

Reguliere expressies

1. Introductie

Reguliere expressies vormen een specificatie-taal die een rol speelt als je in een grote hoeveelheid tekst een bepaald stukje zoekt. Het zoekwerk laat je door je computer doen, maar die wil een nauwkeurige specificatie van wát hij precies moet zoeken. Daarvoor gebruik je een notatie in *reguliere expressietaal*. Om de rest van dit artikel wat eenvoudiger te maken zullen we 'reguliere expressie' vaak afkorten tot *regex*.

Reguliere expressies behoren tot de meest krachtige mechanismen die binnen UNIX worden gebruikt. Reguliere expressies worden overal gebruikt waar in tekstbestanden wordt gezocht. Bijvoorbeeld: alle editors kennen een *search-replace* operatie. Het *search*-gedeelte van zo'n operatie noteer je altijd als reguliere expressie.

Om reguliere expressies hier te demonstreren maken we gebruik van een tekstbestand met de naam `woordjes`, dat het volgende lijstje woorden bevat:

```
appel
aardappel
pennen
aardnoten
piet.
lopen
```

← dit is een volledig lege regel

```
apenoot
pellen.
pinda
mat
maria
pint
```

Behalve een tekstbestand om in te zoeken hebben we ook een zoek-commando nodig als demonstratie-vehikel. UNIX heeft veel mogelijkheden, en wij kiezen het commando `grep`. De gebruiker geeft aan `grep` bij het starten een zoekstring (in *regex*-notatie) plus de namen van een of meer tekstbestanden. Het commando

zoekt dan in die bestanden naar de regels waarin een stukje tekst voorkomt dat “past” op onze regexp. Die regels worden dan getoond als output. In regexp-jargon wordt dat ‘passen op’ een *match* genoemd: `grep` toont alle tekstregels waarbinnen een stukje tekst voorkomt dat op de gegeven regexp matcht.

Je moet op dit punt goed onderscheid maken tussen enerzijds de reguliere expressies als specificatietaal, en anderzijds het specifieke gedrag van een bepaald commando dat we als demonstratie-vehikel gebruiken. Het ‘toon de omhullende tekstregel’ is specifiek voor het commando `grep`. Als we als demonstratie-vehikel een editor hadden gebruikt met een search/replace opdracht dan was het gevonden stukje tekst vervangen door een ander. Maar dit artikel gaat over het zoeken: hoe specificeer je wat je wilt (laten) zoeken, en hoe gaat dat zoeken precies in zijn werk. Alle commando’s die regexp-notatie ondersteunen, en die we dus als demonstratie-vehikel kunnen gebruiken, hebben dat stuk functionaliteit gemeenschappelijk.

2. Drie generaties regexp

De UNIX reguliere expressies zijn ontworpen in de eerste helft van de jaren ’70. Daarna heeft de tijd niet stilgestaan, en zijn nieuwe ideeën toegevoegd. Tegenwoordig kun je *drie* generaties van reguliere expressies tegenkomen:

- BRE’s: de *Basic* Regular Expressions. Dit is de basisvorm die in de meeste commando’s wordt gehanteerd.
- ERE’s: de *Extended* Regular Expressions. Ze hebben meer mogelijkheden dan de BRE’s, maar zijn niet helemaal upward compatible. Dus je moet bij een commando wel weten of dat BRE’s of ERE’s ondersteunt. Veel commando’s doen als default BRE’s, maar met een vlaggetje kun je ze naar ERE-support schakelen. De Posix-standaardisatie heeft preciese specificaties vastgelegd voor zowel BRE’s als ERE’s. Maar als je ergens de term ‘Posix-regular expressions’ tegenkomt dan bedoelt men meestal de ERE’s. Bijvoorbeeld de programmeertaal PHP bevat daarvan vele voorbeelden.
- De ontwerper van de programmeertaal Perl heeft ook veel nieuwe ideeën op regular-expressie gebied ingebracht. De *Perl* Reguliere Expressies vormen dus de jongste van de drie generaties. Ook voor deze generatie geldt dat als een commando ’m ondersteunt, daar meestal een apart vlaggetje voor moet worden meegegeven.

In dit artikel beginnen we met de Basic Regular Expressions.

3. Basisvorm

De eenvoudigste reguliere expressies zijn letterlijke teksten: als je een eenvoudig stukje tekst zoekt dan schrijf je dat recht-toe-recht-aan op. Als we zoeken naar a dan vindt `grep` alle regels met een letter a erin:

```
$ grep 'a' woordjes
appel
aardappel
aardnoten
apenoot
pinda
mat
maria
```

Als je zoekt naar aa dan vindt grep alle regels waarin aa voorkomt:

```
$ grep 'aa' woordjes
aardappel
aardnoten
```

Een regexp zoals `appel` levert alle regels op waarop het `appel` voorkomt; dit kan uiteraard ook onderdeel van een groter woord zijn (hier `aardappel`). Dus strikt genomen zoekt grep niet naar `appel` als woord, maar naar `appel` als *string*.

Vrijwel alle tekens waarmee je een reguliere expressie opschrijft worden gezien als “letterlijke letters”. Maar een paar (lees-)tekens hebben een bijzondere betekenis. We kunnen daarmee extra eisen stellen.

3.1 Verankering in reguliere expressies

Bijvoorbeeld kunnen we grep laten zoeken naar regels die *beginnen met* een a. Hiervoor laten we onze reguliere expressie beginnen met het speciale symbool `^`. De reguliere expressie `^a` wordt vertaald in: zoek naar een a die aan het begin van een regel staat. Dus:

```
$ grep '^a' woordjes
appel
aardappel
aardnoten
apenoot
```

Op dezelfde manier betekent een `$` aan het einde van een regexp dat het gezochte aan het einde van de regel moet staan. Daar mag dan zelfs geen spatie meer achter staan. Dus we schrijven als regexp `a$` om de regels te vinden die *eindigen* op een a:

```
$ grep 'a$' woordjes
pinda
maria
```

Als aa twee op elkaar volgende letters a zoekt, dan moet `^$` een lege regel vinden. Immers op een lege regel volgt het einde meteen op het begin: En inderdaad:

```
$ grep '^$' woordjes  
←dit is de lege regel uit ons bestand
```

Het vasthaken van een reguliere expressie aan het begin en/of het einde van de regel heet *verankering*.

- | `^` voorop in een regexp betekent dat het te zoeken patroon aan het begin van een regel moet staan.
- | `$` achteraan een regexp betekent dat het te zoeken patroon aan het eind van een regel moet staan.

3.2 Meerdere tekens als alternatief op één positie

Kunnen we `grep` nu ook vragen om alle plaatsen waar òf een a òf een l voorkomt? Dat kan door gebruik te maken van een constructie die een keuze uit verschillende tekens mogelijk maakt. Als we schrijven `[al]` dan zoeken we daarmee één teken, maar dat teken mag zowel een a als een l zijn. Dus tussen de blokhaken sommen we alle voor ons acceptabele letters op. Zoeken naar een a of een l gaat als volgt:

```
$ grep '[al]' woordjes  
appel  
aardappel  
aardnoten  
lopen  
apenoot  
pellen.  
pinda  
mat  
maria
```

We zien dat sommige woorden zowel een a als een l bevatten, maar ieder gevonden woord bevat minstens één van de twee letters.

Als we laten zoeken naar `[aeo][aou]` dan zoeken we naar twee *opeenvolgende* letterposities, waarbij we op de eerste een a of e of o accepteren, en op de tweede positie een a of o of u. Merk dus op dat een regexp-specificatie positie voor positie aan elkaar schakelt wat we precies willen zoeken.

grep vindt alle regels eindigend op een a of een l met:

```
$ grep '[al]$\' woordjes  
appel  
aardappel  
pinda  
maria
```

3.3 De complementaire verzameling

We het nog algemener maken: we willen bijvoorbeeld zoeken naar alle regels die *niet* op een a eindigen. Dan zouden we tussen twee blokhaken een uitputtende lijst van alle mogelijke tekens moeten maken die geen a zijn. Dat is veel werk; bovendien is het lastig om zeker te weten of we wel alle tekens hebben gehad. Dus moet er een notatie zijn voor een *niet*-a. De manier om dat op te geven is `[^a]`; iets nauwkeuriger: een willekeurig teken dat alleen niet de waarde van de tekens tussen de blokhaken mag hebben (hier dus *niet* een a). Pas op: het dakje kan in reguliere expressies dus twee verschillende dingen betekenen: aan het begin van een regexp is het het begin-anker symbool, en vlak na een blokhaak-openen geeft hij aan de rest van wat tussen die blokhaken staat een “niet dezen” betekenis.

Dus alle regels die *niet* eindigen op een a vind je via:

```
$ grep '[^a]$\' woordjes  
appel  
aardappel  
pennen  
aardnoten  
piet.  
lopen  
apenoot  
pellen.  
mat  
pint
```

Let op: de lege regel vinden we niet, want die heeft geen enkele letter op de regel dus zelfs geen niet-a.

Nog een voorbeeld: alle regels die met een niet-a beginnen zijn:

```
$ grep '^[^a]' woordjes
pennen
piet.
lopen
pellen.
pinda
mat
maria
pint
```

Het eerste ^ is beginanker: hij matcht het begin van de regel. Het tweede ^ keert de betekenis van de blokhaken-verzameling om. We moeten wel opletten, want `grep '[^a]'` doet niet het omgekeerde van `grep 'a'`. Kijk maar:

```
$ grep '[^a]' woordjes
appel
aardappel
pennen
aardnoten
piet.
lopen
apenoot
pellen.
pinda
mat
maria
pint
```

Iedere regel bevat immers wel een niet-a, behalve die lege regel natuurlijk.

3.4 Eén willekeurig teken

We kunnen ook aangeven dat op een positie een *willekeurig* teken mag staan. Dat is dus ruimer dan een beperkte verzameling opgesomd tussen blokhaken.

Bijvoorbeeld: zoek naar een rijtje van vier letters waarvan de eerste en de laatste een a moeten zijn. Tussen die twee in accepteren we *elk willekeurig* teken. In reguliere expressies schrijven we dat als `a..a`, waarbij de punt (.) de notatie voor “hier een willekeurig teken” is. Bij ons voorbeeldbestand vinden we dan met `grep`:

```
$ grep 'a..a' woordjes
aardappel
maria
```

Bij `aardappel` past namelijk `arda`. Aangezien het jargon in het Engels is, hoor je ook vaak: de reguliere expressie `a..a` *matcht* op de string `aria` in het woord `maria`.

3.5 Speciale tekens ontkrachten

En hoe moeten we de regels zoeken die eindigen op een punt? Niet met `.$` want dat toont alle niet-lege regels (regels met één teken vlak voor het einde). Hiervoor is een systematische oplossing: een teken dat een bijzondere betekenis heeft binnen reguliere expressies kunnen we die bijzondere betekenis afpakken door er een backslash `\` voor te zetten. Daarmee degraderen we dat speciale teken tot: “hier bedoelen we 'm als letterlijk dát teken”. Merk op dat door deze aanpak de backslash zelf ook een speciaal teken in de regexp-taal is geworden. Dus zoeken van één letterlijke backslash moet zo: `\\`

De regels die eindigen op een punt vinden we zo:

```
$ grep '\\.$' woordjes
piet.
pellen.
```

Merk op dat dit gebruik van backslash om een symbool te ontdoen van zijn speciale betekenis ook in de shell-taal voorkomt. De shell-taal kent ook quotes als alternatieve notatie voor ditzelfde 'ontkrachtings'-doel. De regexp-taal kent quotes voor dit doel niet.

3.6 Eén teken in reguliere expressies

Er zijn dus vier methoden om een teken in een reguliere expressie aan te geven. In volgorde van meest tot minst nauwkeurig gespecificeerd, zijn dit:

- a een teken dat letterlijk zo moet voorkomen (hier dus een a).
- `\.` ook een letterlijk teken, met name als dat teken binnen reguliere expressies iets bijzonders betekent (hier in dit voorbeeld een letterlijke punt).
- `[abc]` het gezochte teken moet er één uit het opgesomde rijtje zijn (hier dus een a of een b of een c)
- `[^0123]` we zoeken op deze plek één willekeurig teken, zo lang het maar *niet* eentje uit het opgesomde rijtje is. In dit voorbeeld mag het dus geen 0, 1, 2 of 3 zijn.
- `.` een willekeurig teken.¹

1. Een teken dat soms verwarring kan scheppen is de TAB (in de ASCII-tabel het teken met waarde 9). De TAB telt ook in reguliere expressies als één enkel teken. De TAB zorgt echter voor het schuiven van de cursor naar de volgende tabulator-stop. Op het scherm kun je daarom (afhankelijk van de positie waar de TAB “op het scherm terecht komt”) meerdere spatie-stappen zien.

Reeksen tekens tussen blokhaken mag je afkorten met een liggend streepje: bijvoorbeeld `[a-z]` betekent: een kleine letter het eindteken.

Hier moet een waarschuwing bij: de onderlinge volgorde van tekens ligt vast in de z.g. *ASCII-character set* standaard. Daarin zijn de “kleine letters” onderling opeenvolgend, evenals de “hoofdletters” en de “cijfers”. Maar gebruik nooit iets als `[a-Z]`; dat zal fout gaan omdat je daarbij aanneemt dat de hoofdletters op de kleine volgen. Gebruik in plaats daarvan `[a-zA-Z]`. Letters met accenten komen in de *ASCII-character set* helemaal niet voor. Daarover volgt later meer informatie.

3.7 De Kleene-ster

Stel nu dat we zoeken naar woorden waar een combinatie van de letters `n`, `o` en `t` voorkomt in die volgorde achter elkaar. En het aantal letters `o` maakt ons hierbij niet uit. We willen dus zoeken naar `not`, naar `noot`, naar `noot` enzovoorts.

Deze herhaling kunnen we aangeven door achter een teken een sterretje te plaatsen²: dat teken accepteren we dan *nul* of meer keren achter elkaar. We zoeken dus:

```
$ grep 'no*t' woordjes
aardnoten
apenoot
pint
```

De regel `pint` doet mee want hij bevat een `n` en een `t`, en daartussen *nul* keren een `o`. Als we specifiek dat *nul*-geval niet willen, moeten we schrijven:

```
$ grep 'noo*t' woordjes
aardnoten
apenoot
```

dat wil zeggen, minstens één `o` en eventueel meer. De eerste `o` staat op eigen benen, en de `o` daarachter hoort bij de `*`. Anders gezegd: we willen in ieder geval één `o`, en optioneel accepteren we meerdere daarachter,

Dat laatste toont de werkelijke kracht van de `*`: hij zegt dat een bepaalde component *optioneel* is. Met de `*` bouw je *optionele delen* in je regexp.

Bijvoorbeeld: toen we eerder de `$` behandelden als einde-regel anker, schreven we daarbij als waarschuwing: “de voorafgaande expressie moet aan het einde van een regel matchen; daar mag nog geen spatie achter staan”. Maar stel dat we dit willen afzwakken en eventueel spaties toch mee willen accepteren. Dan laten we onze

2. Dit sterretje is genoemd naar de wiskundige Steve Kleene, die de theoretische grondslag voor de reguliere expressies heeft bedacht: http://en.wikipedia.org/wiki/Kleene_closure

reguliere expressie eindigen op `*$` (spatie ster dollar) om uit te drukken dat vlak voor het einde van de regel *optioneel* nog spaties mogen staan.

De `*` werkt op het voorafgaande element, en dat mag een specificatie in `[...]`-vorm zijn. Dus:

```
$ grep '[aeio][aeio][aeio]*' woordjes
aardappel
aardnoten
piet.
apenoot
maria
```

vraagt om 'twee of meer' voorkomens van a,e,i,o: twee stuks `[aeio]` staan op eigen benen, en optioneel nog (nul of) meerdere `[aeio]` daarachter. Merk op dat dit niet betekent: 'twee of meer a' of 'twee of meer e' enz, maar 'twee of meer elementen die *elk voor zich* matchen op `[aeio]`'.

| * na een teken (of een veldje) betekent dat dat teken *nul* of meer keren herhaald op die plek mag voorkomen.

Stel nu opnieuw dat we alle woorden willen hebben waarin geen a voorkomt. De eis is dan dat het eerste tot en met het laatste teken een niet-a is. Dus:

```
$ grep '^[^a]*$' woordjes
pennen
piet.
lopen

pellen.
pint
```

Door de *verankering*, het `^` vooraan en de `$` aan het eind van de expressie, voorkomen we de problemen die we eerder hadden met `grep '[^a]`' We moeten nu zeker niet het sterretje vergeten, want zonder sterretje zoeken we een regel met precies één teken erop, dat ook nog eens een niet-a moet zijn. Merk op dat we ook de lege regel vinden, omdat het sterretje ook *nul* keer een niet-"a" toestaat.

Op dezelfde wijze kunnen we zoeken naar alle regels die alleen maar de letters g tot en met t bevatten:

```
$ grep '^[g-t]*$' woordjes

pint
```

Als we de lege regel niet willen hebben, wordt het:

```
$ grep '^[g-t][g-t]*$' woordjes
pint
```

Zoeken we nu naar regels die met een a beginnen en op een t eindigen. Tussen de a en de t mogen willekeurige tekens staan (en ook willekeurig veel). Daarmee komen we tot:

```
$ grep 'a.*t' woordjes
aardnoten
apenoot
mat
```

We vinden er natuurlijk te veel, want we zijn de verankering vergeten. We vinden bijvoorbeeld mat omdat we niet eisen dat de regel met een a *begint*. Beter is:

```
$ grep '^a.*t$' woordjes
apenoot
```

Hoe vinden we nu de woorden die twee maal een e bevatten? Als volgt:

```
$ grep 'e.*e' woordjes
pennen
pellen.
```

Zoekt dit alle regels met precies twee maal een e erop? Nee, kijk maar:

```
$ grep 'a.*a' woordjes
aardappel
aardnoten
maria
```

We moeten immers uitsluiten dat alle andere tekens die zelfde letter zijn. Om te weten dat we alle tekens hebben, gebruiken we verankering:

```
$ grep '^^[^a]*a[^a]*a[^a]*$' woordjes
aardnoten
maria
```

3.8 Nog meer herhalingen

De Kleene-star die volgt op een regexp zegt: van die regexp accepteer ik *nul of meer* herhalingen. Het is mogelijk om voor die *nul of meer* andere aantallen te kiezen, hoewel dat in de praktijk weinig wordt gebruikt. Het gaat met de notatie:

$$regexp\{m,n\}$$

Achter de regexp geven we met backslashes en accolades de gewenste aantallen op. De m en n zijn gehele getallen ≥ 0 en geven de gewenste *van-tot/met* range aan. Er zijn twee variaties mogelijk:

$regexp\{m, \}$ → m stuks en meer (minstens m)
 $regexp\{m\}$ → precies m stuks

Dus $regexp^*$ is hetzelfde als $regexp\{0, \}$

3.9 Character classes

We hebben gezien dat je tussen blokhaken een streepje (hyphen) tussen twee characters kunt zetten om een range aan te geven, bijvoorbeeld $[0-9]$ of $[a-zA-Z]$. Maar dit is een range volgens de ASCII-standaard, en daar zit slechts een beperkt aantal characters in. Letters met accenten (officieel: met diacritische tekens) ontbreken, evenals allerlei nationale uitbreidingen zoals de ß, de ð of de ¿.

Daarom zijn er “verzamelnamen” gemaakt, waarmee je in een klap een hele verzameling van tekens kunt aanduiden, inclusief alle non-ASCII variaties die er bestaan:

reguliere expressie	betekenis
$[:alpha:]$	een letter
$[:lower:]$	een lower case letter
$[:upper:]$	een upper case letter
$[:digit:]$	een decimaal cijfer
$[:xdigit:]$	een hexadecimaal cijfer
$[:alnum:]$	een letter of cijfer
$[:punct:]$	alle niet-letters, cijfers, control chars, “space”s
$[:graph:]$	een zichtbaar, printable char (geen spatie/tab)
$[:print:]$	alles behalve control characters
$[:cntrl:]$	een control character
$[:blank:]$	een spatie of een tab
$[:space:]$	spatie, tab, form feed, vertical tab, newline, carriage return

Deze verzamelnamen mag je alleen gebruiken tussen de regexp-blokhaken. Omdat bij de namen zelf ook blokhaken horen moet je zorgen om niet in die haken verstrikt te raken. Bijvoorbeeld:

$[[:alpha:]]2-5]$

staat voor een positie waarop een letter staat óf een van de cijfers 2 t.m. 5.

Het is belangrijk om te beseffen dat we hiermee méér dan de ASCII-verzameling aankunnen, maar we blijven wel binnen het Latijnse alfabet. Uitbreiden naar andere alfabetten, of zelfs naar alfabet-loze talen (Chinees, Japans, Koreaans) is een specialisme apart.

3.10 Het huishoudelijk reglement

Als de expressies ingewikkelder worden krijg je misschien verschillen in interpretatie van hun betekenis. Er is een soort “huishoudelijk reglement” om de eenheid te bewaren:

- Een regexp matcht alleen maar *binnen* een regel. Je zult dus nooit een match krijgen die bestaat uit de staart van een regel, en dan doorlopend in de kop van de volgende regel. Anders gezegd: als in de data een nieuwe regel begint, krijgt het zoeken van een match een soort *reset*.
- Binnen een regel wordt de eerste matchende plek genomen.
 - Deze bepaling geldt alleen maar in editor *search/replace* situaties. Ze heeft dus eigenlijk betrekking op het replacen en niet op het zoeken.
 - In de editor-commandotaal zit altijd wel een mogelijkheid om deze beperking te omzeilen.
- Een regexp is 'greedy' (gulzig). Ze pakt altijd het langst passende stuk.
 - Dit geldt met name voor het gedrag van het sterretje. Als je bijvoorbeeld zoekt naar `e*1` in de tekstregel `veel ervaring dan kun je, als advocaat van de duivel`, zeggen dat niet alleen het stukje `ee1` matcht, maar op diezelfde plek matcht ook `e1` en zelfs de `1` alleen. In dit soort discussies wordt de knoop doorgehakt door de *longest match* bepaling: het wordt de langste die past.
 - Deze bepaling is ondergeschikt aan de vorige. Dus als aan het begin van een regel een korte match zit, maar verderop in de regel is een andere, veel langere match te halen, dan telt toch degene het meest vooraan in de regel.

3.11 Veldjes voor back-reference

De Basic Reguliere Expressies kennen als extra “het markeren van veldjes”. We kunnen dan in die reguliere expressie refereren aan zo’n veldje, en daarmee vragen dat een deel van de expressie wordt herhaald. Deze techniek heet *back-referencing*.

Laten we een voorbeeld nemen. Hierboven keken we naar alle regels waarop `aa` voorkwam. Stel nu dat we alle regels willen hebben waarop een teken twee keer direct na elkaar voorkomt. Dat doen we als volgt:

```
$ grep '\(.\)\1' woordjes
appel
aardappel
pennen
aardnoten
apenoot
pellen.
```

Hier geeft de backslash voor de haakjes juist aan dat die haakjes *niet* letterlijk moeten worden genomen, maar een speciale veldmarkering aangeven³. Dat is

3. In de Extended Regular Expressions hebben ronde haakjes een heel andere betekenis gekregen, dus daar komt *back-referencing* niet voor.

tegenstrijdig met de normale UNIX-betekenis van backslashes, die een speciale betekenis juist ontkrachten. In dit voorbeeld zoeken we een willekeurig teken en door hetzelfde teken gevolgd moet worden: \1 refereert aan wat er tussen de haakjes staat (het nummertje mag 2 of hoger zijn als je meerdere veldjes hebt gemarkeerd). Zoeken naar rijtjes van vier tekens met het eerste en vierde gelijk aan elkaar doen we dus met:

```
$ grep '\(.\)..\1' woordjes
aardappel
pennen
pellen.
maria
```

\ <i>regexpr</i> \)	definieert een veldje, waarvan de inhoud de <i>match</i> van de reguliere expressie <i>regexpr</i> is. De haken zelf hebben geen invloed op wat er precies wordt gematcht. Anders gezegd: een regexp met of zonder dit soort haken matcht precies dezelfde tekst. De haken dienen om het volgende punt te ondersteunen.
\ <i>n</i>	(<i>n</i> is een cijfer 1-9) matcht hetzelfde als waar het veldje dat begint bij het <i>n</i> -de haakje-openen op past.

Als we meerdere veldjes gebruiken, komt de nummering van de veldjes overeen met het haakje openen van de veldjes. Bijvoorbeeld:

```
$ grep '\(.\)(\.)\)\2\1' woordjes
pennen
```

We vinden hier dus alle regels waarop een rijtje van vijf tekens voorkomt, waarbij de eerste twee tekens hetzelfde zijn als de laatste twee en ook het tweede en derde teken gelijk zijn. Merk op dat dit niet hetzelfde is als: ik bedoel dezelfde reguliere expressie als ik tussen de haken had opgeschreven. Het gaat niet om een herhaling van de expressie zelf, maar om een herhaling van wat die expressie had gevonden. Dit voorbeeld laat meteen ook zien dat 'nesting' van dit soort haakjes is toegestaan.

In veel editors zien we dat bij een search-replace operatie deze back-reference notatie zich ook uitstrekt tot het replace-gedeelte van het commando.

4. ERE's: de Extended Regular Expressions

Een aantal jaren na de komst van de 'gewone' Basic Regular Expressions in UNIX zijn, als verbetering en uitbreiding, de Extended Regular Expressions gekomen. Ze zijn niet helemaal compatibel met de Basic RE's. Daarom zijn er geen commando's 'stilzwijgend' uitgebreid naar deze nieuwe mogelijkheden: je moet ófwel een speciaal vlaggetje geven (zoals de `-e` vlag bij `grep`, of de `-r` vlag bij de GNU-versie van `sed`), ófwel je moet gewoon weten dat een bepaald commando alleen ERE's doet (zoals `awk`).

Terugkijkend op wat we in dit artikel al over de BRE's hebben gezegd zijn de belangrijkste veranderingen:

- De back-referencing met `\(` en `\)` haken en de `\n` terugverwijzing is afgeschaft. Ronde haakjes hebben een heel andere betekenis gekregen. De backslash in `\(` die betekende dat een ronde haak een speciale betekenis moest krijgen (dus het tegenovergestelde van *ontkrachting*) is daarmee ook vervallen. Als je nu in ERE's een `\(` of `\)` tegenkomt dan betekent dat dat letterlijk een ronde haak wordt bedoeld, *ontdaan van zijn* (nieuwe) speciale betekenis.
- Ook bij `{` en `}` herhalingsfactor na `[...]` is de backslash als aanduiding van de speciale betekenis voor de accolades vervallen. Accolades zijn nu *altijd* speciaal. Er moet een backslash voorgezet worden nodig om ze te ontkrachten tot 'ik zoek een letterlijke accolade'. De accolades zelf hebben wel hun oude betekenis gehouden van herhalingsfactor na een `[...]` bouwsteen.
- De Kleene star `*` heeft `+` en `?` als broertjes erbij gekregen.
- De belangrijkste uitbreiding is dat meerdere RE's via een Booleaanse *or* met elkaar kunnen worden verbonden.

4.1 + en ? in ERE's

De Kleene-star `*` in zijn betekenis van “nul of meer herhalingen van de voorafgaande bouwsteen” is gebleven. Maar daarnaast is de `+` gekomen met betekenis: “één of meer herhalingen van de voorafgaande bouwsteen”.

We hebben eerder als voorbeeld behandeld:

```
$ grep '^[g-t][g-t]*$' woordjes
```

en dat kan nu eenvoudiger als:

```
$ grep '^[g-t]+$' woordjes
```

Verder is de `?` gekomen in de betekenis van “nul of één herhaling”.

Een consequentie hiervan is dat dus ook de `+` en `?` speciale tekens zijn geworden. Wil je ze ergens letterlijk zoeken dan moet er een `\` voor.

Merk op dat de notatie `regex{1,}` en `regex{0,1}` hetzelfde betekent als deze `+` en `?`. In de Basic Regular Expressions moet voor elke accolade een backslash worden gezet, maar dan hebben we deze specificatie daar dus ook beschikbaar.

4.2 Twee regex'en met Booleaanse or

De belangrijkste uitbreiding in de ERE's is de mogelijkheid om twee reguliere expressies met een Booleaanse *or* aan elkaar te schakelen. Daar komen altijd gewone ronde haken omheen:

```
(eenRegex | andereRegex)
```

zoekt op die plaats naar een stuk tekst dat óf op de *eenRegex* matcht, óf op de *andereRegex*. Een beetje ingewikkelder is:

```
eersteRegex(tweedeRegex|derdeRegex)vierdeRegex
```

zoekt naar tekst waarvan het beginstuk matcht op de *eersteRegex*, daaraan vast moet een stuk zitten dat ófwel matcht op de *tweedeRegex* ófwel op de *derdeRegex*, en daarachter moet een stuk zitten dat matcht op de *vierdeRegex*.

Met deze ronde haken kun je ook 'meertraps' patronen bouwen:

```
([a-z]+[0-9]+ )+
```

tussen de haken staat hier dat je een of meer letters zoekt, gevolgd door een of meer cijfers, gevolgd door een spatie, maar met de + buiten de haken zeg je dat je van die combinatie weer een of meer herhalingen accepteert.

4.3 Filenaam wildcards versus reguliere expressies

Filenaam-wildcard notatie en reguliere expressie notatie worden soms door elkaar gehaald. Maar het zijn twee heel verschillende notatie-systemen!

Bijvoorbeeld, als je op shell-niveau filenamen wilt hebben die met een letter a beginnen dan zeg je:

```
$ ls a*
```

maar als je die via reguliere expressies wilt zoeken dan gaat het zo:

```
ls | grep '^a'
```

Filenaam wildcard expansie wordt door de shell gedaan, nadat je een commandregel hebt afgesloten met *enter*, en voordat het gevraagde commando van start gaat. In UNIX-jargon heet die expansie ook wel *globbing*. De shell doet niet aan reguliere expressies, op een weinig gebruikte uithoek na. En 'gewone' commando's doen niet aan filenaam-expansie op enkele uitzonderingen na, zoals *find* en *tar*.

Filenaam wildcard notatie gaat met de volgende speciale characters:

*	een reeks van nul of meer characters
?	exact één character
[...]	één character uit de verzameling tussen de haken

De [...] notatie is toevallig een component die zowel in de reguliere expressies als in de filenaam wildcards voorkomt. Maar variaties met die blokhaken gaan al meteen verschillend. Als je een character, juist *niet* uit de verzameling wilt hebben dan is dat [^...] bij reguliere expressies en [!...] bij filenaam wildcards. En volgens de Posix-standaard voor filenaam wildcards moet de hyphen tussen blokhaken zich gedragen conform de ingestelde "locale". Dat kan betekenen dat dat gedrag case-insensitive wordt. Dus [a-z] zal in dat geval ook op hoofdletters in de filenaam matchen.