@ at computing

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

---

Versie: 6a

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

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 wat het commando `grep` doet *nadat* het zoeken geslaagd is. 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 zelf: hoe specificeer je wat je wilt (laten) zoeken, en hoe gaat dat zoeken precies in zijn werk. Alle commando’s die regex-notatie ondersteunen, en die we dus als demonstratie-vehikel kunnen gebruiken, hebben dat stuk zoek-notatie en zoek-functionaliteit gemeenschappelijk.

2. Drie generaties regex

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-standaard heeft precieze 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.
- De ontwerper van de programmeertaal Perl heeft ook veel nieuwe ideeën op regular-expressie gebied ingebracht. De *Perl* Regular Expressions vormen dus de jongste van de drie generaties. Ook voor deze generatie geldt dat áls 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 regex zoals `appel` levert alle regels op waarin `appel` voorkomt; dit kan 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 met een `a` *beginnen*. 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 `$` aan het einde van een regex dat het gezochte aan het einde van de regel moet matchen. Daar mag dan zelfs geen spatie meer achter staan. Dus we schrijven als regex `a$` om de regels te vinden die op een `a` *eindigen*:

```
$ 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
            ← deze output is de lege regel uit ons woordjes-bestand
```

Het vasthaken van een reguliere expressie aan het begin en/of het einde van de regel heet *verankering*. In Engels jargon heten deze ^ en $ *anchors*. Het is belangrijk om je te realiseren dat anchors een *positie* aanduiden: de positie *vóór* het eerste teken op de regel, resp. de positie *achter* het laatste teken. De anchors zelf hebben dus een breedte van nul tekens.

> ^ voorop in een regex betekent dat het te zoeken patroon aan het begin van een regel moet staan.
>
> $ achteraan een regex betekent dat het te zoeken patroon aan het eind van een regel moet staan.

Tekstbestanden die zijn gemaakt voor Microsoft Windows geven problemen met het $ anchor. Onder de motorkap zitten die bestanden anders in elkaar dan UNIX/Linux tekstbestanden. Het verschil zit in de manier waarop een regel-einde is gecodeerd. Een UNIX-bestand heeft als regeleinde een ASCII-linefeed code, en een Windows bestand heeft een ASCII-carriage return code, met daarachter een ASCII-linefeed. Als je zoekt naar `tekst$` dan zit bij een Windows-bestand die carriage return in de weg. Er zijn veel tools beschikbaar om tekstbestanden te converteren, maar zonder conversie vind je nooit een match op zo'n $ anchor.

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` òf een `l` gaat als volgt:

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

We zien dat sommige outputregels zowel een `a` als een `l` bevatten, maar iedere gevonden regel bevat minstens één van de twee letters.

Als we laten zoeken naar `[aeo][aou]` dan zoeken we naar twee *opeenvolgende*

letterposities (want elk setje blokhaken telt voor één positie), waarbij we op de eerste positie een a of e of o accepteren, en op de tweede positie een a of o of u. Merk dus op dat een regex-specificatie positie voor positie aan elkaar schakelt wat we precies willen zoeken.

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

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

3.3 De complementaire verzameling

We gaan het nog algemener maken: we willen bijvoorbeeld zoeken naar alle regels die *niet* op een a eindigen. Dan zouden we tussen blokhaken een uitputtende lijst van alle mogelijke tekens moeten maken die geen a zijn. Dat is veel werk. Het is ook lastig om zeker te weten of we alle mogelijke tekens hebben opgeschreven. Dus er is een notatie nodig met de betekenis: *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). Let op: het dakje kan in reguliere expressies dus twee verschillende dingen betekenen: aan het begin van een regex is het het regelbegin-anker symbool, en vlak na een blokhaak-openen geeft het aan dat de rest tussen die blokhaken betekent: “niet dezen”.

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
```

Merk op: de lege regel vinden we niet, want op die regel staat geen enkele letter, 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: dat 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 ergens 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 expliciete verzameling opgesomd tussen blokhaken.

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

```
$ grep 'a..a' woordjes
aardappel           ← hier matcht arda
maria               ← hier matcht aria
```

3.5 Speciale tekens degraderen

En hoe moeten we de regels zoeken die eindigen op een punt? Niet met `.$` want dat toont alle niet-lege regels (regels met één willekeurig teken vlak voor het einde).

Hiervoor is een systematische oplossing: een teken dat een bijzondere betekenis heeft binnen de reguliere expressie-notatie 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 regex-taal is geworden. Dus zoeken van één letterlijke backslash moet zo: \\

De regels die eindigen op een (letterlijke) 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, naast backslash, ook quotes als alternatieve notatie voor ditzelfde 'degradatie'-doel. De regex-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: elke kleine letter uit het alfabet.

Hier moet een waarschuwing bij: de onderlinge volgorde van tekens ligt vast in de z.g. *ASCII-characterset* 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 terwijl in de ASCII-standaard die volgorde andersom is. Gebruik in plaats daarvan `[a-zA-Z]` Dus het liggend streepje koppelt de letter die je vlak ervoor schrijft aan de letter die je vlak erachter schrijft. Wil je een letterlijk liggend streepje zoeken dan kun je 'm als laatste vlak voor de sluit-blokhaak zetten.

Letters met accenten komen in de ASCII-characterset helemaal niet voor. Daarover volgt later in dit verhaal 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 `nooot` enzovoorts.

Zo'n herhaling kunnen we aangeven door achter een teken een *quantifier* te plaatsen, waarmee we iets zeggen over hoe vaak dat teken achter elkaar mag voorkomen. De meest gebruikte quantifier is het *sterretje*[2]. Als we achter een teken een sterretje plaatsen, accepteren we dat teken *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 daarachter nog meerdere `o`'s.

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

---

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

Bijvoorbeeld: toen we eerder de `$` behandelden als einde-regel anker, schreven we daarbij als waarschuwing: “de voorafgaande expressie moet aan het einde van een regel matchen. Daar mag dan zelfs geen spatie meer achter staan”. Maar stel dat we dit willen afzwakken en eventueel spaties achteraan de regel toch mee willen accepteren. Dan laten we onze reguliere expressie eindigen op `*$` (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; 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. Het sterretje hoort dus uitsluitend bij de laatste van de drie `[aeio]` stukken, want dat is het optionele stuk. De eerste twee `[aeio]` stukken gaan over wat *verplicht* aanwezig moet zijn.

Let ook op de juiste interpretatie. Het betekent niet: 'twee of meer `a`' of 'twee of meer `e`' enz, maar 'twee of meer elementen die *elk voor zich* matchen op `[aeio]`'. Dus elke mix van `a`’s, `e`’s, `i`’s en `o`’s is geldig, als die maar minstens twee lang is.

> `*` 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 regels 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
                                ← de lege regel wordt ook gevonden
pint
```

Als we de lege regel niet erbij 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). We proberen eerst dit:

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

We vinden meer dan de bedoeling is, want we zijn de verankering vergeten. We vinden bijvoorbeeld `mat` omdat onze regex niet eist dat de regel met een `a` moet *beginnen*. Beter is:

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

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

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

Zoekt dit alle regels die *exact* twee maal een `e` bevatten? Nee, herhaal de opdracht maar eens met een letter `a`:

```
$ grep 'a.*a' woordjes
aardappel               ← hier zitten drie a's in
aardnoten
maria
```

Als we *exact* twee keer een a eisen, moeten we ook uitsluiten dat tussen de rest van de tekens in die regel nog 'n extra letter a zit. Om alle tekens van de regel mee te nemen in onze specificatie, gebruiken we verankering:

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

In woorden uitgedrukt kun je bovenstaande lezen als: vanaf het begin van de regel eerst "nul of meer niet-a's" (dat is het stukje "`[^a]*`"). Daarachter volgt de eerste a, dan volgt weer "nul of meer niet-a's", dan weer (wel een) a, en daarna tot aan het einde van de regel nog "nul of meer niet-a's". Of, nog anders gezegd: vanwege de twee ankers overdekt deze specificatie de gehele regel, en je ziet in de regex de beide (wel-)a's staan. Daarvoor, daartussen en daarachter mogen alleen niet-a's staan. Op een schematische manier getekend:

```
^     a     a     $
 [^a]* [^a]* [^a]*
```

Constructies in deze trant komen veel voor, dus het is de moeite waard om dit voorbeeld goed te bestuderen.

3.8 Nog meer herhalingen

De Kleene-ster die volgt op een regex zegt: van die regex accepteer ik *nul of meer* herhalingen. Er bestaat een notatie om andere aantallen te kiezen in plaats van *nul of meer*, hoewel die in de praktijk weinig wordt gebruikt. Dit is de notatie:

regex`\{`*m*`,`*n*`\}`

Achter de regex 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:

regex`\{`*m*`,\}` → *m* stuks en meer (minstens *m*)
regex`\{`*m*`\}` → precies *m* stuks

Dus *regex*`*` is hetzelfde als *regex*`\{0,\}`

Het gebruik van de backslash is hier strijdig met de normale betekenis: hier maakt een backslash de accolade juist speciaal in plaats van dat hij 'm degradeert.

3.9 Groeperen met haakjes

We hebben gezien dat herhalingen steeds van toepassing zijn op het direct voorgaande teken. Om ook in staat te zijn een groep van meerdere tekens aan te geven voor herhaling, kun je tekens groeperen door ze tussen haakjes te zetten. De haakjes moeten bij BRE's wel vooraf gegaan worden door een backslash. Dus:

```
\(dat\)*
```

zoekt naar nul of meer voorkomens van de string “dat”. Verderop zullen we zien dat groeperen met haakjes nog een bijeffect heeft.

Ook hier is, net als in de vorige paragraaf, het gebruik van de backslashes weer strijdig met de normale betekenis: ze maken de haakjes *speciaal*, dus promoveren in plaats van degraderen. De reden is dat deze speciale notaties niet vanaf het allereerste begin in de reguliere expressies hebben gezeten, dus haakjes en accolades waren oorspronkelijk *gewone* characters. Om compatibel te blijven moest daarom een *promoveren* notatie worden bedacht. In de Extended Reguliere Expressies (ERE’s), die we later in dit verhaal bespreken, is dit rechtgetrokken: haakjes en accolades zijn daar speciaal, en met een backslash kun je ze *degraderen*.

3.10 Character classes

Eerder zagen we 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 (officiële term: met diacritische tekens) ontbreken, evenals allerlei nationale uitbreidingen zoals de ß, de ð of de ¿

Daarom zijn er “verzamelnamen” gemaakt, waarmee je in één 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 leestekens
`[: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 regex-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 dat we nog steeds binnen het Latijnse alfabet blijven. Uitbreiden naar andere alfabetten, of zelfs naar alfabet-loze talen (Chinees, Japans, Koreaans) is een specialisme apart.

3.11 Het huishoudelijk reglement

Als de expressies ingewikkelder worden krijg je misschien verschillen in interpretatie van hun betekenis. Er is een soort "huishoudelijk reglement" om moeilijke knopen door te hakken en zo de eenheid te bewaren:

- Een regex matcht alleen maar *binnen* één 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, dan krijgt het zoeken van een match een soort *reset*.
- Binnen een regel wordt de eerste matchende plek genomen:
  - Deze *first match* bepaling is alleen maar van toepassing in editor *search*/*replace* situaties. Ze heeft dus eigenlijk betrekking op het replacen en niet op het zoeken.
  - In de editor-commandotaal zit meestal wel een mogelijkheid om deze *first match* beperking te omzeilen.
- Quantifiers zijn *greedy* (gulzig). Ze pakken altijd het langst passende stuk.
  - Als je bijvoorbeeld zoekt naar `e*l` in de tekstregel `veel ervaring` dan kun je, als advocaat van de duivel, zeggen dat niet alleen het stukje `eel` matcht, maar op diezelfde plek matcht ook `el` en zelfs de `l` alleen. In dit soort discussies wordt de knoop doorgehakt door deze *longest match* bepaling: het wordt de langste die past.
  - Deze bepaling is ondergeschikt aan de vorige (maar die vorige is dus niet altijd van toepassing). Als aan het begin van een regel een korte match zit, maar verderop in die regel is een andere, veel langere match te halen, dan telt toch degene het meest vooraan in de regel.
- Matching is hoofdlettergevoelig *(case sensitive)*. Doorgaans is er wel een mogelijkheid om het matchen case *in*sensitive te laten uitvoeren. Maar dat zit dan in de argumenten van het gebruikte commando of de gebruikte programmeertaal; niet in het regex-gedeelte.

3.12 Back-referencing

Eerder is genoemd dat het groeperen van tekens tussen haakjes een bijeffect heeft. Wat tussen de haakjes staat wordt niet alleen gegroepeerd, maar ook *onthouden*. We kunnen verderop in de regex naar deze onthouden tekens refereren. 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           ← match op het gedeelte  pp
aardappel       ← match op het gedeelte  aa  en op  pp
pennen          ← match op het gedeelte  nn
aardnoten       ← match op het gedeelte  aa
apenoot         ← match op het gedeelte  oo
pellen.         ← match op het gedeelte  ll
```

In dit voorbeeld zoeken we een willekeurig teken dat door hetzelfde teken gevolgd moet worden: `\1` refereert aan wat er tussen de haakjes staat (je kunt naar maximaal 9 groepen terugverwijzen op die manier).
Dus zoeken naar setjes van vier tekens met het eerste en vierde gelijk aan elkaar:

```
$ grep '\(.\)..\1' woordjes
aardappel       ← match op het gedeelte  arda
pennen          ← match op het gedeelte  enne  en op  nnen
pellen.         ← match op het gedeelte  elle
maria           ← match op het gedeelte  aria
```

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

Als we meerdere groepen gebruiken, komt de nummering van de groepen overeen met het haakje openen van de groepen. Bijvoorbeeld bij deze geneste haakjes:

```
$ grep '\(.\(.\)\)\2\1' woordjes
pennen              ← match op het gedeelte  ennen
```

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 `\n` ook nog gebruikt mag worden in het replace-gedeelte van het commando.

4. ERE's: de Extended Regular Expressions

Enkele 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 is geen enkel commando '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:

- Ronde haakjes worden niet langer door een backslash voorafgegaan. Bovendien ondersteunen ze geen backreferences. Er zijn wel implementaties die dat doen, maar dat is dus niet officieel.
- Ook bij `{` en `}` herhalingsfactor (quantifier) is de backslash als aanduider van de speciale betekenis voor de accolades vervallen. Accolades zijn nu *altijd* speciaal. Er moet een backslash voorgezet worden om ze te degraderen tot 'ik zoek een letterlijke accolade'. De accolades zelf hebben wel hun oude betekenis gehouden van herhalingsfactor.
- De Kleene ster `*` heeft `+` en `?` als broertjes erbij gekregen.
- De belangrijkste uitbreiding is dat meerdere RE's via een Booleaanse *or* met elkaar kunnen worden verbonden.

Een bekende implementatie van ERE's is de *regex*-library van Henry Spencer. Die is niet helemaal compatible met de Posix-standaard voor ERE's, maar wel grotendeels. Deze library wordt o.a. gebruikt in MySQL. MariaDB, de "opvolger" van MySQL, gebruikt Perl-compatible regex'en.

4.1 Groeperen in ERE's

Met de 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. Merk op dat we bij de haken geen backslashes meer gebruiken!

4.2 + en ? quantifiers in ERE's

De Kleene-ster `*` 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 -E '^[g-t]+$' woordjes
```
← flag `-E` schakelt `grep` naar ERE's

Verder is de `?` erbij 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 is deze specificatie met accolades ook beschikbaar, maar daar moet vóór elke accolade een backslash erbij worden gezet.

4.3 Twee regex'en met Booleaanse OR (alternation)

De belangrijkste uitbreiding in de ERE's is de mogelijkheid om twee reguliere expressies met een Booleaanse *or* aan elkaar te schakelen.

```
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*.

Merk op dat je in dit voorbeeld de twee alternatieven moet groeperen (met ronde haken) om ze te kunnen onderscheiden van de rest van de regex.

5. Filenaam wildcards versus reguliere expressies

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

Bijv. op shell-niveau filenamen zoeken die met een letter `a` beginnen gaat zo:

```
$ 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, en 'gewone' commando's doen niet aan filenaam-expansie (globbing) op enkele uitzonderingen na, zoals `find`, `tar` en `locate`[3].

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, met identieke betekenis. 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 het globbing gedrag case-insensitive wordt. Dus `[a-z]` zal in dat geval ook op hoofdletters in de filenaam matchen.

6. Perl Reguliere Expressies

■ Op dit punt in het verhaal maken we een grote stap voorwaarts: van Basic en Extended Regular Expressions gaan we nu naar de derde generatie: Perl Regular Expressions. Daarmee wordt het meteen een flink stuk ingewikkelder. Het is misschien verstandig om op dit punt op te houden met lezen en pas terug te komen als je ervaring met de BRE's en ERE's hebt opgedaan. De Perl Regular Expressions zijn vooral nuttig voor programmeurs. Maar ze zitten ook in zwaarkaliber gereedschap, zoals het URL-rewrite module van de Apache webserver. In commando's aan de prompt, of in shell scripts, kom je ze niet zo vaak tegen. □

De scripttaal Perl heeft veel van zijn populariteit te danken aan het gebruik van reguliere expressies. Perl is in 1987 ontstaan, en was in 1995 bij versie 5 beland. Deze versie bracht vele uitbreidingen ten opzichte van ERE's. Dit is lange tijd de benchmark voor anderen geweest om zich aan te meten op dit vlak.

De meeste script- en programmeertalen ondersteunen tegenwoordig in meer of mindere mate reguliere expressies. Ze implementeren daarbij doorgaans zoveel mogelijk de Perl syntax hiervoor. Soms is de ondersteuning ingebouwd in de taal, maar meestal zit de ondersteuning in een bibliotheek.

In dit hoofdstuk komen de meeste uitbreidingen aan bod die Perl ons heeft gebracht.

Om Perl reguliere expressies op vergelijkbare wijze te demonstreren zoals we eerder

---

3. Als 'gewone' commando's aan globbing doen dan is dat altijd omdat ze ergens anders moeten zoeken dan in de huidige directory. Bijv. in een hele fileboom, in een tar-file of in een database van namen.
4. De GNU-versie van `grep` heeft de `-P` vlag om Perl reguliere expressies te behandelen. Maar ter ere van de bedenker stappen we over op voorbeelden met het commando `perl` in plaats van `grep`.

met `grep`[4] deden, hanteren we de volgende vorm op de commandoregel aan de prompt:

```
$  perl -ne 'print if (/regex/)' tekstfile
```

Dit laat `perl` de regels uit de *`tekstfile`* tonen waarin de gegeven regex matcht. Merk op dat in dit geval de regex tussen slashes wordt geplaatst. De slash is in Perl het scheidingsteken voor letterlijke reguliere expressies, net zoals letterlijke strings in de meeste talen tussen aanhalingstekens worden gezet.

6.1 Quantifiers

Een eenvoudige uitbreiding op de bestaande {*n*,*m*}-varianten is:

```
{,m}
```

Dit betekent: *nul tot en met m* keer het voorgaande. Sommige regex-implementaties ondersteunen deze vorm niet. Je kunt dan natuurlijk ook gewoon {0,*m*} schrijven.

6.1.1 Lazy quantifiers (match zo weinig mogelijk tekens)

Veel belangrijker is de mogelijkheid om quantifiers *lazy* te maken. Tot nu toe zagen we quantifiers die altijd *zo veel mogelijk* tekens proberen te matchen, en die we daarom *greedy* noemen. Dus, gegeven de tekst:

```
<p>Een html-paragraaf.</p><p>De tweede paragraaf.</p>
```

met de volgende regex: `<p>(.*)</p>`

levert een match op de groep (d.w.z. het gedeelte tussen haakjes):

```
Een html-paragraaf.</p><p>De tweede paragraaf.
```

en dus niet slechts op:

```
Een html-paragraaf.
```

Misschien was die lange match niet de bedoeling. Als je inderdaad slechts de tekst tussen het eerste ... -paar in een groep wil 'vangen', kun je de * in de regex *lazy* maken. Dat betekent: match zo *weinig* mogelijk tekens.

Je maakt een quantifier lazy door er een vraagteken ? achter te plaatsen.
Dus als we de regex op die manier aanpassen zal de groep in:

```
<p>(.*?)</p>
```

in ons voorbeeld wel alleen

```
Een html-paragraaf.
```

bevatten, omdat het matchen stopt bij de eerste

6.1.2 Possessive quantifiers

Er bestaat nog een derde vorm van quantifiers, waarvan het gedrag lijkt op lazy quantifiers, maar subtiel anders is. Je maakt een quantifier *possessive* door er een + (plus) achter te plaatsen. Dat ziet er dan bijv. zo uit: `*+` of `{m,n}+` of `?+` en zelfs `++`

Een uitleg van de exacte werking en toepassing van possessive quantifiers valt buiten het bereik van dit artikel. Er is kennis over de interne werking van regex-implementaties nodig om dit te kunnen verduidelijken.

6.2 Grouping

Grouping wordt op dezelfde wijze genoteerd als bij ERE's, dus tussen ronde haken, zoals in bovenstaand voorbeelden bij de lazy quantifiers al te zien was.

6.2.1 Non-capturing groups

Nieuw bij de Perl Regular Expressions is dat je nu expliciet kunt aangeven dat de gematchte tekens in de groep *niet* onthouden hoeven te worden voor later gebruik als backreference. Dat onthouden kost immers geheugen en extra werk, dus als je alleen haakjes zet om een paar tekens om ze te groeperen kun je het systeem dat werk besparen, en kan de zoekopdracht sneller worden uitgevoerd.

Een groep waarvan de gematchte tekens *niet* hoeven te worden onthouden heet een *non-capturing group*. Je noteert een non-capturing group door direct na het haakje-openen een vraagteken en een dubbele punt te plaatsen:

```
abc(?:de)*fg
```

In deze regex wordt `de` gegroepeerd, zodat de `*` op de hele groep werkt. De substring `de` hoeft verder niet onthouden te worden, vandaar dat we de groep non-capturing hebben gemaakt. De regex beschrijft dus: `abc` gevolgd door nul of meer keer `de`, gevolgd door `fg`.

Let op: non-capturing groups *tellen niet mee* in de telling van capturing groups! Neem dus in je telling alleen de haakjes-openen mee die *niet* gevolgd worden door een vraagteken. In de regex:

```
ab(cd)*e(?:gh)*i(jk)lm\1\2
```

refereert `\1` naar `cd`, en `\2` naar `jk`. De groep met `gh` telt dus niet mee!

6.2.2 Atomic grouping

Atomic grouping wordt niet veel gebruikt, en voor het begrijpelijk uitleggen van wat het precies is, is kennis nodig over de interne werking van regex-implementaties. We laten ze in dit artikel verder buiten beschouwing. Ze zijn te herkennen aan een vraagteken en een > teken, direct na het haakje-openen. Dat ziet er zo uit:

```
(?>Regex)
```

Net als non-capturing groups tellen atomic groups *niet* mee bij de telling van

groepen voor backreferencing.

6.3 Backreferences

De notatie voor backreferences die we al kennen uit de BRE's is gehandhaafd. Je kunt dus met `\1` tot en met `\9` refereren aan de eerste tot en met de negende groep. Ter herinnering: groepering gaat in BRE's met `\(...\)` en in ERE's met `(...)`

Om meer dan 9 groepen te ondersteunen in een regex is een alternatieve notatie ingevoerd: `\g{1}` tot en met `\g{99}`. Hiermee verwijs je dus naar de eerste tot en met de negenennegentigste groep.

6.4 Character classes

6.4.1 Handige afkortingen

Naast de bekende zelf samen te stellen verzamelingen tekens die op een bepaalde plek in een tekst mogen matchen, genoteerd met `[...]`, zijn er enkele afkortingen toegevoegd voor veelgebruikte character classes.

Op elke positie waar een *white space* character mag staan kun je `\s` noteren. Onder *white space* wordt begrepen: spatie, tab, linefeed, vertical tab, form feed, enz. Als de regex-implementatie Unicode ondersteunt vallen ook alle daar gedefinieerde vormen van *white space* eronder.

Net zoals je in de `[...]`-notatie ook kunt aangeven dat je de genoemde tekens op die plek in de regex *niet* wilt matchen, door als eerste teken in de verzameling een `^` te noteren als volgt `[^...]`, kun je ook aangeven dat je op een bepaalde plek in de regex juist *geen* white space wilt hebben. Je kunt dat noteren door `\S` te gebruiken, dus met een hoofdletter.

Op vergelijkbare wijze stelt `\d` een cijfer voor, en is dus hetzelfde als `[0-9]` en staat `\D` voor elk teken *behalve* een cijfer, dus `[^0-9]`

Tenslotte is er `\w` voor een *word character*, wat gedefinieerd is als een letter, cijfer of underscore, dus: `[a-zA-Z0-9_]` Let op: als de regex-implementatie Unicode ondersteunt dan wordt het begrip *letter* een stuk breder opgevat. Daar vallen in dat geval ook alle variaties met accenten onder, en zelfs de verschillende lettertekens uit andere alfabetten dan het Latijnse, zoals Grieks, Cyrillisch, Arabisch, Hebreeuws enz.

Net als bij de andere afkortingen kun je de betekenis omdraaien door een hoofdletter te gebruiken: `\W` staat voor elk teken dat geen letter, cijfer of underscore is.

Samengevat:

notatie	*betekenis*
`\s`	whitespace
`\S`	alles behalve whitespace
`\w`	woordcharacter (letter, cijfer, underscore)
`\W`	alles behalve een woordcharacter
`\d`	cijfer
`\D`	alles behalve een cijfer

Tabel 1. Korte notatie voor veelgebruikte character classes

6.4.2 Unicode code points

In regex-implementaties die Unicode ondersteunen kun je willekeurige tekens matchen via hun Unicode code point. Het copyright-teken (©), bijvoorbeeld, heeft als Unicode code point U+00A9. Als je deze wilt opnemen in een regex, noteer je (in de meeste talen):

```
\u00a9
```

Die notatie werkt niet in Perl en in PCRE,[5] waar je de volgende notatie gebruikt:

```
\x{a9}
```

Merk op dat je daarbij voorloopnullen mag weglaten. Deze notatie wordt niet verward met de quantifier `{n}`, omdat `\x` geen geldige losstaande notatie is. Wanneer je noteert:

```
\x{2500}{50}
```

wordt dat dus opgevat als 50 keer het code point U+2500.

Het matchen van Unicode code points kent wel valkuilen. Sommige tekens hebben in Unicode meerdere representaties: geaccentueerde letters, bijvoorbeeld, hebben een concreet code point voor het gehele teken, maar kunnen ook beschreven worden met een code point voor de kale letter zonder accent, gevolgd door een speciaal code point voor het accent dat daarmee gecombineerd moet worden.

Dus een é kan gerepresenteerd worden door U+00E9 (voor het hele teken), maar ook door de combinatie van e (U+0065), gevolgd door het accent (U+0301).

5. PCRE is de Perl Compatible Regular Expression library. Dit is een open source bibliotheek, geschreven door Philip Hazel van Cambridge University, die door C/C++-programmeurs gebruikt kan worden om reguliere expressies in hun eigen software te kunnen ondersteunen. De taal PHP en de MariaDB database (“opvolger” van MySQL) zijn ook voorbeelden van waar de PCRE wordt gebruikt.

6.4.3 Unicode character classes

Met Unicode-ondersteuning heb je de beschikking over een extra arsenaal aan character classes. Er is een aantal *hoofdcategorieën*, zoals letters, symbolen, diacritische tekens, interpunctie, en dergelijke, die elk weer zijn onderverdeeld in *subcategorieën* voor nog fijner onderscheid.

Elke (sub)categorie kun je schrijven in een uitgebreide, leesbare vorm en in een afgekorte vorm. Een *letter* kun je aangeven als `\p{Letter}` of als `\p{L}`. Een subcategorie van `\p{Letter}` is bijvoorbeeld *hoofdletter*: `\p{Uppercase_Letter}`, afgekort als: `\p{Lu}`

Een overzicht van alle Unicode (sub)categorieën vind je in Appendix A van dit verhaal.

6.5 Anchors

Bij de Perl Regular Expressions kun je met `\b` een woordgrens aangeven. Een woordgrens is de overgang tussen een *word character* en een niet-*word character*. Bedenk wel dat net als bij de eerder behandelde anchors (^ en $) deze geen breedte hebben! Een anchor geeft altijd een plek *tussen* twee tekens aan.

Als je juist wilt aangeven dat je op een bepaalde plek in de regex *geen* woordgrens wilt matchen, gebruik je weer de hoofdletter-variant, dus: `\B`

De regex `\bvak\b` matcht wel in de zin:

```
Dit vak is soms best moeilijk.
```

maar niet in:

```
We gaan op vakantie.
```

Maar als je woorden zou willen vinden die beginnen met `vak`, maar niet het losse woord `vak`, kun je dus de regex `\bvak\B` gebruiken. Die zou in de tweede zin wèl de eerste drie letters van het woord `vakantie` matchen.

6.6 Assertions

Assertions bestaan niet in BRE's en ERE's. Dus dit is een nieuw begrip. Een assertion is te beschouwen als een programmeerbare anchor. Waar je bijvoorbeeld met de anchor `\b` aangeeft dat op die plek in de regex een woordgrens moet zijn, kun je met een assertion *een willekeurige regex* als test aangeven, waaraan de te matchen tekst moet voldoen op die plek.

Net als een anchor heeft een assertion *geen breedte*, maar test hij of een positie *tussen* twee tekens aan een gegeven regex voldoet, of juist niet. Daarbij kan een assertion naar links (*lookbehind*) of naar rechts (*lookahead*) 'kijken'.

De combinaties van enerzijds wel of niet aan een test voldoen en anderzijds naar links of rechts kijken leveren vier soorten assertions op:

notatie	*betekenis*	*naam*
`(?=`*regex*`)`	rechts hiervan moet *regex* matchen	positive lookahead
`(?!`*regex*`)`	rechts hiervan mag *regex* niet matchen	negative lookahead
`(?<=`*regex*`)`	links hiervan moet *regex* matchen	positive lookbehind
`(?<!`*regex*`)`	links hiervan mag *regex* niet matchen	negative lookbehind

Tabel 2. Lookaround assertions

De regex `[a-z](?=\d)` matcht een kleine letter, gevolgd door een cijfer. Dat zou je ook kunnen vinden met `[a-z]\d`, maar dan maakt het cijfer ook echt onderdeel uit van de match. Dat is van belang wanneer je de regex gebruikt bij zoeken gevolgd door vervangen:

`[a-z]\d` matcht in `abc123` de substring `c1`
`[a-z](?=\d)` matcht in `abc123` de substring `c`

Het is belangrijk te blijven beseffen dat een assertion nooit tekens matcht, maar slechts test of een bepaalde conditie geldt op een gegeven positie in de regex.

Verder geldt ook hier dat assertions *niet* meetellen bij de telling van capturing groups.

Voorbeeld: we tonen alle regels uit ons `woordjes` bestand, waarin een `p` voorkomt die *niet* door een `a` vooraf gegaan wordt.

```
$ perl -ne 'print if (/(?<!a)p/)' woordjes
```

Dat levert als resultaat de volgende matchende regels:

```
appel
aardappel
pennen
piet.
lopen
pellen.
pinda
pint
```

We zien ook regels terug waarin wel een `p` met een `a` ervoor staat, maar in elk geval heeft zo'n regel óók een `p` zonder `a` ervoor.

Tenslotte nog een voorbeeld van een positive lookahead, die in de context van Perl negatief gebruikt wordt. We zijn op zoek naar de regels waarin *niet* twee maal dezelfde letter achter elkaar voorkomt.

```
$ perl -ne 'print unless (/(.)(?=\1)/)' woordjes
```

De regex matcht elke regel waarin een letter staat, direct gevolgd door dezelfde letter. Het woordje `unless` is Perl-taal om elke regel te laten printen, *tenzij* die matcht met de gegeven regex. Dit geeft de volgende outputregels als resultaat:

```
piet.
lopen

pinda
mat
maria
pint
```

6.7 Backtracking

Bij gebruik van greedy quantifiers speelt het mechanisme van *backtracking* een rol. Dit houdt in dat de regex 'machine' tijdens het scannen van de tekst steeds in de gaten houdt op welke plek er een match is, en die plek onthoudt voor het geval later blijkt dat dit de langste match was. Als hij verderop in de tekst een langere match vindt, vergeet hij de eerste onthouden positie en onthoudt nu deze nieuwe. Bij het einde van de tekst aangekomen komt hij vaak tot de ontdekking dat er geen langere match is dan de laatst ontdekte, en *keert hij op zijn schreden terug*. Dat heet backtracking.

Een mooi voorbeeld dat laat zien hoe je je met backtracking in de voet kunt schieten is de volgende combinatie van *twee opeenvolgende* greedy quantifiers. Met deze regex:

```
.*(\d+)
```

zoeken we een match in de tekst:

```
ov9292
```

De strikvraag is: wat zit er in de groep?

Antwoord: alleen de laatste 2!
Verklaring: de `*` die op de `.` werkt probeert zoveel mogelijk tekens te matchen, liefst tot en met het laatste teken in de tekst aan toe! Bij het laatste teken aangekomen (de laatste 2), wordt geconstateerd dat er geen match is, tenzij er een stap terug wordt gedaan. De 2 blijft dan over voor het deel in de groep, dat één of meer cijfers wil matchen. Je ziet hier dat de eerste greedy quantifier het 'wint' van de tweede.

Als we een vergelijkbare regex willen maken waarmee wel zoveel mogelijk van de cijfers terecht komen in de groep, dan lukt dat door de `*` lazy te maken:

```
$ echo 'ov9292' | perl -ne 'print $1,"\n" if (/.*?(\d+)/)'
```

Dit zou een complete commandoregel zijn om deze match te testen. De `$1` bij het `print`-statement is een Perl-specifieke back-reference notatie, die aangeeft dat we alleen het stukje willen printen dat door het groep-gedeelte van de regex gematcht wordt. Elke programmeertaal heeft zijn eigen manier om buiten een reguliere

expressie te refereren naar de inhoud van een genummerde groep.

6.8 Er is meer mogelijk

Er zijn met Perl regular expressions meer constructies mogelijk dan we hier behandeld hebben. Wat nog rest zijn specialistische expressies die buiten het bereik van dit artikel vallen. Wat hier wel behandeld is wordt door de meeste implementaties ondersteund, en is daardoor breed toepasbaar.

A. Regular expression cheat sheet

Huishoudelijk reglement:

1. Regex matcht alleen binnen één regel
2. De *eerste* match telt
3. Standaard quantifiers zijn *greedy*
4. Matching is *case sensitive*

Metacharacters:

De volgende tekens moet je escapen met een backslash '\' als je ze letterlijk wil matchen:

```
^ $ ( ) [ { \ | . * + ?
```

Anchors:

notatie	*betekenis*
`^`	begin van de regel
`$`	eind van de regel
`\b`	woordgrens
`\B`	niet een woordgrens

Alternation:

notatie	*betekenis*
`eenRegex|andereRegex`	matcht `eenRegex` óf `andereRegex`

Quantifiers:

Greedy: match *zoveel* mogelijk tekens.
Lazy: match zo *weinig* mogelijk tekens.
Possessive: niet behandeld in dit artikel.

betekenis	*greedy*	*lazy*	*possessive*
nul of meer	`*`	`*?`	`*+`
een of meer	`+`	`+?`	`++`
nul of een	`?`	`??`	`?+`
precies *n*	`{n}`	`{n}?`	`{n}+`
minstens *n*	`{n,}`	`{n,}?`	`{n,}+`
hoogstens *m*	`{,m}`	`{,m}?`	`{,m}+`
n tot en met *m*	`{n,m}`	`{n,m}?`	`{n,m}+`

Grouping:

notatie	*betekenis*
`(regex)`	*regex* is een groep; wat matcht wordt onthouden *(capturing group)*
`(?:regex)`	*regex* is een groep; wat matcht wordt *niet* onthouden *(non-capturing group)*, en niet genummerd
`(?>regex)`	atomic grouping (niet behandeld in dit artikel)

Character classes:

notatie	*betekenis*
`\s`	whitespace
`\S`	niet whitespace
`\w`	woordcharacter (letter, cijfer, underscore)
`\W`	niet een woordcharacter
`\d`	cijfer
`\D`	niet een cijfer
`[...]`	verzameling *gewenste* tekens en/of combinatie van character classes
`[^...]`	verzameling *ongewenste* tekens en/of combinatie van character classes
`[a-z]`	*range* van `a` tot en met `z`

Unicode character class categories:

afgekort	*lange vorm*	*betekenis*
`\p{L}`	`\p{Letter}`	letter, van wat voor alfabet dan ook
`\p{M}`	`\p{Mark}`	een teken dat bedoeld is te combineren met een ander teken
`\p{Z}`	`\p{Separator}`	elke soort witruimte of onzichtbaar scheidingsteken
`\p{S}`	`\p{Symbol}`	wiskundige symbolen, lijntjes, valutatekens, enz.
`\p{N}`	`\p{Number}`	numeriek teken
`\p{P}`	`\p{Punctuation}`	interpunctie
`\p{C}`	`\p{Other}`	onzichtbare stuurtekens en ongebruikte code points

Unicode subcategorieën:

afgekort	*lange vorm*	*betekenis*
\p{Ll}	\p{Lowercase_Letter}	kleine letter die een hoofdletter-variant heeft
\p{Lu}	\p{Uppercase_Letter}	hoofdletter die een kleine letter-variant heeft
\p{Lt}	\p{Titlecase_Letter}	hoofdletter aan het begin van een woord
\p{L&}	\p{Cased_Letter}	combinatie van \p{Ll}, \p{Lu} en \p{Lt}
\p{Lm}	\p{Modifier_Letter}	speciaal teken dat gebruikt wordt als letter
\p{Lo}	\p{Other_Letter}	letter die geen hoofd- en klein onderscheid heeft
\p{Mn}	\p{Non_Spacing_Mark}	een teken dat bedoeld is om gecombineerd te worden met een ander teken, zonder extra ruimte in beslag te nemen (bijv. umlaut, accenten, enz.)
\p{Mc}	\p{Spacing_Combining_Mark}	een teken dat bedoeld is om gecombineerd te worden met een ander teken, dat daarbij wel extra ruimte in beslag neemt (klinker-tekens in oosterse talen)
\p{Me}	\p{Enclosing_Mark}	een teken dat het teken insluit waarmee het gecombineerd wordt (bijv. cirkel, vierkant, toetsenbord-toets)
\p{Zs}	\p{Space_Separator}	een onzichtbaar witruimte-teken dat wel ruimte in beslag neemt
\p{Zl}	\p{Line_Separator}	regelscheider U+2028
\p{Zp}	\p{Paragraph_Separator}	alineascheider U+2029
\p{Sm}	\p{Math_Symbol}	elk wiskundig symbool
\p{Sc}	\p{Currency_Symbol}	elk valutateken
\p{Sk}	\p{Modifier_Symbol}	een te combineren teken, maar dan zelfstandig
\p{So}	\p{Other_Symbol}	verschillende symbolen die niet in een bovenstaande subcategorie vallen
\p{Nd}	\p{Decimal_Digit_Number}	cijfer 0 tot en met 9
\p{Nl}	\p{Letter_Number}	een cijfer dat er uitziet als een letter, zoals een romeins cijfer
\p{No}	\p{Other_Number}	een sub- of superscript cijfer, of een ander cijfer dat niet 0-9 is
\p{Pd}	\p{Dash_Punctuation}	elke soort liggend streepje
\p{Ps}	\p{Open_Punctuation}	elke soort haakje openen
\p{Pe}	\p{Close_Punctuation}	elke soort haakje sluiten
\p{Pi}	\p{Initial_Punctuation}	elke soort aanhalingsteken openen
\p{Pf}	\p{Final_Punctuation}	elke soort aanhalingsteken sluiten
\p{Pc}	\p{Connector_Punctuation}	een verbindingsteken zoals underscore
\p{Po}	\p{Other_Punctuation}	elke soort interpunctie die niet onder bovenstaande subcategorieën valt
\p{Cc}	\p{Control}	een ASCII 0x00-0x1F of Latin-1 0x80-0x9F stuurteken

`\p{Cf}`	`\p{Format}`	onzichtbaar vormgevingsteken
`\p{Co}`	`\p{Private_Use}`	elk code point gereserveerd voor eigen gebruik
`\p{Cs}`	`\p{Surrogate}`	de helft van een surrogaat-paar in UTF-16
`\p{Cn}`	`\p{Unassigned}`	elk code point waaraan nog geen teken is toegekend

POSIX character classes en hun ASCII en Unicode equivalenten:

POSIX	*betekenis*	*ASCII*	*Unicode*
`[:alnum:]`	alfanumeriek teken	`[a-zA-Z0-9]`	`[\p{L}\{Nl}\p{Nd}]`
`[:alpha:]`	alfabetisch teken	`[a-zA-Z]`	`[\p{L}\p{Nl}]`
`[:ascii:]`	ASCII teken	`[\x00-\x7F]`	`\p{InBasicLatin}`
`[:blank:]`	spatie of tab	`[ \t]`	`[\p{Zs}\t]`
`[:cntrl:]`	stuurteken	`[\x00-\x1F\x7F]`	`\pCc`
`[:digit:]`	cijfer	`[0-9]`	`\p{Nd}`
`[:graph:]`	zichtbaar teken	`[\x21-\x7E]`	`[\p{Z}\p{C}]`
`[:lower:]`	kleine letter	`[a-z]`	`\p{Ll}`
`[:print:]`	zichtbaar teken of spatie	`[\x20-\x7E]`	`\P{C}`
`[:punct:]`	interpunctie en symbolen	`[!"\#$%&'()*+,\-./:;<=>?@\[\\\]^_{|}~]`	`\p{P}`
`[:space:]`	alle witruimte-tekens	`[ \t\n\r\v\f]`	`[\p{Z}\t\n\r\v\f]`
`[:upper:]`	hoofdletter	`[A-Z]`	`\p{Lu}`
`[:word:]`	woord-teken	`[A-Za-z0-9_]`	`[\p{L}\p{Nl}\p{Nd}\p{Pc}]`
`[:xdigit:]`	hexadecimaal teken	`[a-fA-F0-9]`	`[a-fA-F0-9]`

Backreferences:

Telling: capturing groups zijn genummerd vanaf 1 in de volgorde van de haakjes-openen '('

notatie	*betekenis (met k = 1, 2, ..., 9)*
`\k`	gematchte tekst van de *k*-de groep *(alleen* binnen *de regex)*

Lookaround assertions:

Deze worden net als non-capturing groups met haakjes genoteerd, waarbij binnen de haakjes eerst een vraagteken volgt. Deze vormen tellen *niet* mee met de nummering van capturing groups!

notatie	*betekenis*	*naam*
`(?=`*regex*`)`	rechts hiervan moet *regex* matchen	positive lookahead
`(?!`*regex*`)`	rechts hiervan mag *regex* niet matchen	negative lookahead
`(?<=`*regex*`)`	links hiervan moet *regex* matchen	positive lookbehind
`(?<!`*regex*`)`	links hiervan mag *regex* niet matchen	negative lookbehind

Backtracking:

- Probeer te matchen vanaf het begin.
- Meerdere mogelijkheden? Onthoud huidige plek en mogelijkheden.
- Probeer er een.
- Geen match? Keer terug naar laatst onthouden plek en probeer volgende mogelijkheid.
- Geen mogelijkheden meer? Ga verder totdat ofwel een match is gevonden, ofwel de hele match faalt.

Flags:

De meeste regex implementaties bieden de mogelijkheid om het gedrag van de regex machine te beïnvloeden. Dit gaat met een soort 'schakelaars' die *aan* of *uit* staan; ze worden vaak *flags* genoemd. Het precieze gebruik van deze flags is afhankelijk van de omgeving (commando, programmeertaal) waar je de reguliere expressies gebruikt.

Veel programmeertalen gebruiken niet de onderstaande enkelvoudige flag-letters om opties voor het matchen aan te geven. In plaats daarvan moet je als programmeur extra argumenten meegeven aan de functies die het matchen voor je doen. Deze argumenten hebben vaak de vorm van voorgedefinieerde constanten, die in de bijbehorende documentatie worden beschreven.

De volgende flags worden doorgaans ondersteund:

flag	*betekenis*	*uitleg*
`i`	case insensitive	geen onderscheid hoofd- en kleine letters
`g`	global search	doorgaan na de eerste match (bv. bij *search & replace*)
`s`	single-line text	`.` (willekeurig teken) matcht ook op `\n`
`m`	multi-line anchors	anker `^` matcht ook achter `\n` en $ matcht ook vóór `\n`
`x`	comment mode	whitespace negeren; commentaar mag achter #