Bash Debugger

Alexander Mayer,
Techn. Universität München

Die Shellprogrammierung hat keinen guten Ruf. Die Vorurteile lauten, daß sie eine Spielwiese für »Quick and Dirty Hacks« sei, sie sei schwierig in der Handhabung, zumal die Fehlersuche mangels Debugger oder Entwicklungsumgebung sehr komplex werden kann. Das stimmt aber so nicht. Für die Bash gibt es mittlerweile sogar einen Debugger.

Für das Debugging von Shellskripten gibt es bereits seit langer Zeit eine Reihe von Lösungsrezepten, die sich in der Praxis als ganz praktikabel herausgestellt haben. Als erstes läßt der versierte Shell-programmierer einen groben Syntaxprüflauf über das Skript. Das ermöglichen fast alle Shells, auch die Bash, mit dem Kommandozeilenparameter *-n*:

```
$ bash -n skript
```

Mit dieser Angabe führt die Shell das Skript nicht aus, sondern überprüft lediglich die Syntax des Listings. Allerdings sollte man von diesem Testlauf nicht allzu viel erwarten. Erkannt werden nur Fehler, die bestimmte sprachliche Elemente der Shell betreffen. Beispielsweise ist der *test*-Befehl *[...]*, mit dem viele Vergleiche realisiert werden, aus historischen Gründen kein Sprachelement der Shell, sondern ein Shellbefehl (Builtin). Eine fehlerhafte Zeile wie

```
[ 1 = 2
```

in der die schließende eckige Klammer fehlt, fällt deshalb erst zur Laufzeit auf. Verwendet man dagegen das neuere Sprachkonstrukt *[[...]]*, ist der Syntaxcheck erfolgreich:

```
[[ 1 = 2
./skript: line 1: unexpected argument to
 conditional binary operator
./skript: line 1: syntax error near unex
pected token `='
./skript: line 1: `[[ 1 = 2'
```

Abgesehen davon ist es für die Shell sowieso unmöglich, beispielsweise *eval*-Ausdrücke vor der Ausführung des Skripts einer Prüfung zu unterziehen. Das ist aber ein Problem aller Interpreter und gilt deshalb für andere Sprachen genauso.
Der zweite Schritt dieses »handgemachten« Debuggings ist weitaus hilfreicher. Liefern einige Zeilen nicht das gewünschte Ergebnis, kann man sie mit den Zeilen

```
set -x
...
set +x
```

umrahmen. *set -x* sorgt dafür, daß die Shell jede Zeile vor ihrer Ausführung zur Fehlersuche ausgibt. Dabei sieht man unmittelbar, wie Variablen ersetzt oder Jokerzeichen aufgelöst werden. Eine Zeile wie

```
mv *.bak $HOME/.Trash
```

könnte die Shell beispielsweise mit folgender Ausgabe quittieren:

```
+ mv a.bak b.bak /home/quark/.Trash
```

Installation

Das letzte und wahrhaft allerletzte Handwerkszeug der Fehlersuche pflegen nicht nur Shellprogrammierer zu verwenden: den *echo*-Befehl. Auch damit lassen sich Variableninhalte ausgeben oder Jokerzeichen auflösen. Empfehlenswert ist diese Methode vor allem, um gefährliche Zeilen zu entschärfen, indem man ganz einfach *echo* voranstellt:

```
echo mv *.bak $HOME/.Trash
```

Erfahrene Shellprogrammierer mögen sich mit diesem Rüstzeug zufriedengeben, doch gerade für Anfänger ist normalerweise wesentlich mehr Hilfe notwendig. Beispielsweise werden viele das schrittweise Ausführen, das in einem handelsüblichen Debugger möglich ist, vermissen. Diese Lücke schließt der Bash-Debugger.
Er wurde inspiriert von einem Korn-Shellskript aus dem Buch »Learning the Korn Shell« von Bill Rosenblatt. Es nutzte die Möglichkeiten der Korn-Shell aus, um einen kleinen Debugger zu entwerfen. Leider stellte sich für Bash-Debugger-Entwickler Rocky Bernstein bei der Portierung heraus, daß die Bash die für den Debugger notwendige Funktionalität (vor allem das Shell-Signal *SIGDEBUG* oder die Zeilennumerierung von Funktionen betreffend) nicht oder nur unzureichend enthielt. Erst ab Version 2.05 konnte der Quelltext der Bash geeignet gepatcht werden. Seit der kürzlich erschienenen Version 3.0 wird der Bash-Debugger schließlich offiziell unterstützt, so daß nur noch wenige Veränderungen am Bash-Quelltext für den Einbau des Debuggers notwendig sind.
Der Bash-Debugger ist momentan

offiziell in der Betaphase, macht aber bereits einen ganz stabilen Eindruck. Auf Grund fehlender Unterstützung durch gängige Distributionen ist man bei der Installation des Debuggers leider noch auf sich allein gestellt. Man benötigt zwei Pakete: die Bash 3.0 [2] selbst im Quelltext und das zur Bash-Version passende bashdb-Paket [1], für die 3.0 ist das *bashdb-3.00-0.01.tar.gz*. Als erstes müssen natürlich die Pakete ausgepackt werden:

```
$ tar zxf bash-3.0.tar.gz
$ tar zxf bashdb-3.00-0.01.tar.gz
```

Nun gilt es, die für den Bash-Debugger notwendigen Patches in die Bash-Quellen einzufügen:

```
$ cd bash-3.0
$ patch -p1 < ../bashdb-3.00-0.01/\
patches/bash-3.00.patch
```

Befehl	Beschreibung
q (quit)	Debugger beenden.
h (help)	Hilfe ausgeben.
e *Kommando*	Bash-Kommando ausführen.
b *Zeilennummer* if *let-Ausdruck*	An Haltepunkt nur dann stoppen, wenn angegebene Bedingung erfüllt ist.
disp *Kommando*	Kommando als Watchpoint setzen, *disp echo $variable* entspricht somit *W variable*.
a *ZeileKommando*	Kommando bei angegebener Zeile ausführen.
L	Zeige alle Watchpoints.
T	Zeigt einen Stack-Trace.
ret	Führe den Rest der aktuellen Funktion oder des aktuellen Skripts aus.
finish	Führe den Rest des Skripts aus.

Tabelle 1: Weitere interessante Befehle

Data Display Debugger

Das Kompilieren der Bash geschieht auf übliche Weise, allerdings muß dem *configure*-Skript eine Option übergeben werden, die die Bash debuggerkompatibel macht:

```
$ ./configure --enable-debugger
$ make
$ su -c 'make install'
```

Es ist zu beachten, daß – wenn man sonst keine Pfadangaben macht, – sich nun im Verzeichnis */usr/local/bin* eine neue Kopie der Bash 3.0 befindet. Um die neue Bash von der eventuell vorhandenen Bash zu unterscheiden, solle man beispielsweise als *root* die Datei *bash* in *bash3* umbenennen oder die Reihenfolge der Verzeichnisse in *PATH* ändern.
Anschließend ist der Bash-Debugger bereit, kompiliert zu werden. Auch hier muß dem *configure*-Skript ein Parameter übergeben werden:

```
$ cd ../bashdb-3.00-00.1
$ ./configure --datadir=/usr/local/lib
$ make
$ su -c 'make install'
```

Die Option *--datadir* ist in der aktuell verfügbaren Version notwendig, um einen Programmfehler zu umgehen. Ohne diesen Workaround würde der Bash-Debugger seine Skripten in */usr/local/share* ablegen, obwohl die Bash in */usr/local/lib* nach ihnen sucht. Auf eine Anfrage der freeX antwortete Rocky Bernstein, daß das nächste Release sich automatisch eine vorhandene Kopie der Bash suchen wird. Solange jedoch die nächste offizielle Version noch nicht erschienen ist, wird dieser Bug vermutlich noch bestehen bleiben.
Den Bash-Debugger bedient man üblicherweise über ein shell-ähnliches Prompt. Aber wer den umfangreichen und sehr beliebten Data Display Debugger (DDD) [3] kennt, kann auch in dessen Bedienungsumgebung Bash-Skripte debuggen, vorausgesetzt daß mindestens DDD 3.3.7 installiert ist:

```
$ ddd --bash skript
```

bashdb läßt sich auch innerhalb des Emacs-Debuggers *gud* verwenden und durch Eingabe der Tastenkombination *[META]-[x] bashdb* (für »Meta« ist häufig die [Esc]-Taste konfiguriert) starten. Da aber letztlich beide GUIs auf dem nativen Interface des Bash-Debuggers basieren, beschränkt sich dieser Beitrag auf die unmittelbare Steuerung über die Kommandozeile. Prinzipiell gibt es zwei Möglichkeiten, den Bash-Debugger zu verwenden. Man kann entweder das Debugger-Skript *bashdb* direkt aufrufen oder die bash mit aktiviertem Debugger aufrufen:

```
bashdb Skript Argument(e)
bash --debugger Skript Argument(e)
```

Schrittweise Überprüfung der Funktionen

Es ist empfehlenswert, die zweite Variante zu verwenden, da sie in den meisten Fällen funktioniert. Nachteilig ist lediglich, daß es nicht möglich ist, dem Debugger Parameter zu übergeben. Nur in diesem Ausnahmefall sollte man *bashdb* direkt aufrufen.
Um die wichtigsten Funktionen des *bashdb* zu zeigen, wird er mit dem kleinen Beispielskript *mvs* (Listing 1) aufgerufen. Dieses Skript arbeitet wie der *mv*-Befehl, überschreibt aber keine Dateien im Zielverzeichnis, statt

dessen werden vorhandene Dateien umbenannt:

```
$ bash3 --debugger mvs a b verz
(/home/quark/scripts/mvs:10):
10:     ziel=$(getdir "$@")
bashdb<0>
```

Als erstes gibt der Debugger den Namen des Skripts sowie die Position der ersten Zeile, die ausgeführt wird, aus. Dahinter folgt der Inhalt dieser Zeile. Das Skript *mvs* wird aber vorerst nicht ausgeführt, vielmehr wartet *bashdb* auf eine Anweisung. Um das Skript zu studieren, muß Zeile für Zeile einzeln überprüft werden. Dafür gibt es den Befehl *s* (step), der genau eine Zeile ausführt. In diesem Fall wird aber ein anderer Befehl aufgerufen: Die aktuelle Zeile ruft die Funktion *lastarg* auf, *s* würde deshalb in diese Funktion springen. Geht man davon aus, daß diese Funktion funktioniert, kann man auch den Befehl *n* (next) aufrufen. Er führt Funktionen ohne Interaktion aus und kehrt bei der Zeile nach dem Aufruf wieder zurück:

```
bashdb<0> n
(/home/quark/scripts/mvs:12):
12:     for datei; do
```

Um zu überprüfen, daß der Variablen *ziel* tatsächlich die Zeichenkette *verz* (die das letzte Argument ist) zugewiesen wurde, bietet sich der Befehl *p* (print) an:

```
bashdb<1> p $ziel
verz
```

Beobachtungspunkte

In der nun folgenden Schleife, die für alle übergebenen Argumente ausgeführt wird, kann man die Variable *datei* natürlich stets mit *p* abfragen. Es geht aber auch einfacher: Der Befehl *W variable* (watchpoint) ermöglicht es, eine Variable zu beobachten. Nach dem Aufruf *W datei* an dieser Stelle beschwert sich aber der Debugger mit dem Hinweis, daß die Variable *datei* nicht vorhanden sei. Da die Variable erst innerhalb der Schleife gültig ist, ist also erst einmal ein Aufruf von *s* notwendig:

```
# liefert letztes Argument
function lastarg {
  typeset last
  for arg; do
    last=$arg
  done
  echo $last
}

ziel=$(lastarg "$@")

for datei; do
  # falls es die Datei am Ziel bereits
  # gibt=> umbenennen
  if [[ -f $ziel/$datei ]]; then
    mv "$ziel/$datei" "$ziel/
${datei}.$(date)"
  fi

  if [[ $datei != $ziel ]]; then
    mv "$datei" "$ziel"
  fi
done
```

Listing 1: mvs verschiebt Dateien in ein Verzeichnis und rettet Dateien, die eventuell überschrieben würden, durch Umbenennen

```
bashdb<2> W datei
Can't set watch: no such variable datei.
bashdb<3> s
(/home/quark/scripts/mvs:14):
14:     if [[ -f $ziel/$datei ]]; then
bashdb<4> W datei
 0: ($datei)==a arith: 0
```

Ab nun meldet sich *bashdb*, sobald sich der Wert von *$datei* ändert. *arith:0* informiert übrigens darüber, daß der Wert nicht als mathematischer Ausdruck, wie er zum Beispiel beim *let*-Befehl erwartet wird, gedeutet werden kann. *arith:1* bedeutet das Gegenteil.

Angenommen, man möchte nicht alle Zeilen der Schleife zeilenweise überprüfen und sich statt dessen auf die Zeile konzentrieren, in der eine bereits vorhandene Datei umbenannt wird. Dazu setzt man einen sogenannten Haltepunkt im Skript. Um die Zeilennummer der besagten Stelle ausfindig zu machen, ruft man *l* (list) auf. Diese Anweisung gibt die Skriptzeilen ab der aktuellen Position sowie die Zeilennummern aus:

```
bashdb<5> l
 14:==>if [[ -f $ziel/$datei ]]; then
 15:   mv "$ziel/$datei" "$ziel/${datei}
.$(date)"
 16:   fi
 17:
 18:   if [[ $datei != $ziel ]]; then
 19:   mv "$datei" "$ziel"
 20:   fi
 21:   done
 22:
```

Die gesuchte Zeilennummer lautet 15. Man setzt einen Haltepunkt an Zeile 15 mit dem Befehl *b zeile* (breakpoint):

```
bashdb<6> b 15
Breakpoint 1 set in file /home/quark/scr
ipts/mvs, line 15.
```

Mit dem Befehl *c* (continue) läßt sich das Skript nun ohne Zwischenstopp bis zum nächsten Haltepunkt ausführen:

```
bashdb<7> c
Breakpoint 1 hit (1 times).
(/home/quark/scripts/mvs:15):
15:     mv "$ziel/$datei" "$ziel/
${datei}.$(date)"
```

Man sieht bereits an den hier beschriebenen Befehlen, daß der Bash-Debugger ein sehr hilfreiches Tool sein kann. Allerdings sollte man sich stets bewußt sein, daß auch der Debugger nichts anderes als ein Shellskript und deshalb anfällig für Fehleingaben ist, beispielsweise wenn man einen fehlerhaften Watchpoint-Ausdruck angibt. Sieht man von dieser systembedingten Schwäche und der naturgemäßen Instabilität der Betaversion ab, wird der Bash-Debugger sicherlich so manchem Shellprogrammierer eine große Hilfe sein. ◆

Links

[1] Bash-Debugger: *http://bashdb.sf.net*

[2] Bash: *http://www.gnu.org/software/bash/bash.html*

[3] DDD: *http://www.gnu.org/software/ddd*