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

Mike Doornenbal - 1628421

3 oktober 2025

INHOUDSOPGAVE

[1 Challenge 3](#_Toc210388670)

[2 Onderzoek 4](#_Toc210388671)

[3 Implementatie 6](#_Toc210388672)

[3.1 Immutability 6](#_Toc210388673)

[3.2 Recursie 6](#_Toc210388674)

[3.3 Pure functies 6](#_Toc210388675)

[3.4 Lazy evaluation 7](#_Toc210388676)

[4 Reflectie 8](#_Toc210388677)

[5 Conclusie 9](#_Toc210388678)

[6 Bronvermelding 10](#_Toc210388679)

[6.1 Bronnen voor kennis over Haskell 10](#_Toc210388680)

[6.2 Bronnen voor kennis van concepten 10](#_Toc210388681)

# 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

Bij de implementatie van deze challgenge, heb ik gebruik gemaakt van verschillende functionele concepten. In dit hoofdstuk staat een uitleg van deze concepten, waar ik gebruik heb gemaakt van een concept en waarom ik dat heb gedaan. De regelnummers refereren naar de regels in [dit bestand](https://github.com/MikTheMilkMan/APP-Functioneel-Paradigma/blob/main/app/Main.hs).

## Immutability

Een variabele die immutable is, kan na het toewijzen van een waarde niet aangepast worden. Variabelen in Haskell zijn altijd immutable.

Op regels 40, 43, 46 en 49 staan variabelen gedeclareerd. Deze variabelen doen dienst als testvariabelen die, voordat de code geëxecuteerd wordt, aangepast kunnen worden door een gebruiker om aan te tonen dat de compressie- en decompressie-functionaliteiten naar behoren functioneren. Dit maakt het gemakkelijk om de variabelen waarmee de functionaliteit wordt aangeroepen aan te passen om te controleren of de functionaliteit ook intact blijft met andere variabelen.

Doordat variabelen naderhand niet aan te passen zijn, is het niet mogelijk om een simpele for-loop te schrijven (zoals dat in andere talen wel het geval is). In plaats daarvan wordt daarom gebruik gemaakt van recursie.

## Recursie

Een recursieve functie is een functie die zichzelf aanroept. Recursie kan voor veel verschillende zaken gebruikt worden, maar ik heb het voornamelijk gebruikt ter vervanging van for-loops. Omdat variabelen in Haskell immutable zijn, is het überhaupt niet mogelijk om gebruik te maken van een reguliere for-loop.

We nemen de functie die op regel 56 geschreven staat als voorbeeld. In deze functie staan op regel 67 en 69 recursieve aanroepen. In dit geval heb ik gekozen voor een recursieve aanroep, omdat deze functie ieder karakter van een meegegeven String moet doorlopen om te controleren of deze hetzelfde is als het vorige karakter en, afhankelijk van de conclusie, andere logica uitvoert. Omdat ik in een non-immutable taal een for-loop zou gebruiken om ieder karakter individueel te controleren, heb ik besloten om het in deze implementatie met recursie op te lossen.

## Pure functies

Een pure functie is een functie die, gegeven dezelfde input, altijd dezelfde output zal geven. Deze functies hebben geen bijwerkingen. Omdat variabelen in Haskell immutable zijn, moet iedere functie per definitie een pure functie zijn.

Laten we kijken naar één van de simpelere functies die ik heb geschreven: characterGiver. Deze staat op regel 104. Wanneer deze functie wordt aangeroepen, worden altijd een karakter en een aantal meegegeven. Als het aantal groter is dan 1, wordt het karakter teruggegeven in een lijst, waarna de characterGiver-functie recursief wordt aangeroepen. Het aantal wordt bij deze recursieve aanroep verminderd met 1. Zodra het aantal gelijk is aan 1, wordt enkel nog het gevraagde karakter teruggegeven in een lijst. Voor Haskell is dit hetzelfde als het teruggeven van een String. Ongeacht de verdere uitgevoerde acties in dit programma, zal deze functie bij een overeenkomende input altijd dezelfde output geven.

## Lazy evaluation

Bij Lazy Evaluation wordt een uitdrukking pas geëvalueerd wanneer deze daadwerkelijk aangeroepen wordt of anderszins nodig is. Dit heeft als voordeel dat, bijvoorbeeld, een lijst met een oneindige getallenreeks niet direct wordt geïnstantieerd met het maximum aantal getallen dat in die lijst past. Ook wanneer een lijst met alle getallen van 1 tot en met 10,000 wordt geïnstantieerd, wordt de lijst enkel gevuld wanneer dit nodig is. Dit spaart geheugen en rekenkracht uit wanneer het maximum aantal getallen niet gebruikt wordt.

Op regel 88 wordt de functie “decode” aangeroepen. Deze functie krijgt als eerste argument het resultaat van de functie “split” mee. In dit geval splitst “split” een String (lijst van karakters) op ieder punt waar een letter staat. Dit gebeurt doordat de delimiter die “split” gebruikt aangegeven staat als een lijst van alle karakters. Doordat Haskell by default gebruik maakt van Lazy Evaluation, wordt deze lijst gevuld tot en met het vereiste karakter wanneer de lijst gebruikt of aangeroepen wordt.

# Reflectie

Ik had veel moeite met het beginnen met werken aan deze opdracht. Dit had twee hoofdredenen. De eerste reden is dat ik geen idee had hoe ik RLE-compressie moest uitwerken in code. De tweede reden is dat ik geen idee bij de syntax van Haskell of hoe ik het verder goed moest gebruiken. Een derde, minder grote reden was dat het lokaal werkend krijgen van Haskell ook een redelijke uitdaging was. Dat ik geparalyseerd raak door hoe overweldigend ingewikkeld een taak lijkt te zijn, overkomt mij vaker.

Normaliter lukt het mij niet om uit deze paralyse te ontsnappen, waardoor ik helemaal niets doe. Dit keer was echter anders. Omdat ik na de projecten van het afgelopen schooljaar nog redelijk in de mindset van het “opsplitsen van te grote taken die derhalve de maximaal aan een taak te besteden tijd overschrijden“ zit, zocht ik naar een manier om dat hier ook te doen. Die oplossing heb ik uiteindelijk gevonden in het splitsen van het schrijven van de logica en het schrijven van Haskell-code.

Hoewel het implementeren van een nog onbekende functionaliteit in een nog onbekende taal een gigantisch obstakel leek, vielen de individuele onderdelen reuze mee. Ik begon met het implementeren van RLE-compressie in Java, een taal waar ik inmiddels de nodige uren aan ervaring mee heb opgedaan.

De basis van dit algoritme bestond vrij snel. Dat is in [deze](https://github.com/MikTheMilkMan/APP-Functioneel-Paradigma/blob/32d8a9fdf4a85e4003695e6a7a7ada894d84c1de/Programma/java_tests/src/main/java/nl/han/APP/Challenge/Challenge.java) commit te zien. Vervolgens heb ik de Java-code vrij direct in Haskell overgenomen. Dit werkte uiteraard niet direct, omdat de syntax van Java en Haskell aanzienlijk verschilt. Een beginsel van deze vertaling is [hier](https://github.com/MikTheMilkMan/APP-Functioneel-Paradigma/blob/ebfbc9712c44a1939b326b97c24838447c6b5f1f/Programma/hello.hs) te zien. Stapsgewijs leerde ik steeds meer over de syntax en semantiek van Haskell, en hoe deze verschilde van de syntax en semantiek die ik voor deze opdracht kende. In [dit](https://github.com/MikTheMilkMan/APP-Functioneel-Paradigma/blob/126876d77026297846e4fe6b3ee7bc1bc4ec37d2/Programma/hello.hs) bestand staan op regel 33 en 41 twee implementaties van een functie genaamd “compression”. Hier is nog te zien hoe Java-ig de functionaliteit op dat moment was. De eerstvolgende commit bevatte een werkende implementatie van het algoritme. Die implementatie is [hier](https://github.com/MikTheMilkMan/APP-Functioneel-Paradigma/blob/3fe34b988b77902476930033961ab0705316aba6/Programma/hello.hs) te zien op regel 23. Tevens staan daar op regel 37 en 47 de eerdere, onjuist vertaalde implementaties.

Gedurende een proces wat in totaal over 3 dagen verspreid was, had ik mijn Java-implementatie vertaald naar Haskell. Als je deze versie van de functie vergelijkt met de [uiteindelijke versie](https://github.com/MikTheMilkMan/APP-Functioneel-Paradigma/blob/main/app/Main.hs) (regel 56), is direct duidelijk dat de eerste versie niet perfect (laat staan goed) was. Het was echter goed genoeg om op verder te itereren. Het doel was immers niet om direct een perfect compressie-algoritme te implementeren, maar om genoeg kennis op te doen over Haskel dat ik de huidige implementatie kon verbeteren en de implementatie van een decompressie-algoritme direct in Haskell kon schrijven, en dat doel had ik behaald.

Achteraf gezien ben ik trots dat ik niet stil ben blijven staan bij de gedachte dat het uitwerken van deze opdracht enorm intimiderend leek en dat ik in plaats daarvan de tijd heb genomen om een manier te verzinnen waarop ik deze opdracht in kleinere stukken aan kon vliegen.

Terugkijkend op deze opdracht, heeft het enorm geholpen dat ik een challenge heb gekozen waarvan ik snel begreep wat de bedoeling was. De obstakels van het niet kennen van de taal en het niet weten hoe ik de implementatie precies zou moeten schrijven, zou ik bij iedere opdracht tegen zijn gekomen. Als ik bijvoorbeeld voor LZW-compressie had gekozen in plaats van RLE-compressie, had ik eerst onderzoek moeten doen naar hoe dat algoritme überhaupt werkt. Doordat dat onderzoek mij bespaard is gebleven, was de drempel om te beginnen lager dan deze had kunnen zijn.

# Conclusie

Met het maken van dit beroepsproduct heb ik een aantal dingen geleerd. Zo weet ik nu hoe (één van de vormen van) bestandscompressie werkt, maar ook hoe je een intimiderende opdracht minder intimiderend kan maken door het benodigde werk zo ver mogelijk op te splitsen en de onderdelen op die manier behapbaar te maken.

Daarnaast heb ik ook geleerd dat recursie niet per definitie slecht is en dat programmeren in een taal als Haskell, waar immutability en pure functies centraal staan, een andere manier van denken vereist dan programmeren in Java, Python of Javascript.

# Bronvermelding

## Bronnen voor kennis over Haskell

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

<https://learnyouahaskell.com/recursion#hello-recursion>

<https://hackage.haskell.org/package/split-0.2.5/docs/Data-List-Split.html>

<https://hackage.haskell.org/package/integer-conversion>

<https://jutge.org/doc/haskell-cheat-sheet.pdf>

## Bronnen voor kennis van concepten

<https://en.wikipedia.org/wiki/Run-length_encoding>

<https://en.wikipedia.org/wiki/Immutable_object>

<https://en.wikipedia.org/wiki/Recursion_(computer_science)>

<https://en.wikipedia.org/wiki/Pure_function>

<https://en.wikipedia.org/wiki/Lazy_evaluation>

