Expressie	Is waar als	Voorbeeld	
		Waar	Onwaar
string1 = string2	de strings identiek aan elkaar zijn	a = a	a = b
string1 != string2	de strings van elkaar verschillen	a != b	a != a
string1 < string2	string1 alfabetisch voor string2 staat (hoofdletters komen als eerste)	a < b	a < a
string1 > string2	string1 alfabetisch na string2 staat (hoofdletters komen als eerste)	B < a	b < a

Tabel III: Strings vergelijken

> **PRO-TIP: Kijk eens naar het afhandelen van opties met getops en hoe de for-loop werkt**

In tabel III zie je een overzicht voor strings. Als voorbeeld passen we het script aan voor de correcte weergave van distributienamen:

```
while read woord ; do
      woord=${woord,,}
      if [ $woord = suse ]
; then
            echo ${woord^^}
      else
            echo ${woord^}
      fi
  < "$lijst"
```

Met het volgende bestand:

```
cat lijst.txt
mINT
SuSe
```

Hierbij krijg je het volgende resultaat:

```
./mijn_script lijst.txt
Mint
SUSE
```

Bij het testen op ongelijkheid gaat het echter mis:

```
if [ a < b ] ; then echo
Kleiner ; fi
bash: b: No such file or
directory
```

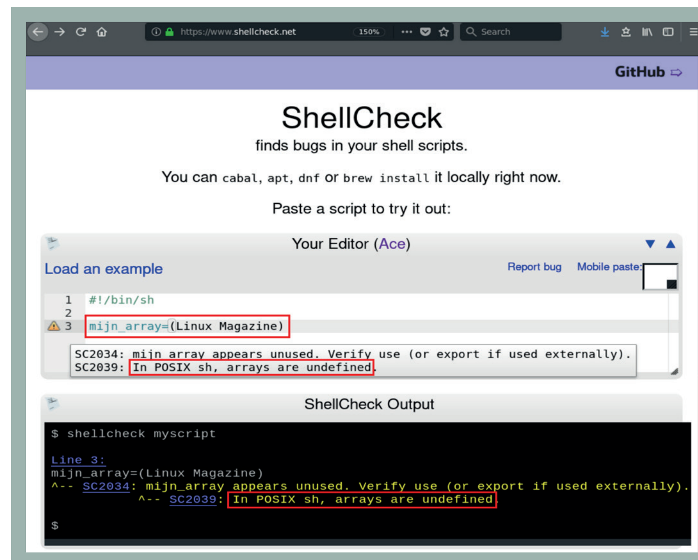
De shell voert redirectie uit om bestand b in te lezen. Dat bestand kan hij niet vinden en faalt. Met een escape (\) is dit te verhelpen:

```
if [ a \< b ]
```

Nu interpreteert de shell het-kleiner-dan teken niet meer als redirectie en geeft de vergelijking **a < b** letterlijk aan het test-commando door. Het alternatief is dubbele vierkante haken:

```
if [[ a < b ]]
```

De openingshaken **[[** zijn in tegenstelling tot **[** geen commando. Ze betekenen voor de shell dat ertussen een vergelijking staat. Ze functioneren daarom ook voor getallen:



Afbeelding 4: Script is niet conform POSIX

```
if [[ 0 -lt 9 ]]
```

En bestanden:

```
if [[ -f $lijst ]]
```

De dubbele haken zijn eveneens te gebruiken bij de while-loop.

POSIX

In deel 1 meldden we al dat de gebruikte shell invloed heeft op je script. Als voorbeeld nemen we dash. Bij Debian is die standaard aanwezig, maar in het geval van Fedora moet je het pakket dash zelf installeren. Start dash op:

```
dash
```

Voer het commando van hierboven uit om een array te definiëren:

```
mijn_array=(Linux Magazine)
dash: 1: Syntax error: "("
unexpected
```

Je ziet dat dash geen arrays ondersteunt. Andere shells hebben weer andere beperkingen. Hoe zorg je dan dat je script op andere systemen werkt? Dat doe je door je aan POSIX te houden.

POSIX is een standaard, waaraan alle huidige UNIX- en Linux-systemen zich conformeren. Die bepaalt onder meer wat de shell minimaal ondersteunt. Daaronder vallen geen arrays. Een POSIX shell hoeft dat dus niet te implementeren, zoals je zag bij dash. Dat mag natuurlijk wel, maar dat is dan extra functionaliteit bovenop POSIX.

Als je je dus beperkt tot POSIX, werken je scripts eigenlijk in iedere shell. Door te testen in dash weet je al behoorlijk zeker dat je script voldoet aan de POSIX norm. Een echte POSIX shell is Posh. Die biedt alleen maar de POSIX functionaliteit. Posh vind je echter alleen bij Debian en afgeleiden.

Een handige tool is verder ShellCheck (zie afbeelding 4). Als die niet voor jouw Linux-distributie beschikbaar is, gebruik je de online variant. Hiermee controleer je scripts op fouten. Hierbij zie je onder andere of je script de POSIX standaard te buiten gaat.

Maar POSIX dicteert tevens welke commando's beschikbaar moeten zijn. Daaronder vallen bijvoorbeeld niet het welbekende which of mktemp. Op sommige systemen ontbreken die of gedragen zich anders dan je gewend bent. Dat geldt ook voor opties. Het commando '**grep -w woord**' hoeft niet te werken, omdat POSIX de optie **-w** niet eist voor **grep**.

Bovenstaande tests helpen je hierbij niet. Uiteindelijk is alleen de documentatie van POSIX leidend. Daar staat precies beschreven welke commando's, opties en shell functionaliteit je mag verwachten.

Tot besluit nog iets over de shebang:

```
#!/bin/bash
```

Bash hoeft niet aanwezig te zijn. Wel is op een POSIX systeem altijd **sh** bekend. Op een Linux systeem is dat gewoon een symlink naar Bash of dash. Gebruik daarom de volgende shebang voor POSIX:

```
#!/bin/sh
```

Je ziet ook geregeld een variant hierop:

```
#!/usr/bin/env sh
```

Het commando **env** krijgt als argument **sh** mee. Hij zoekt die executable in $PATH en voert het dan uit. Op sommige UNIX systemen is /bin/sh namelijk geen POSIX shell. De POSIX variant staat niet in een standaardpad, maar vind je wel via de omgevingsvariabele $PATH. Door het commando **env** krijg je zodoende wel de POSIX shell te pakken.

EN VERDER

Met de informatie uit deze workshop kom je goed beslagen ten ijs. Je weet nu hoe je allerlei shell functionaliteit gebruikt. Ga van hieruit verder op onderzoek. Kijk bijvoorbeeld eens naar het afhandelen van opties met getops, hoe de for-loop werkt of wat een here-document is.

LINKS

Officiële handleiding
gnu.org/software/bash/manual

POSIX standaard
pubs.opengroup.org/onlinepubs/9699919799/

Posh
packages.debian.org/sid/posh

ShellCheck
shellcheck.net

34-37-LM418 - Workshop Shel Scripting Deel 2.indd 37 23-08-18 14:19