Korte uitleg: Haakjes en Aanhalingstekens

Wie naar een wat ingewikkelder `bash` script kijkt ziet een verwarrende hoeveelheid van allerlei soorten haakjes en aanhalingstekens. Die verwarring is normaal gezien de vele mogelijkheden, maar het begrip wordt er vaak niet beter van als de schrijver van het script er geen consequent gebruik van maakt. Hij heeft vaak aanhalingstekens toegevoegd "tot het werkte" en begrijpt niet waarom. Daarom laat hij ze elders weg, tot het daar ook mis gaat.

Bij haakjes, aanhalingstekens, maar ook bij alle tekens die een speciale betekenis hebben voor de shell, moet je nooit vergeten dat ze soms in bepaalde constructies weer een andere betekenis hebben. Bij die andere constructies denken we bijvoorbeeld aan "patterns" (voor het zoeken op strings met een bepaalde inhoud zoals: begint met een hoofdletter en eindigt op `pht`), rekenkundige expressies en meer.

Accolades

```
{ ..... }
```

Om shell commando's te groeperen. Dat gebeurt in ieder geval om alle commando's van een functie te groeperen:

```
doeiets() {
echo "Wat zullen we eens doen"
ls -l
cat /etc/fstab
}
```

Ook handig is om te groeperen zodat je in één keer de uitvoer kan redirecten. Inplaats van:

```
echo "Wat zullen we eens doen" >/tmp/wzwed-uitvoer
ls - >>/tmp/wzwed-uitvoer
cat /etc/fstab >>/tmp/wzwed-uitvoer
```

dus

```
{
echo "Wat zullen we eens doen"
ls -l
cat /etc/fstab
} >/tmp/wzwed-uitvoer
```

of de variant:

```
{ echo "Wat zullen we eens doen" ; ls -l ; cat /etc/fstab ; } >/tmp/wzwed-uitvoer
```

Let in het laatste voorbeeld op de `;` en de "white space" voor de `}`. De `}` moet als een apart commando gezien worden.

Ronde haken

```
( ..... )
```

Ook hier mee kun je commando's groeperen. Alleen deze groep wordt in een aparte shell (dus een apart proces) uitgevoerd. In veel gevallen is dat nergens voor nodig en moet er dus voor niets een apart `bash` proces worden gestart. Omdat dit een kind proces is kunnen er geen waardes naar de ouder worden doorgegeven:

```
henk@boven:~/test> A=aap
henk@boven:~/test> echo ${A}
aap
henk@boven:~/test> ( echo "We zetten A op noot" ; A=noot ; echo ${A} )
We zetten A op noot
noot
henk@boven:~/test> echo ${A}
aap
henk@boven:~/test>
```

Dubbele ronde haken

```
(( ..... ))
```

Hierbinnen kun je berekeningen met gehele getallen maken:

```
henk@boven:~/test> (( I = 6 ))
henk@boven:~/test> echo $I
6
henk@boven:~/test> (( I += 1 ))
henk@boven:~/test> echo $I
7
henk@boven:~/test>
```

Zoals je ziet bestaan er speciale rekenkundige operatoren zoals `+=`, die komen uit de programmeertaal C.

Achterover liggende aanhalingstekens
Dollar met ronde haken

```
` ..... `
```

```
$( ..... )
```

Hiertussen kun je één of meer commando's zetten. Wat er uit die commando's komt (op stdout) wordt hier ingevoegd:

```
henk@boven:~/test/bestanden> A=$(ls -l verf.jpeg)
henk@boven:~/test/bestanden> echo $A
-rw-r--r-- 1 henk wij 157943 10 mei 20:38 verf.jpeg
henk@boven:~/test/bestanden>
```

De vorm met de "back-quotes" `` `.....` `` is de oudste en zie je helaas nog veel gebruikt. De vorm `$( ..... )` is veel makkelijker te lezen en kan bovendien genest worden:

```
henk@boven:~/test> A=$(getent hosts $(hostname))
henk@boven:~/test> echo $A
10.0.0.154 boven.henm.xs4all.nl boven
henk@boven:~/test>
```

Dollar met accolades

```
${...}
```

Dit is parameter substitutie. De waarde van een variabele wordt hiervoor in de plaats gezet. Er zijn allerlei varianten mogelijk zoals `${A##*-}` en die vereisen dat de `{` en `}` aanwezig zijn. Ook als er geen "white space" achter staat zijn ze nodig, maar in de eenvoudigste vorm kan je de `{` en `}` weglaten:

```
henk@boven:~/test> A=lala
henk@boven:~/test> echo $A
lala
henk@boven:~/test> echo ${A}
lala
henk@boven:~/test> echo troe${A}
troelala
henk@boven:~/test> echo troe$A
troelala
henk@boven:~/test> echo $Atafel

henk@boven:~/test> echo ${A}tafel
lalatafel
henk@boven:~/test>
```

Verklaar waarom het één na laatse commando een lege regel oplevert.

In een serieus script gebruik ik ze eigenlijk altijd. Iets meer werk om ze te maken, maar ik kan het resultaat beter begrijpen.

Overigens is hier ook nesten mogelijk en dan is het wel zeker dat je ze

moet gebruiken, al was het maar om te begrijpen wat je maakt.

Vierkante haken
Dubbbele vierkante haken

```
[ ..... ]
[[ ..... ]]
```

Oorspronkelijk was er het programma `test`. Omdat het veel in scripts werd gebruikt en omdat dat dat er leuk uitzag hebben ze dat programma een tweede naam gegeven: `[`. En dat bestaat nog steeds:

```
henk@boven:~/test> ls -l $(which '[')
-rwxr-xr-x 1 root root 39760 23 jan 14:04 /usr/bin/[
henk@boven:~/test>
```

Echter, er worden veel tests uitgevoerd in scripts en om het iedere keer opstarten van een kind proces te vermijden is `[` al lang geleden ingebouwd in de shell als build in command. In dat geval vereist de shell ook dat het commando met een `]` wordt afgesloten.

De Korn shell introduceerde een betere implementatie, maar om de oude constructie compatibel te houden moest een nieuwe notatie worden gekozen. En die werd `[[ ..... ]]`.

Overigens worden `[` en `]` ook gebruikt bij arrays in de shell.

```
henk@boven:~/test> Array[1]="Hoera hoera"
henk@boven:~/test> echo ${Array[1]}
Hoera hoera
henk@boven:~/test>
```

En binnen Regular Expressions:

```
henk@boven:~/test> N=1
henk@boven:~/test> [[ ${Array[N]} = [A-Z]* ]] && echo "Begint met een kapitaal"
Begint met een kapitaal
henk@boven:~/test>
```

Oftewel: drie verschillende soorten [gebruik in één commando.

Enkele aanhalingstekens

```
' ..... '
```

De zogenaamde "single quotes". Als je een stuk van een commando wil afschermen voor interpretatie door de shell kun je dat stuk tussen single

quotes zetten. Alle tekens tussen `'` en `'` worden niet als speciale tekens beschouwd, maar gewoon als tekst. Dus:

```
henk@boven:~/test> echo ${Array[1]}
Hoera hoera
henk@boven:~/test> echo '${Array[1]}'
${Array[1]}
henk@boven:~/test>
```

Dubbele aanhalingstekens

```
" ..... "
```

De zogenaamde "double quotes". Doet hetzelfde als de single quotes behalve dat `$`, `` ` `` (back quoute) en `\` (escape) hun betekenis blijven behouden. Wordt veel gebruikt in scripts en met intikken, omdat parameter substitutie blijft werken en "white space" interpretatie (als woordscheiding) door de shell wordt onderdrukt.

Achteroverliggende schuine streep

```
\
```

Dat is geen begin-eind combinatie, maar een teken alleen: de "escape". Het doet hetzelfde alsof het volgende teken tussen `'` en `'` zou staan. Is dus handig om een enkel teken (veelal een spatie) echt als spatie te laten gelden:

```
henk@boven:~/test/bestanden> ls -dl gewone\ file
drwxr-xr-x 2 henk wij 4096 12 mei 12:43 gewone file
henk@boven:~/test/bestanden>
```

Ook veel gebruikt om lange commando's over meer regels te verdelen. Als hij aan het eind van de regel staat wordt de special betekenis van de New-line (einde commando) uitgeschakeld:

```
henk@boven:~/test/bestanden> ls -d\
> l verf.gif
-rw-r--r-- 1 henk wij 157943 10 mei 20:39 verf.gif
henk@boven:~/test/bestanden>
```

Let op hoe de shell in dit interaktieve geval een afwijkende prompt afgeeft om aan te geven dat het commando niet compleet is.