrite/create)

Zsh allows multiple '<'s and copies the input from the files successively.

Zsh allows multiple '>'s and outputs a copy to each of the files.

e.g.,

```
% awk '{print $1}' <foo <bar \
   2>errs >out1 >out2
```

Note: \ escapes the newline for continuation of a command line (like in C source)

More redirection

Here documents

```
% cat <<-FOO
  Copied to stdout
   after stripping initial spaces
FOO
```

Use <<'FOO' to suppress variable and command substitution within the body.

"Now" strings

```
% args <<<"This gets put on stdin"
```

(T)csh Redirection is broken

- No 2> form is available
- stderr can not be redirected separately from stdout; must use the "subshell hack"

```
% ( ls * >out ) >& err
```


Pipelines

```
% ps aux | grep xnewbiff |\
     awk '{print $2}'
```

or, better (`exec`-s are expensive!)

```
% ps aux | awk '/xnewbiff/ \
        {print $2}'
```

- Does not create a temporary file; uses popen system call and lets OS manage the buffers
- First process (producer) stalls if consumer is too slow (common for interactive consumers such as `more` and `less`)

Process substitution

`<(pipeline)`	Substitute a named pipe fed by the given pipeline
`>(pipeline)`	Pipe to given pipeline
`=(pipeline)`	Substitute a temporary file containing the output of the given pipeline (This is `lseek`-able)

e.g.,

```
% paste <(cut -f1 one) \
        <(cut -f2 two) \
   | tee >(sed 's/foo/bar' >out) \
     out

% args =(echo foo)
argv[1]=/tmp/zsha13911
```

Subshells vs. Command Lists

```
% rsh uni cat /tmp/arch/foo.tgz |\
     (cd decompress; tar xvzf - )

% rsh uni cat /tmp/arch/foo.tgz |\
     {cd decompress; tar xvzf - }
```

First is slower, but protects initial shell from side-effects (in this case, cd)

Second doesn't exec another shell, so is faster, but can influence top level shell; directory will be changed after pipeline is executed.

e.g., copying a whole directory tree:

```
% cd dest
% (cd src; tar cf - . ) | \
     tar xvpf -
```


Control structures

for loop:

```
i=0
for arg in "$@"; do
  echo "argv[$i] = $arg"
  i=‘expr $i + 1‘
done
```

until loop:

```
until ping -c 1 pahtoo; do
  sleep 5
done
```

while loop:

```
while ! ps aux | grep ical; do
  sleep 1
done
```

`continue` and `break` act as in `C`

Exit Codes (Return Values)

A process returns 0 to indicate successful completion, or non-zero to indicate failure

e.g., `/bin/true` returns 0; `/bin/false`, 1 (opposite of most other languages)

In Zsh, you can have it print non-zero exit values using `setopt printexitvalue`

Pipelines return the exit code of the consumer.

Conditionals simply test the exit codes, as do the short-circuiting && and || operators

```
% grep foo <in || echo "no foo"
% mkdir newdir && cd newdir
% test -f bar && ls >>bar
```


Conditionals

```
% if [ "$foo" = test ]; then
%       echo "\$foo is test"
% fi
```

[is just another name for test which requires] as its last argument

[-t 0] true iff stdin is a tty (interactive)
 (false from your .xsession)
[-r readable] [-e exists]
[-f ordinary_file] [-d directory]
[-s non_empty (size > 0)]

See test(1)

Zsh also has [[ ]] special form, which allows patterns, and does not do an `exec` (faster)

```
% if [[ ":${PATH}:" != \
          *:/uns/bin:* ]]; then
%     export PATH=$PATH:/uns/bin
% fi
```


Job Control

Trailing & on pipeline to run asynchronously

`Ctrl-Z` sends SIGSUSPEND, to pause foreground process

`bg`, `fg`, `jobs`, `disown`

Use %<number> to refer to a pipeline
Also %<name>, %-

These are **not** expansions, only notation understood by the job-control builtins!

Pre-command Modifiers

`noglob`	like escaping all `*`, `?`, `[..]`
`exec`	do not fork, replace instead
`command`	do not use builtin
`var=val`	set only in child's environment

e.g.,

```
% HOME=/local/gjb xterm -e pine &|
```


Other interactive features

```
% compctl -g '*.dvi(D) *(-/D)' \
          xdvi dvips
% vared PATH
% dirs; pushd; popd; cd -
```

`Ctrl-V` in line editor quotes next char, e.g, <Tab> or <Escape>

bindkey maps character sequences to command line editing functions (Tcsh/Zsh)

```
% bindkey "\e[k" backward-kill-line
```

Bash uses `readline`, and its `.inputrc`

Xterm resources specify what keysyms send the appropriate control sequences:

```
XTerm*VT100*Translations: \
...
!Meta<Key>BackSpace: string("^[[k") \n\
...
```

Shell Scripts

```
#!/bin/sh
# $Id$
xrdb -cpp cpp-sed-nl "$@"

#!/uns/bin/zsh -f
# $Id$ -*- zsh -*-
if [[ ! $# = "1" ]]; then
  echo "Usage: $0 <file.man>" 1>&2
  exit 1
fi

# If stdout is a tty
if [[ -t 1 ]]; then
    OUT=${1:t}
    echo "Outputting to $OUT.ps" 1>&2
    groff -Tps -mandoc $1 >| $OUT.ps
else # stdout is a pipe or a file
     # so let stdout go as usual
    groff -Tps -mandoc $1
fi
```


Script and Function Arguments

`$0`	name of program as exec-d
`$#`	number of arguments
`$1`	first argument
`$*`	all arguments
`"$*"`	all arguments, as a string
`"$@"`	all arguments, individually quoted

e.g., myscript foo bar baz
would set (inside myscript):

`"$*"` == "foo bar baz"
but `"$@"` == `"foo" "bar" "baz"`

Use `"$@"` to pass arguments to other pipelines

`shift` renames $2 to $1, $3 to $2, etc.

`getopt` built-in also can help with argument processing

Functions

```
# cd to a new directory and ls
z() {
  if [ ! "$1" = "" ]; then
    cd $1
  fi
  ls
}
```

optional

```
function add-path() {
  if [ $# -gt 2 -o $# -eq 0 ]; then
    echo 'Usage: add-path /some/dir \
       [PATH_VARIABLE]' 2>&1
    return 1
  elif [ $# -eq 1 ]; then
    2=PATH
  fi
  setopt local_options no_glob_subst
  eval "$2=\"\$$2:$1\""
  cleanup-path $2
}
```

`$FPATH` variable controls search for `autoload`ed functions (Zsh)

Aliases vs. Functions vs. Scripts

- **Aliases**
  - can inherit completion from expansion
  - useful for adding default arguments
  - poor at dealing with arguments
  - % alias cp='cp -p'
- **Functions**
  - do not require an exec, so faster
  - often shell-specific
  - can affect current environment
  - hard to change due to caching in shell state
    - `refunc() { unfunction $1; autoload $1 }`
  - consumes memory (use `autoload`)
  - special: `precmd`, `chpwd`, `periodic` (Zsh)
- **Scripts**
  - more universal (not shell dependent)
    - usable from window manager, Emacs
  - cannot change current environment
  - requires exec, but can choose language