|  |
| --- |
| Functioneel Paradigma\_  Algoritmes, Paradigma’s en Programmeertalen |

Mike Doornenbal - 1628421

[kies een datum]

INHOUDSOPGAVE

[1 Challenge 3](#_Toc210134815)

[2 Onderzoek 4](#_Toc210134816)

[3 Implementatie 6](#_Toc210134817)

[4 Reflectie 7](#_Toc210134818)

[5 Conclusie 8](#_Toc210134819)

[6 Bronvermelding 9](#_Toc210134820)

[10](#_Toc210134821)

# Challenge

Voor deze opdracht heb ik gekozen om RLE-compressie te implementeren. Van de vooraf gegeven opties, leken degene die met compressie te maken hebben de meest interessante. Uiteindelijk heb ik RLE-compressie gekozen over LZW-compressie, omdat ik (na het bestuderen van de Wikipedia-pagina’s voor beide compressie-algoritmen) bij RLE-compressie al snel een idee kreeg bij hoe het algoritme comprimeert.

Ondanks het initiële begrip, denk ik dat deze opdracht mij alsnog een flinke uitdaging zal bezorgen. Het begrijpen van een een puzzel betekent immers niet dat je direct de oplossing hebt, of überhaupt de weg naar de oplossing ziet. Compressie boeit mij al langer omdat het een beetje voelt als magie: je slaat exact dezelfde data op maar het neemt minder ruimte in beslag en door creatief om te gaan met de resulterende data kan je de originele data weer terugkrijgen. Natuurlijk bestaat er ook lossy compressie, waarbij data verloren gaat omwille van ruimtebesparing. Dit voelt minder speciaal en uitdagend, omdat je feitelijk strategisch data verwijdert zonder de verwachting deze later terug te krijgen. Van de andere kant is deze vorm van compressie weer bijzonder door andere redenen, zoals het feit dat het dataverlies soms niet makkelijk te merken is.

Ik heb nog nooit eerder geprobeerd data te comprimeren (en zeker niet in een functionele taal) en daarom lijkt dit me tegelijkertijd een leuke maar ook zeker een uitdagende challenge.

# Onderzoek

Om te beginnen ben ik online op zoek gegaan naar een kant en klare uitwerking van RLE-compressie in Haskell. Deze had ik snel gevonden, wat betekent dat ik daarvan kan leren hoe RLE-compressie in Haskell precies werkt wanneer ik er zelf niet uitkom. Ik hoop hier geen gebruik van te hoeven maken, maar nood breekt wet en als ik begrijp wat de uitwerking doet en hoe deze tot stand is gekomen, kan ik altijd proberen deze zelf op een andere manier te schrijven.

Een zekerheid is wel dat ik op dit moment geen compressiealgoritme in Haskell ga kunnen schrijven, omdat ik simpelweg de syntax van Haskell niet goed weet. Daarom heb ik besloten om een eerste uitwerking te maken in Java. Die probeer ik dan zo functioneel mogelijk te maken, zodat ik deze direct kan vertalen naar Haskell.

Allereerst ben ik begonnen met het uitschrijven van de stappen die ik moet zetten om tot een werkend algoritme te komen. Dit heb ik gedaan in codecommentaar, zodat ik meteen kan zien wat de volgende stap in het proces moet zijn mocht ik vastlopen. Daarna ben ik begonnen met het schrijven van een implementatie in Java op een non-functionele wijze. Het resultaat daarvan is [hier](https://github.com/MikTheMilkMan/APP-Functioneel-Paradigma/blob/f1dd7cba50ce26fadf72d17c0f7e6a094210a221/Programma/java_tests/src/main/java/nl/han/APP/Challenge/Challenge.java) te zien. Deze implementatie werkte niet 100% goed, maar het was goed genoeg voor een begin.

Vervolgens heb ik geprobeerd deze implementatie te herschrijven naar eentje die functioneel is. Het resultaat daarvan is [hier](https://github.com/MikTheMilkMan/APP-Functioneel-Paradigma/blob/32d8a9fdf4a85e4003695e6a7a7ada894d84c1de/Programma/java_tests/src/main/java/nl/han/APP/Challenge/Challenge.java) te zien. Ook hier ben ik niet geheel tevreden mee, maar wederom is dit een goede tweede stap. Deze implementatie lijkt wel 100% goed te werken (gebaseerd op mijn gelimiteerde testcases), dus vanuit dat opzicht kan ik hier tevreden mee zijn.

De volgende stap is het vertalen van de Java-code naar Haskell. Dit blijkt een aanzienlijke uitdaging, gezien de syntax van Haskell enorm verschilt van die van iedere andere taal waar ik op dit moment bekend mee ben. Het feit dat ik er moeite mee heb, is in [deze](https://github.com/MikTheMilkMan/APP-Functioneel-Paradigma/commit/ebfbc9712c44a1939b326b97c24838447c6b5f1f) commit te lezen.

Ik dacht op [dit](https://github.com/MikTheMilkMan/APP-Functioneel-Paradigma/blob/3fe34b988b77902476930033961ab0705316aba6/Programma/hello.hs) punt klaar te zijn met het implementeren van de compressie. Later bleek echter dat dit niet klopte. Hier kom ik op een later moment op terug. Eerst kwam het schrijven en implementeren van een decompressie-functie. Door de Haskell-kennis die ik heb opgedaan tijdens het schrijven van de compressie-functie, heb ik de decompressie-functie direct in Haskell geschreven. Dit bleek uiteindelijk de juiste keuze, omdat ik de Haskell-syntax inmiddels dusdanig onder de knie had dat het gevoelsmatig tijd bespaarde ten opzichte van het eerst uitprogrammeren van de functionaliteit in Java.

De [eerste versie](https://github.com/MikTheMilkMan/APP-Functioneel-Paradigma/blob/f52e5afac098ca233fbfa939ae02ca894fd53e7c/app/Main.hs) van de decompressie-functionaliteit werkte al vrij snel. Het probleem zat hem echter in het decomprimeren van Strings met meer dan 9 gelijke achtereenvolgende karakters. Omdat de waarden in de gecomprimeerde String direct achter elkaar werden gezet, bestond er geen simpele manier om de karakters en bijbehorende aantallen te scheiden van de andere paren. Althans, dat dacht ik. Uiteindelijk heb ik dit opgelost door gebruik te maken van een package genaamd “Split”. Hier staan functies in die op basis van een instelbare “delimiter” een String op kan splitsen. Door die “delimiter” in te stellen op alle letters van het alfabet, worden de gecomprimeerde Strings opgesplitst in paren van karakter en aantal. De verbeterde oplossing is [hier](https://github.com/MikTheMilkMan/APP-Functioneel-Paradigma/blob/386f6107650f70e8a384c67db37426fcdfd540d1/app/Main.hs) te zien.

Zoals ik eerder zei, werkte het comprimeren nog niet volledig naar behoren: enkel Strings waar niet meer dan 9 gelijke karakters achter elkaar stonden, werden compleet gecomprimeerd. Dit kwam doordat de functie die ik gebruikte om de waarde van de teller in de String te zetten, “intToDigit”, enkel het eerste getal van de teller converteerde naar een Char, die vervolgens in de String werd gezet. Wanneer de teller dus op 21 stond, werd enkel de 2 in de gecomprimeerde String gezet. Hoewel ik eerst bedacht had om ook hier gebruik te maken van een package genaamd “toString”. Bij het lezen van de logica, kreeg ik het gevoel dat ik hier zelf ook prima een functie voor kon schrijven. Echter, toen ik daarmee begonnen was realiseerde ik mij dat de ingebouwde functie “show” het meegegeven argument teruggaf als String. Als ik dus het correcte karakter in een array zette (en er op die manier een String van maakte van 1 karakter lang) en daar de uitkomst van “show aantal” achter zette, had ik een volledige String die het correcte aantal bevatte. De resulterende code is [hier](https://github.com/MikTheMilkMan/APP-Functioneel-Paradigma/blob/43aa12b1de86722601decfd59f4b76e12cf1bf2b/app/Main.hs) te zien.

Als laatste heb ik, na overleg met de docent, ook twee functies geschreven voor het comprimeren en decomprimeren van meerdere regels aan Strings tegelijkertijd. Daarnaast heb ik de andere code in [dezelfde commit](https://github.com/MikTheMilkMan/APP-Functioneel-Paradigma/blob/5c17ccd46b3e6249ff4789545194c46c54d876e3/app/Main.hs) aangepast zodat deze er netter uitziet (if-then-else vervangen door guards, overbodige prints verwijderd).

# Implementatie

* Recursie
* Pure functies (want tis Haskell)
* Immutability (want tis Haskell)
* Lazy evaluation (want tis Haskell)
* First-class functions

# Reflectie

# Conclusie

Samenvatting van de belangrijkste (leer)punten

# Bronvermelding

Bronvermelding

<https://www.seas.upenn.edu/~cis1940/spring13/lectures.html>

