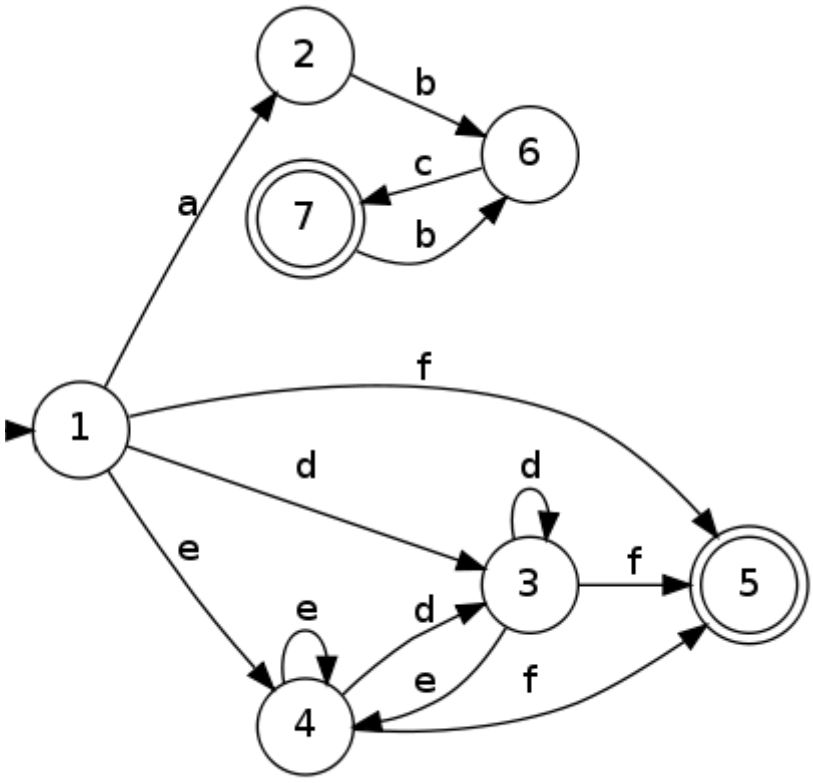
De Ontwikkeling van DuoRex



“De ontwikkeling van een reguliere expressie-implementatie geschreven in Java”

Sander van Veen – 1e jaars student informatica (UvA) – Sep. / Dec. 2009

1. De alternatieve opdracht

1.1 Inleiding

In het eerste jaar en vanaf het eerste blok van de studie Informatica wordt het vak “Programmeren in Java” gegeven. Tijdens het eerste hoorcollege van dit vak, werd er rondgevraagd of er studenten waren die al ervaring hadden met programmeren. Voor die studenten is het mogelijk om in plaats van alle practicumopgaven een alternatieve eindopdracht te maken. De opdracht was het maken van een sudoko-oplosser.

Persoonlijk vond ik dit een moeilijke keuze om te maken, want ik wist niet wat alle practicumopgaven waren en ik vond de alternatieve opdracht geen echte uitdaging. Ik had namelijk het idee dat het maken van een sudoko-oplosser relatief snel gemaakt was. Gelukkig kwam Tim van Deurzen (een van de begeleiders van het Java practicum) met het idee voor een tweede alternatieve opdracht. Deze opdracht was dan wel een stuk complexer: een implementatie van een reguliere expressie-verwerker (= *“regex engine”*) maken in de taal Java.

Het maken van een *regex engine* vond ik een leuke, maar ook serieuze uitdaging. Ik werk met grote regelmaat met reguliere expressies en om een goed presterende expressie te maken, is het van belang om de achterliggende werking te kennen. Ik had nog nooit een regel Java code geschreven, dus naast het feit dat ik kennis zou maken met de taal Java, bood de opdracht me ook de mogelijkheid om de theorie van een *regex engine* te leren.

De opdracht moest af zijn aan het einde van het tweede blok, wat neerkomt op 16 weken. Dit leek mij een reële tijdsduur en ik startte met de opdracht.

1.2 Stappen van de opdracht

Om aan de opdracht te kunnen beginnen, was het uiteraard nodig om meer te weten te komen over de achterliggende theorie van een *regex engine*. Daardoor besloot ik om literatuur over *automata*, *finite state machines* en *regular expressions* op te zoeken en te bestuderen. Later hadden Marin van Beek (andere begeleider), Tim van Deurzen en ik de opdracht in verschillende stappen ingedeeld:

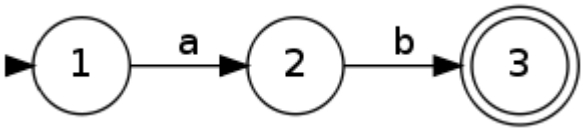
1. Genereren van een *finite state machine* aan de hand van een reguliere expressie.
2. Koppelen van een *finite state machine* aan de te doorzoeken data.
3. Ondersteuning voor numerieke reguliere expressies.
4. Ondersteuning voor *HTML-tag* expressies.
5. Resultaten gegroepeerd teruggeven door middel van een tekstuele tabel.
6. Bonus onderdeel: Matches vervangen door tekst.
7. Verslag schrijven over de opdracht en ontwikkeling.

2. De werking van een *regex engine*

2.1 Basistheorie automaten

Om een *regex engine* te kunnen bouwen, is het noodzakelijk om te weten wat het moet gaan doen. Een *regex engine* is in feite een *finite state machine* toegepast op een dataset. Een *finite state machine* bestaat uit een of meerdere staten, die wel of niet een accepterende toestand geven, en verbindingen (ook wel *"transitions"* genoemd) tussen die staten. Een andere, Nederlandse naam voor een *finite state machine* is "automaat".

Rechts is een illustratie van een *finite state machine* gegeven. De zwarte linker driehoek geeft de startpositie aan van de automaat. De cirkels staan voor de *states* en de pijlen voor de verbindingen. Boven een pijl staat een symbool, wat aangeeft onder welke voorwaarde de verbinding mag worden gebruikt. De cijfers in de cirkels zijn in de illustratie aangebracht om makkelijker naar een staat te kunnen verwijzen. De automaat doet verder niets met de cijfers. Tot slot is er nog een cirkel te zien met een dubbele rand, wat aangeeft dat de staat een accepterende toestand geeft. Illustratie 1 wordt gelezen als het symbool "a" gevolgd door "b" geeft een accepterende toestand.



Illustratie 1: Concatenation: "ab"

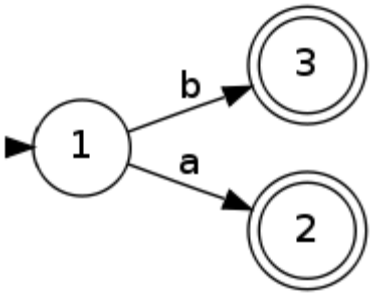
Het doorzoeken van een dataset gebeurt met een automaat en een accepterende toestand zorgt voor het vinden van een resultaat. Dan is er een mogelijkheid om door te zoeken naar meer resultaten of de zoekactie te staken, wat afhangt van het programma. Als de automaat helemaal geen accepterende toestand bereikt, is er ook geen resultaat gevonden.

2.2 Veel voorkomende automaten

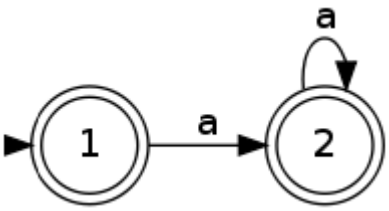
Een reguliere expressie onderscheidt zich van een normale zoekactie door het mogelijk te maken om een repetitie van symbolen en/of een keuze uit symbolen op te kunnen geven.

Illustratie 1 is een voorbeeld van *Concatenation*. Dat staat voor een verzameling symbolen A gevolgd door een verzameling symbolen B. Een verzameling is in deze context een lijst met symbolen die het mogelijk maken om een verbinding te kunnen gebruiken. Notatie: "AB".

Illustratie 2 geeft een automaat weer die *Alternation* wordt genoemd, omdat er een keuze kan worden gemaakt. Door middel van een staand streepje (= "|"), kan er worden uitgedrukt dat de verzameling van symbolen links van het staande streepje een andere keuze is dan de verzameling van symbolen rechts. Een verzameling wordt in een expressie aangegeven door meerdere symbolen tussen "(" en ")", of een enkel symbool zonder haakjes.



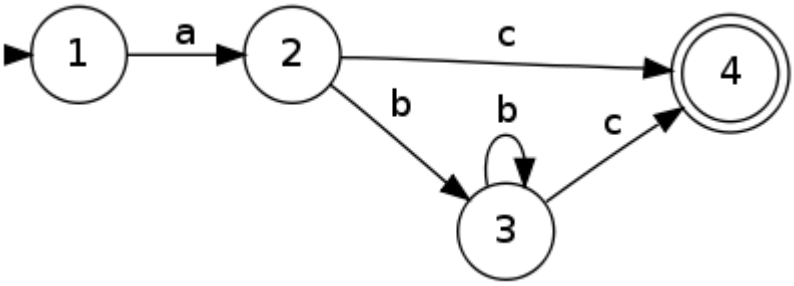
Illustratie 2: Alternation: "a|b"



Illustratie 3: Kleene star: “a”*

Zoals eerder in deze paragraaf genoemd, kan er gebruik worden gemaakt van repetitie. Er bestaan meerdere varianten van repetitie, maar de belangrijkste is de *Kleene star*. De notatie voor een reguliere expressie is “A*”, waarbij A staat voor een verzameling van symbolen. Met een *Kleene star* is het mogelijk om nul, een of meerdere keren voorgaande verzameling te herhalen. Staat 2 heeft een verbinding naar zichzelf en dat wordt reflexief genoemd en geeft aan dat bij het nemen van de verbinding, bij dezelfde staat wordt teruggekeerd.

Illustratie 3 is een voorbeeld van een *Kleene star*. De automaat start in staat 1. Dat is een accepterende toestand. De automaat heeft nu een match gevonden. Dit voorbeeld is niet erg praktisch, want elke tekst zou hierdoor een match opleveren. De Kleene star is pas goed te gebruiken in combinatie met eerder genoemde automaten (*concatenation* en *alternation*). Een combinatie van *concatenation* en de *Kleene star* kan bijvoorbeeld worden genoteerd als “AB*C”, waarbij A, B en C staan voor een verzameling symbolen. Een voorbeeld van die automaat ziet er dan uit als illustratie 4. Om een resultaat te krijgen, dient er eerst een “a” te worden gevonden, gevolgd door nul, een of meerdere keren een “b” en tot slot een “c”.



*Illustratie 4: Finite state machine voor expressie “ab*c”*

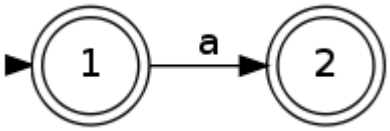
2.3 Toegevoegde operatoren

Bij de Java opdracht kreeg ik een link naar een relatief eenvoudige implementatie van een *regex engine* geschreven in de taal C++.[1] Ondanks dat ik geen ervaring had in C++ (maar wel in PHP, dat een soortgelijke syntaxis heeft), heb ik die code omgezet naar Java en ben ik gaan experimenteren met het toevoegen van andere automaten. Het omzetten van de C++ code naar Java code ging niet erg soepel, omdat ik de taalspecifieke functies van beide talen niet goed beheerste. Na meerdere testrondes is het uiteindelijk toch gelukt. De ontwikkelde Java implementatie heeft daarom functies die grotendeels overeenkomen met de C++ code.

In paragraaf 2.2 zijn enkele automaten genoemd, die kunnen worden gezien als een basis. Om de eindgebruiker de mogelijkheid te bieden om meer te kunnen uitdrukken in een reguliere expressie, is het handig om de verschillende typen operatoren uit te breiden. Dit maakt de expressie korter, wat in de meeste gevallen zorgt voor een betere overzichtelijkheid.

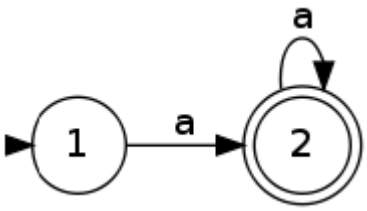
1 Simpele regex engine: http://www.codeproject.com/KB/recipes/OwnRegExpressionsParser.aspx

Een veelgebruikte operator, die is af te leiden uit *alternation*, is 0 of 1 keer de voorgaande verzameling (zie illustratie 5). De notatie is "`A?`", waarbij A een verzameling van symbolen is. Deze automaat kan ook worden gegenereerd door de reguliere expressie "`(A|)`", want na het staande streepje bevindt zich een verzameling die geen symbolen bezit. Het is niet nodig om haakjes te gebruiken bij de notatie "`A?`", terwijl een *alternation* dit wel nodig heeft, als er een of meerdere verzamelingen voor en/of na de automaat komen. Simpel voorbeeld: "`ab?c`" ≠ "`ab|c`".



Illustratie 5: Automaat "`a?`"

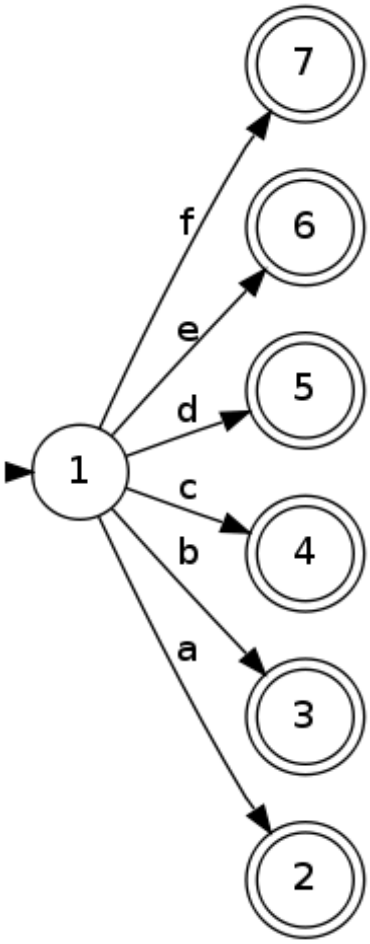
Naast de toevoeging van het vraagteken is er een plusteken toegevoegd. Het plusteken staat voor een of meerdere van de voorgaande verzameling symbolen. De notatie is "`A+`". Illustratie 6 is een voorbeeld van het gebruik van een plusteken. Het plusteken kan worden gezien als een uitbreiding op de *Kleene star*. Het plusteken verkort enkel de notatie, want met een *Kleene star* en *concatenation* kan hetzelfde worden bereikt. Voorbeeld: de expressie "`aa*`" geeft dezelfde automaat als bij de expressie "`a+`". Dit voorbeeld geeft misschien niet de indruk dat het toevoegen van een plusteken een grote hoeveelheid tekens kan besparen, maar als "`a`" wordt vervangen door een aantal subexpressies, wordt duidelijk dat die subexpressies niet twee keer hoeven worden geschreven. Dat scheelt dus de helft van de tekens om dezelfde automaat te genereren.



Illustratie 6: Automaat "`a+`"

De laatste toevoeging is de *set*. Daarmee kan een *alternation* van een opeenvolgende reeks van ASCII tekens[2] worden verkort tot slechts het eerste en laatste teken van de reeks. Een set wordt gestart met "`[`" en eindigt met "`]`". Binnen de blokhaken heeft het min-teken (een "`-`") een andere betekenis: het symbool links van het min-teken start een reeks en het symbool rechts stopt de reeks. Voorbeeld: de *alternation* "`a|b|c|d|e|f`" kan geschreven worden als de set "`[a-f]`" (zie illustratie 7). De symbolen, binnen een set, die links of rechts geen min-teken hebben, worden gezien als losstaande tekens. Voorbeeld: de set "`[a-dxy]`" is hetzelfde als de *concatenation* "`a|b|c|d|x|y`".

Er zijn nog veel meer mogelijke operatoren om toe te voegen, maar de opdracht is te voltooien met de bovenstaande toevoegingen. Het toevoegen van meer operatoren maakt de *regex engine* ook steeds complexer.



Illustratie 7: Automaat "`[a-f]`"

2 Lijst van alle ASCII tekens: http://www.asciitable.com/

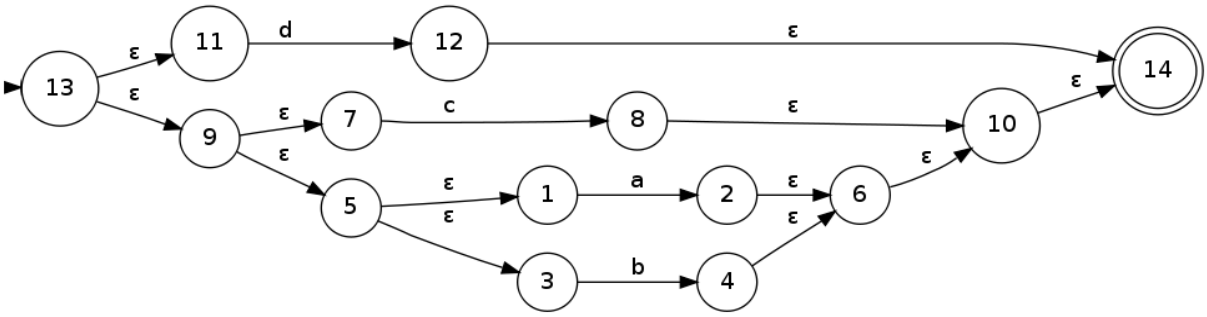
3. De uitwerking

3.1 Regex parser

De voorbeeld implementatie van een *regex engine* in C++ (zie hoofdstuk 2 §3 *"Toegevoegde operatoren"*) verwerkt een reguliere expressie op gelijke wijze als het parsen van een algebraïsche vergelijking. De functie "`isPresedence(char left, char right)`" bepaalt of de operator links (= "`char left`") beter bindt dan aan de operator rechts (= "`char right`"). Bij gelijke operatoren, wordt er ook *true* terug gegeven. Anders *false*.

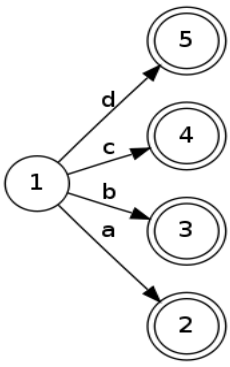
Om de opdracht te kunnen voltooien, moesten de operatoren "`?`", "`+`", "`[`", "`-`" en "`]`" aan de *regex* parser worden toegevoegd. De functie "`prepareConcatenation(String regex)`" bewerkt de regex voor, door aan te geven waar *concatenation* plaatsvindt. Op die plek wordt "`(char) 0`" ingevoegd. Deze functie is ook uitgebreid met het omzetten van *sets* in *alternation*. Dit voorwerk zorgt er voor dat de generatie van een automaat relatief simpel blijft.

Nu is het tijd voor het genereren van de automaat. Eerst wordt de NFA (Non-deterministic Finite-state Automaton) gegenereerd en daaruit gevolgd door de DFA (Deterministic Finite-state Automaton). Om te voorkomen dat dit verslag een boek wordt, is er voor gekozen om geen uitgebreide beschrijving te geven van de omzetting van NFA ⟶ DFA of het genereren van de NFA. Het idee is dat een NFA bestaat uit verbindingen die zonder symbool kunnen worden gebruikt. Deze verbindingen worden *epsilon transitions* genoemd en worden aangegeven door middel van een pijl met de kleine griekse letter "ε" erboven (zie illustratie 8).



Illustratie 8: Non-deterministic Finite-state Automaton voor "`[a-d]`" of "`a|b|c|d`"

Een *epsilon transition* kan worden gezien als een verbinding die kan worden gebruikt om later naar terug te keren, als de automaat in een niet accepterende toestand terecht komt. Om er voor te zorgen dat er niet steeds hoeft te worden teruggesprongen, wordt de NFA in een DFA omgezet. Bij de omzetting van NFA ⟶ DFA worden de *epsilon transitions* uitgewerkt. Illustratie 9 toont de uitwerking van NFA ⟶ DFA met de NFA van illustratie 8. Het genereren van een NFA is dus in feite een tussenstap om het genereren van de DFA te vergemakkelijken. Dit algoritme komt uit de implementatie in C++ en is vertaald in Java code.



Illustratie 9: De DFA voor "`[a-d]`"

3.2 Commandline

Daarnaast was er ook een tweede parser nodig voor het verwerken van de argumenten van de *commandline*. Deze parser bestond in zijn geheel niet in de voorbeeld implementatie. De parser bevindt zich in de *public class DuoRex* te vinden in “DuoRex.java”. Deze *class* start, na de commandline te hebben geparsed, de *regex* parser.

De commandline parser maakt het mogelijk om bepaalde *properties* te wijzigen, bij het starten van het programma. Een voorbeeld is het argument “`-graphs`” en geeft aan dat de grafen van de NFA en DFA dienen te worden gegenereerd. Dit argument is steeds gebruikt bij het maken van de illustraties uit dit verslag.

Een ander voorbeeld is “`-benchmark`”. Dit argument zorgt voor het tonen van het aantal millisecondes tussen de verschillende fases van het programma: voorbereiden regex, generatie NFA, omzetting NFA ⟶ DFA, zoeken naar resultaten, het eventueel vervangen van de resultaten door een andere tekst en uiteindelijk het weergeven van de resultaten. In hoofdstuk 4 §1 *“Mogelijke verbeteringen”* komt het gebruik van dit argument verder aan bod.

3.3 Modifiers

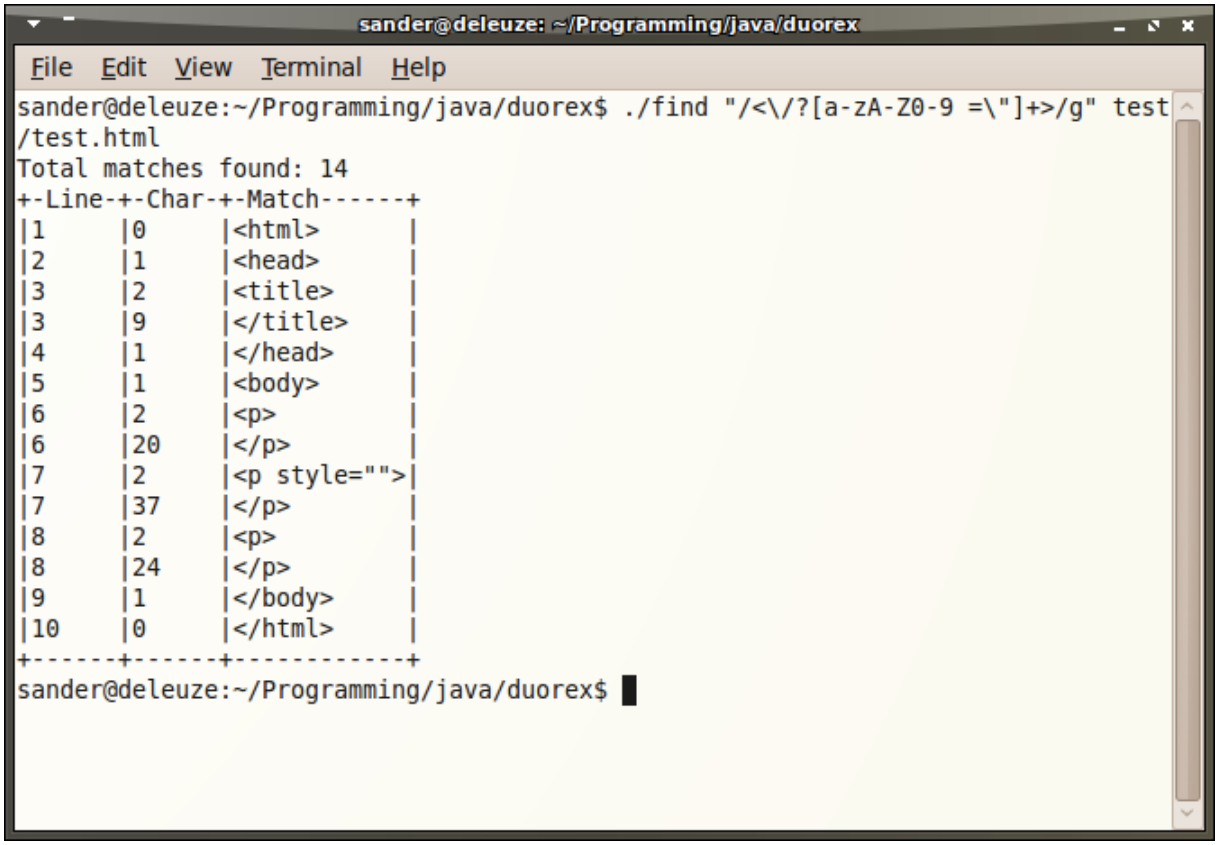
Bij veel *regex engines* is het gebruikelijk om zogenoemde *modifiers* toe te kunnen voegen aan een *regex*. Dit past de werking van de engine aan. Zo geeft de *modifier* “`g`” (wat staat voor *global*) aan dat er niet moet worden gestopt met zoeken, als een resultaat is gevonden. *Modifiers* worden van de *regex* gescheiden door twee *delimiters*. Het eerste teken van de expressie wordt gezien als *delimiter* (en dus genegeerd) en dat teken bepaalt wanneer de expressie stopt. Daarna volgen de *modifiers*. Een voorbeeld van een reguliere expressie is: “`/<[a-zA-Z0-9]+>/g`”. De slashes (= “`/`”) dienen als *delimiters* en “`g`” is een *modifier*.

Om de fase “zoeken naar resultaten” te vergemakkelijken, is er de mogelijkheid om alle mogelijke combinaties van geldige resultaten te accepteren. De modifier “`a`”, die dus is bedacht voor het tonen van extra informatie, geeft een beter inzicht in de werking van het vinden van resultaten. Normaal gesproken zouden alleen de langste *matches* worden gevonden: alle tussenresultaten worden weggegooid, omdat ze in de meeste gevallen overbodig zijn. Het vinden van de langste resultaten, wordt ook wel *greediness* genoemd. De *engine* probeert de meest lange *matches* te vinden. Er bestaat ook een modifier voor *laziness* (het tegenovergestelde), maar die is niet geïmplementeerd in deze opdracht. Als er enkel een *modifier* “`a`” wordt gegeven, zal de engine wel stoppen na de eerste match (en is dus in feite een soort *laziness*). Om daadwerkelijk alle mogelijke combinaties te kunnen tonen, dient zowel de *modifier* “`a`” als “`g`” te worden opgegeven. Zonder “`g`” stopt de *engine* immers na het eerst gevonden resultaat.

3.4 De matcher

De *matcher* zorgt voor het vinden van de resultaten. Kortweg gezegd leest het een (tekst)bestand in met een buffer en loopt vervolgens de paden van de DFA af. De *modifiers* bepalen of de *matcher* dient te stoppen na het vinden van een match. In deze implementatie is het ook mogelijk om alle mogelijke combinaties als match te beschouwen, door de modifier “`a`” toe te voegen.

Een van de eisen aan de opdracht was dat de gevonden resultaten werden weergegeven in een tekstuele tabel. De reguliere expressie werd immers met de commandline aangeroepen en dan is het wel zo handig als de resultaten overzichtelijk worden teruggegeven. De tabel bestaat uit drie kolommen: “*line*”, “*char*” en “*match*”. Logischerwijs staat “line” voor de regel waar de match is gevonden. De kolom “*char*” staat voor het aantal tekens vanaf het begin van de regel en “*match*” toont het gevonden resultaat.



Afbeelding 1: De tekstuele tabel na het uitvoeren van regex “`/<\/?[a-zA-Z0-9 =\"]/g`”.

Door middel van de functie “`Math.ceil(Math.log(n))`” toe te passen op de *line* en *char* van alle resultaten, werd de maximale lengte van de kolommen *line* en *char* bepaald. De lengte van de kolom *match* is gelijk aan de lengte van de langste *match*.

Tot slot was er nog “matches vervangen door een tekst.” Nu de matches waren gevonden, inclusief regel en startpositie, was het vrij eenvoudig om de matches te vervangen. Wel moest er op worden gelet dat de startpositie verandert bij meerdere matches op een regel.

4. Een terugblik op de opdracht

4.1 Mogelijke verbeteringen

Achteraf heb ik ingezien dat het omzetten van een *set* naar *alternation*, geen goed idee was. De implementatie in C++ maakte gebruik van een symbool per verbinding en dat heb ik aangehouden. Hierdoor ontstond het probleem dat grote sets, bijv. “`[a-zA-Z0-9!@#$%&*?~{}]`”, voor enorm veel transities zorgen, in dit geval 26 + 26 + 10 + 10 = 72 verbindingen. Deze verbindingen zouden vervangen kunnen worden door slechts een verbinding, als de set werd gebruikt als verzameling van symbolen. Het probleem zorgt er met name voor dat *regexes* met grote sets erg traag zijn bij het omzetten van NFA ⟶ DFA.

Een ander punt wat zou kunnen worden verbeterd, is het bieden van ondersteuning voor *greediness* en *laziness*. In veel *regex engines* is het mogelijk om aan te geven of een repetitie respectievelijk de meest lange match of juist de kortste match moet vinden. Dit gebeurt als er een “`?`” na bijv. een “`+`” of “`*`” wordt geplaatst. Deze optie is niet geïmplementeerd, omdat het geen onderdeel was van de opdracht, maar het zou er wel voor zorgen dat de *engine* vele malen complexer zou maken. In deze implementatie wordt dus alleen “*greedy*” toegepast. Het is wel mogelijk om een soort “*laziness*”-effect te krijgen met enkel de “`a`” modifier.

4.2 Nawoord

De opdracht was om een *regex* implementatie te maken en niet om een *regex* implementatie te maken die gelijk of beter presteert aan/dan de bestaande *engines*. Het project bestaat nu uit 1600 regels Java code en er zijn meer regels geschreven, omdat sommige functies moesten worden herschreven. Toen alle onderdelen van de opdracht waren geïmplementeerd, ben ik gestopt met de uitbreiding van de *engine*. Daarnaast zijn er extra opties (genereren van grafen, “`a`” en “`g`” *modifiers* en *bash scripts* als *shortcuts* voor het aanroepen van de *engine*) aangebracht, die allemaal geen onderdeel waren van de opdracht.

Ik heb er voor gekozen om de eigen “`r`” modifier ("reverse mode") niet verder te ontwikkelen. Het maakt het project veel complexer, zonder echt een significante tijdswinst te bieden. Het algoritme zou namelijk alleen in bepaalde omstandigheden goed werken.

Naast dat ik heb leren programmeren in Java (voor deze opdracht had ik nog geen een regel Java code geschreven, maar wel veel JavaScript en PHP), heb ik me ook kunnen verdiepen in de werking van een *regex engine* (inclusief de theorie erachter), dot graphs en bash scripts. Deze kennis kan ik ook buiten de taal Java gebruiken.

Ik wou graag Tim van Deurzen en Marin van Beek bedanken voor het aanbieden van deze alternatieve opdracht. Daarnaast hebben zij het voor mij mogelijk gemaakt om enkele lessen "Automaten" te kunnen volgen! Ik vond het een zeer leerzaam project en ik denk dat ik meer bij deze opdracht heb geleerd, dan als ik voor de "sudoko-oplosser" had gekozen.