Shell Programmierung

David Frey

Einführung

Dieser Text gibt eine Einführung in die *Bourne* (`sh`) und *Bourne-Again-Shell* (`bash`) und deren Programmierung.

Die Konfiguration wird aus Platzgründen nicht behandelt und wäre ein Vortrag für sich.

Die `bash` ist aufwärtskompatibel zur `sh` [1, 2]. Somit ist es wichtig die Unterschiede erwähnen, um zu verhindern. dass man aus Versehen ein `bash`-Programm anstelle eines `sh`-Programmes schreibt.

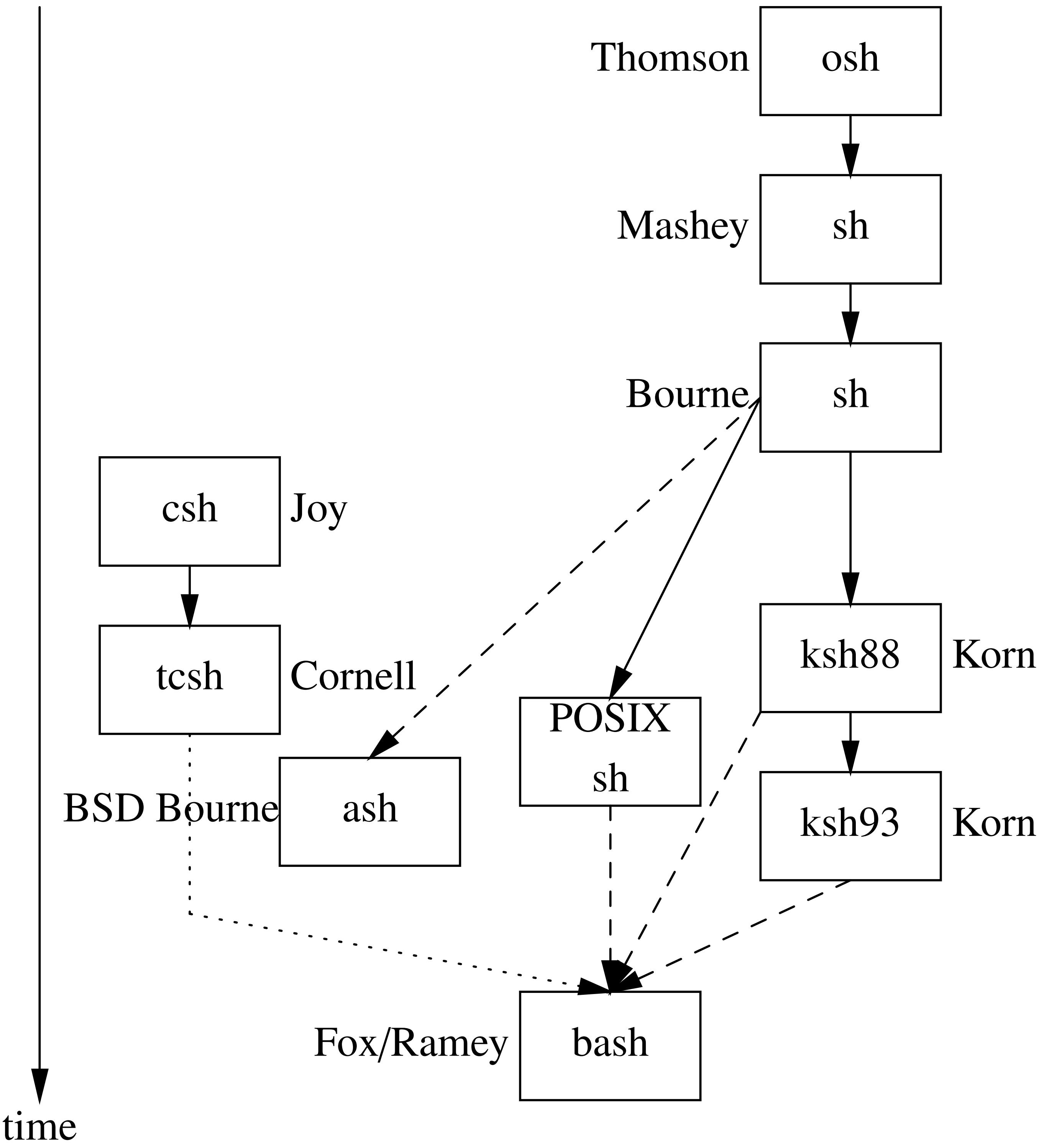
Die Shell erfüllt 2 Funktionen:

1. interaktive Arbeitsumgebung
2. Programmiersprache, um obige zu automatisieren

Was macht die Shell?

- Eingaben entgegennehmen und parsen. Allenfalls Editiermöglichkeiten bereitstellen (`bash`,`ksh`),
- Programme starten,
- Prozesse verwalten: in den Hintergrund setzen, vom Hintergrund in den Vordergrund holen, suspenden, killen,
- Environmentvariablen verwalten und anderen Programme zur Verfügung stellen (“exportieren”).

Geschichte



Thomson
osh
Mashey
sh
Bourne
sh
csh
Joy
tcsh
Cornell
ksh88
Korn
POSIX
sh
ksh93
Korn
BSD Bourne
ash
Fox/Ramey
bash
time

einfache Kommandi

`vi /etc/crontabs`, `crontab -r`.

Die Shell unterscheidet zwischen eingebauten (*internen*) und *externen* Kommandi. Der Exitstatus wird in der Variablen `$?` zurückgegeben.

interne

Klassische interne Kommandos ändern den Zustand der Shell oder erweitern sie:

- set, eval
- Funktionen

Mit der Zeit wurden einige externe Kommandos aus Performancegründen zu Internen:

- history
- test, Arithmetik

externe

Alle nicht-internen, typischerweise im Pfad.
`ls`,`diff`,`mutt`,`emacs`,`vi`,`nethack`, `latex`,...`pom`

Filenamen expandieren (“Globbing”)

Das Expandieren von Wildcards in Filenamen.

`*`	beliebige Anzahl Zeichen passen
`?`	beliebiger Buchstabe passt
`[...]`	Die Zeichen im Bereich sollen passen
`[^...]`	Die Zeichen im Bereich sollen nicht passen
`{...}`	Aufzählung (`bash` mit Brace-Expansion)

Zeichenklassen (“character classes”)

Sind in ‘`[:`’ und ‘`:]`’ eingeschlossen:

alpha, alnum	alphabetischer, -numerisches Zeichen
cntrl, ascii	Control-, ASCII-Zeichen
blank, space	Space oder Tabulator, White-Space[a]
digit, xdigit	Dezimalzahlen, Hexzahlen
lower, upper	Klein-, Grossbuchstaben
print, graph	druckbares Zeichen mit und ohne Space
punct, word	Punktuation, Identifier[b]

[a] space, FF, NL, CR, tab und vertical tab

[b] Buchstaben, Zahlen und ‘_’

Filenamen unter Unix

Nicht alle Buchstaben sind unter Unix in Filenamen erlaubt oder intelligent:

1. Im Kernel speziell (unmöglich):

`\0`	String-Ende-Kennzeichen in C
`/`	Directory-Trennzeichen unter Unix
`. ..`	Current-, Parent-Directory

2. In der Shell speziell (quoten):
   (a) Alle Buchstaben in `$IFS`[a]-Variablen.[b]
   (b) Zeichen mit Sonderbedeutung in der Shell-Grammatik:

`*?[]^`	Globbing-Sonderzeichen
`>|<`	I/O-Redirection, Pipes
`&;`	Hintergundjobs, Command-Trenner
`'"`‘`\`	Quoting, Quote-Character
`$[]`	Variablenexpansion, Arrays
`(){}`	Subshells
`,`	innerhalb der Brace-Expansion[c]

[a] per Default Space,Tabulator, Newline
[b] Der Inhalt des `$IFS` wird zum Word-Splitting verwendet.

(c) White-Space und Control-Character sind aus praktischen Gründen (Sichtbarkeit im normalen `ls`) nicht empfehlenswert.

(d) Einige Utilities verwenden '`@`' und '`:`' für remote-Files (vor allem `rsh` und `ssh`)

(e) Problematisch: '`!`' wird für History-Expansion[d] und UUCP-Bang-Paths verwendet.

(f) Nicht-ASCII-Zeichen (z.B. Umlaute) sind Zeichensatz-abhängig.

P.S: Nicht alle Utilities kennen, die Option '`-0`'.

[d] abschaltbar

Redirection

Redirection wird zum Lesen und Schreiben von Files (bei der `bash` auch Sockets) verwendet.

< (≡0<)	stdin von File umleiten
`n<`	File auf Filedeskriptor *n* lesend umleiten
> (≡1>)	stdout auf File umleiten
2>	stderr in File umleiten
`n>`	File auf Filedeskriptor *n* schreibend umleiten
`n>&m`	Filedeskriptor *n* nach *m* umleiten
`n<&m`	Input-Filedeskriptor von *m* auf *n* kopieren
`n<>`	Filedeskriptor *n* zum Lesen und Schreiben öffnen.

Ein >> anstelle von > bedeutet anhängen.

`&>word` ist gleichbedeutend mit `>word 2>&1`.

Here-Dokuments

```
prog<<EOF
input
EOF
```

und

```
prog<<-EOF
<TAB>input
EOF
```

`<<-` schneidet führende Tabulatoren weg.

Here-Documents sind “syntactic sugar” um statische Files zu vemeiden; anstelle von `prog < input` kann man nun schreiben:

```
prog<<EOF
Bla fasel bla gunsch

Fnord.
   $LOGNAME
EOF
```

Environmentvariablen

Skalare Variablen werden mit `VAR=VALUE` gesetzt und mit `$VAR` ausgelesen

Variablen mit Spezialbedeutung:

1. Spezialbedeutung für die Shell
   - IFS
     Input-Field-Separator
   - SHELLOPTS, HISTCONTROL, HISTSIZE, DIRSTACKSIZE alle `bash`.
   - PS1, PS2, PS3, PS4
     Prompts in aufsteigender Schachtelungstiefe
   - PATH
     diese Directories werden nach ausführbaren Programmen durchsucht
   - ?
   - $,!
   - @, 1...9, *, #
2. von der Shell ausgewertet, aber für anderen Programme gedacht:
   - TERM

- LANG, LC_CTYPE
- MAIL, MAILCHECK für die “You have mail.”-Meldung. MAIL wurde von `login`(1) gesetzt.
- EDITOR (ksh)

3. von der Shell für andere Programme zur Verfügung gestellt:
   - UID, EUID
   - PWD
   - LOGNAME, USER
     von `login`(1) gesetzt.
   - HOME
   - DISPLAY vom X-Server (`xdm` gesetzt)
   - SHLVL
4. Für andere Programme vom User gesetzt:
   - SHELL
   - MAILER, MAILNAME
   - MANPATH, INFOPATH, INFOCOMPATH
   - PAGER
   - EDITOR
   - LANG
   - PRINTER

- LESS, LESSBINFMT, LESSCHARSET
- LS_OPTIONS, LS_COLORS, VERSION_CONTROL
- TIMEFORMAT, TIME,
- NETHACKOPTIONS
- IRCNICK, IRCNAME,
- NNTPSERVER
- RSYNC_RSH
- …

Prompting

PS1 ist Hauptprompt, z.B:

`PS1="At your service? "` mit mehreren Expansions, z. B. `\$`, `\u`, `\h` und `\w`:

`PS1=(\u@\h) \w\$\[\033]0;\u@\h:\w\007\]`

Evaluation (1)

Arithmetic Expansion (bash, ksh)

Integer (2^{32})-Arithmetik mit den aus C wohlbekannten Operatoren:

`++,--,!,~,**,*,/, %,+,-,<<,>>,<=,>=,`
`<,>,==,!=,&,^,|, &&,||,?:,,, [*/%+-<<>>&^|]=,,.`

Für Floating-Point `dc` (oder `bc`) benutzen.

In der `sh` kann man `expr` um zu Rechnen.

Arrays (ksh, bash)

Wie in C von 0 an indizierte, in der `ksh` auf maximal 2048 Elemente begrenzt.

Zugriff durch `${ARRAY[$IDX]}`, wobei ‘*’ für alle Elemente steht.

Evaluation (2)

Parameter-/Textsubstitution (bash)

`${...}` wobei ... für folgendes steht:

`p:-w`	*w* nehmen, falls *p* nicht gesetzt oder Null
`p:=w`	*w* zuweisen, falls *p* nicht gesetzt oder Null
`p:?w`	Aussteigen, falls *p* nicht gesetzt oder Null
`p:+w`	Falls *p* gesetzt, *p* auf *w* setzen
`p:o`	Substring herauslesen
`p:o:l`	Substring der Länge *l* herauslesen
`!p*`	Variablen, die mit *p* anfangen, aufzählen
`#p`	Länge von *p*
`p#w`	kürzesten
`p##w`	längsten Substring am Anfang entfernen
`p%w`	kürzesten
`p%%w`	längsten Substring am Ende entfernen
`p/r/s`	Erste
`p//r/s`	alle Strings ersetzen

In der `sh` mit `expr` resp. `sed` lösen.

Quoting und Expansion

Quoting

'\' quotet das nachfolgende Zeichen und nimmt die Spezialbedeutung desselben weg.

`echo *` ⇒ Inhalt des current directories

`echo \*` ⇒ *

Expansion

`"..."` *double quotes*
Variablen und gequotete Zeichen werden expandiert

`'...'` *single quotes*
Literaler Text, keine Expansion

`` `...` `` *backticks, backquotes*
Text in den Backquotes wird als Shell-Eingabe interpretiert und ausgeführt, der Output auf stdout wird dann eingesetzt

Beispiele

```
#!/bin/bash
SEQ='|/-\'
OFF=1
while $@; do
  printf "%c\b" ${SEQ:$OFF:1}
  OFF=$(( (OFF+1) % 4))
done

gpg --recv-key $(\
gpg --list-sigs --with-colons "David Frey"|\
awk -F: '$5 != "" &&\
         $10 ~ /\[User id not found\]/ {\
           print $5;\
         }'|sort|uniq)
```

zusammengesetzte Kommandi (1)

Backgroundjobs

Werden im Hintergrund ausgeführt:
`cc longcompilation.cc&`.

Sie blockieren das momentane Terminal nicht und laufen asynchron. Man kann mit `wait $!` auf das Kommando warten. Die PID des zuletzt im Hintergrund gestarteten Programms ist in der Variablen `$!` gespeichert.

zusammengesetzte Kommandi (2)

Pipelines

Pipes verketten einfache Kommandi so, dass der **stdout** des Programms vor der Pipe zum **stdin** des Programmes nach der Pipe wird. Pipes sind unidirektional.

Dies die einfachste Art der Interprozesskommunikation und sehr effektiv. Im Vergleich zur Bauernmethode , `prog1 > tmp; prog2 < tmp` sind Pipes eleganter, schöner, schneller (volle Memory-Bandwidth steht zur Verfügung) und brauchen keinen Diskplatz.

```
#!/bin/sh
# freq: The classical pipe for getting word
#       frequencies.
cat $*|col -b|\
tr -cs '[:alnum:]' '\n'|\
tr '[:upper:]' '[:lower:]'|\
sort|uniq -c|sort -nr
```

zusammengesetzte Kommandi (3)

Logik

- `true`
  Alles was einen Returncode von 0 zurückgibt. Gleichnamiger Befehl.
- `false`
  Alles was einen Returncode $\neq 0$ zurückgibt (in der Manpage zu den Befehlen steht die Bedeutung der Returncodes).. Gleichnamiger Befehl gibt 1 zurück.
- &&
  Short-circuit Boolean AND
- `||`
  Short-circuit Boolean OR

Shell-Programmierung (1)

Tests

Werden mit **test** resp. häufiger mit **[** und **]** gemacht.

Auswahl der Testoperationen:

`-e`,`-f`	File existiert, ein reguläres File
`-r`,`-w`,`-x`	File ist lesbar, schreibbar, ausführbar
`-s`	File ist nicht leer
`-z` (`-n`)	String ist (nicht) leer
=, `!`=,<,>	String sind gleich, ungleich, kleiner, grösser
`-eq`,`-ne`	Zahlen sind gleich, ungleich,
`-lt`,`-le`	kleiner, kleiner gleich,
`-gt`,`-ge`	grösser, grösser gleich

if *bool* then . . . [elif . . .][else . . .] fi

Shell-Programmierung (2)

for *var* in *list* do *stmts* done

```
for f in *.jpg; do
  mv $f $(echo $f|sed -e s/img_00//);
done
```

```
#!/bin/sh
# jpegrot90
for f in $*; do
  jpegtran -rotate 90 $f > $f.new
  touch -r $f $f.new
  mv $f.new $f
done
```

for ((*expr1*; *expr2*; *expr3*)); do *list*; done

Arithmetisches `for` (`bash`).

while ... do ... done

do ... until ... done

until ... do ... done

Typische Schlaufen wie in C.

“Hello world” in bash

```
#!/bin/bash
#(c) David Frey, August 2002, GPL

function help {
  cat<<EOH
This is GNU Hello, THE greeting printing program.
Usage: hello [-htvm]
 -h  Print a summary of the options
 -t  Use traditional greeting format
 -v  Print the version number
 -m  Print your mail
EOH
  exit 0
}

function thello {
  echo "hello world"; exit 0
}

function hello {
  echo "Hello, world!"; exit 0
}
```

“Hello world” in bash (cont’d)

```
while getopts htvm s; do
  case $s in
    h) help;;
    t) thello;;
    v) echo "BASH Hello, version 0.0" >&2; exit 0;;
    m) { [ -n "$MAIL" ] && cat $MAIL; }; exit 0;;
  esac
done
shift $(($OPTIND - 1))

if [ "$1" = "sailor" ]; then
  echo "Nothing happens here.";
elif [ "$#" -eq 0 ]; then
  hello
else
  help
fi
```

Shell-Programme und Funktionen

Shell-Programme und Funktionen erhalten als

Parameter:	`$0`	⇒ Name des Programms
	`$1...$9`	⇒ Parameter 1...9

Funktionen

```
lssize ()
{
  find ${1:-.} -type f -xdev -printf "%4k %h/%f\n"|\
  sort -r -n
}
```

Lokale Variablen (bash)

Werden mit `local` gekennzeichnet.

Subshells

(*list*)	*list* wird in einer Subshell ausgeführt.
{*list*;}	*list* wird in derselben Shell ausgeführt.

eval

Führt ein Shell-Kommando aus (evaluiert es) und lässt den Output die Shell ausführen.

Unterschiede sh ↔ bash (“bashism”)

Die `bash` ist zu der in POSIX.2 [3] standardisierten Shell (“`sh`”) aufwärtskompatibel.

Die wichtigsten Sachen, die in der `sh` **nicht vorhanden** sind, aber in der Bash schon: (→ Bash-FAQ C1):

- History, Brace, Tilde-Expansion, Prozess-Substition
- die **function**,**select**-Schlüsselworte,
- Arithmetik, Arrays,
- gewisse Quoting-Mechanismen (z.B: `$()`)
- eine Reihe von Shell-Variablen,
- gewisse Redirections, z.B. <>

Unterschiede bash ↔ ksh

Die Unterschiede zwischen der `bash` und den `ksh`s sind klein und subtil. Die `ksh` spielt unter Linux praktisch keine Rolle, daher lasse ich die Unterschiede hier weg.

Fehler

typische Anfängerfehler

- UUOC: Useless Use of a cat:
  `cat naganaga|filter`
- Falsches Quoting:
  - " mit ' verwechselt,
  - ' mit ` verwechselt.

Fortgeschrittenenfehler

- `$*` mit `$@` verwechselt.
- `export PETNAME=Oyntygraufr` in einer `sh`.
- `function f()` in der `sh`.

▹ Debugging mit `set -x` einschalten,
▹ Syntax mit `set -n` überprüfen

Die interaktive Shell

In der interaktiven Shell sind folgende Features interessant:

- Aliases
- Jobcontrol
- Tilde-Expansion
- Brace-Expansion
- Shell-History
- Programmable-Completion

interaktive Shells: Login-Shells

`.profile`	wird von der Login-Shell gelesen
`.bashrc`	wird von allen Shells gelesen.

I concluded that the rule books are written to comfort the author of the shell. For the rest of humanity, the Sunday barbecue method has a lot to recommend itself;

1) write script;
2) stir in `^`, `$`, `.`, and `\`;
3) add small amounts of \ until the flavour comes right.

Feed to guests in small portions. If entire neighbourhood is decimated, read disclaimer at front of cookbook.

Frank G. Bennett, Jr.
<fbennett@rumple.soas.ac.uk>

Literatur

[1] Steve R. Bourne. *An Introduction to the UNIX Shell*, seventh edition, ca. 1977.
7th edition UNIX manual.
`~/lib/Unix/7thEdMan/vol2/beginners.ps.gz`.

[2] *7th Edition UNIX — Summary*, seventh edition, ca. 1977.
7th edition UNIX manual.
`~/lib/Unix/7thEdMan/vol2/shell.ps.gz`.

[3] IEEE, editor. *Portable Operating System Interface (POSIX) — Part 2: Shell and Utilities (Volume 1 and 2)*. Information technology—Portable Operating System Interface (POSIX). IEEE Computer Society, 345 E. 47th St, New York, NY 10017, USA, December 1993.
ISBN 1-55937-406-3.

[4] Brian W. Kernighan and Rob Pike. *The UNIX Programming Environment*. Prentice–Hall International, Inc., Eaglewood Cliffs, New

Jersey, NJ 07632, USA, 1984.
ISBN 0-13-937699-2 (hardcover),
0-13-937681-X (paperback).

[5] Evan Schaffler and Mike Wolf. The UNIX Shell As A Fourth Generation Language. Technical report, Revolutionary Software, Inc, ca. 1988.
`/usr/share/doc/nosql/4gl.ps.gz`.

[6] Jeffrey E. F. Friedl. *Mastering Regular Expressions*. O'Reilly & Associates, Inc., 103 Morris Street, Suite A Sebastopol, CA 95472, USA, first edition, January 1997.
ISBN 1-56592-257-3.

[7] Tom Christiansen. Csh Programming Considered Harmful. Technical report, University of Colorado at Boulder, October 1993.
Classic paper why not to program in csh.
`/var/local/ftp/pub/Usenet/answers/unix-faq/shell/csh-whynot`.