Jochem Jansen (1673402)

Datum: 3-10-2025  
Docent: Dennis Breuker  
Klas: ITN-CNI-A-s

Paradigma Opdracht Rapport

APP – Versie 1

Inhoud

[Inleiding 2](#_Toc210397622)

[Onderzoek 3](#_Toc210397623)

[Challenge 4](#_Toc210397624)

[Implementatie 5](#_Toc210397625)

[Reflectie 6](#_Toc210397626)

[Conclusie 7](#_Toc210397627)

[Bibliografie 8](#_Toc210397628)

# Inleiding

Door middel van de paradigma opdracht wil ik achterhalen hoe de werking van functionele programmeertalen in elkaar zit. Ook ga ik functionele concepten koppelen aan specifieke onderdelen van mijn uit geprogrammeerde algoritme. In dit rapport worden vervolgens mijn bevindingen en onderzoek verwoord.

De programmeertaal die ik heb gekozen is Haskell, een puur functionele programmeertaal die bekend staat om zijn sterke typesysteem en nadruk op zuiver functioneel programmeren. (haskell.org, sd) De reden dat ik Haskell had gekozen van de 4 mogelijke programmeertalen is omdat de docent had vermeld dat Haskell wel vaker wordt gebruikt als programmeertaal in het bedrijfsleven. Ik zag in Haskell dus de meeste potentie.

# Onderzoek

Zoals in de inleiding was benoemd is Haskell een **zuiver functionele programmeertaal**. Dit houdt in dat alleen gebruik wordt gemaakt van expressies die een waarde opleveren. Zelfs de IO functie waarmee je een input en output kan geven zijn gemaakt uit zuivere code. (haskell.org, sd)

Een ander concept wat centraal staat bij Haskell is **lazy evaluation**. Expressies worden pas geëvalueerd wanneer het resultaat nodig is. Zo kun je structuren als if/else schrijven door gewoon functies te schrijven. Ook verbetert dit de performance van je applicatie aangezien je niet door onnodige berekeningen hoeft te lopen. (haskell.org, sd)

Elke expressie in Haskell heeft een type die is bepaald tijdens het compileren. Als een type niet overeenkomt met die van de functie wordt het programma afgewezen door de compiler. Ook maakt Haskell gebruik van **type inference** waardoor je niet elk type expliciet hoeft uit te schrijven. Dus wanneer je een variabel aanmaakt hoef je daar niet per se ‘String’ aan te koppelen.

Verder kan je gebruik maken van **higher-order functies** in Haskell en dus functies meegeven als argument aan andere functies of functies retourneren. (Higher order function, 2010)

De variabelen in Haskell zijn standaard **immutable** waardoor je niet variabelen kan aanpassen. Ook support Haskell **pattern matching** op datastructuren. Dit maakt het mogelijk om functies te maken die automatisch inspelen op de structuur van hun invoer. (A brief introduction to Haskell, 2024)

# Challenge

Ik heb ervoor gekozen om een Markdown-to-HTML converter te maken. Dit houdt in dat mijn programma een markdown bestand pakt en deze converteert naar html code.

De uitdaging in deze challenge zit hem in het toepassen van functionele concepten als pattern matching en het nadenken in de termen van een functionele programmeertaal. Markdown bestaat uit verschillende elementen zoals lijsten, koppen en tabellen. Om deze elementen om te zetten naar html moet je door middel van het gebruik van pattern matching de correcte elementen bij de correcte html tags koppelen.

Daarnaast maakt Haskell het lastig om een tekst stap voor stap te manipuleren zoals bij een imperatieve programmeertaal, aangezien in Haskell je focust op wat voor resultaat je wilt hebben en niet op hoe je een resultaat bereikt. Wat bij deze uitdaging een probleem gaat zijn is dus het lopen door meerdere regels tekst, aangezien een markdown bestand vrijwel altijd bestaat uit meer dan één regel tekst.

# Implementatie

In de [code](../Implementatie/converter.hs) staat een diepgaande uitleg over vrijwel alle regels.

Wanneer je de converter runt zal deze vragen om een input en output file. Dit houdt in dat je een markdown-file en een html-file moet meegeven, zodat hij de markdown tekst kan omzetten naar html. Wanneer je de files hebt meegegeven gaat hij door alle lijnen code heen door middel van recursie en zal die zo controleren wat voor markdown elementen erin staan om die te koppelen aan de correcte html-tags.   
De converter werkt momenteel alleen voor de volgende elementen:

|  |  |  |
| --- | --- | --- |
| **Naam** | **Markdown** | **Html** |
| Hoofdstukken (1 t/m 6) | # Hoofdstuk | <h1>Hoofdstuk</h1> |
| Lijsten | * Lijst | <ul><li>Lijst</li></ul> |
| Bold | \*\*Bold\*\* | <strong>Bold</strong> |
| Italic | \_Italic\_ | <em>Italic</em> |
| Tekst | Tekst | <p>Tekst</p> |

Tabel 1: Toegepaste elementen in de converter (Basic Syntax | Markdown Guide, sd)

Wanneer alle tekst is omgezet naar html wordt de body aan een html boilerplate toegevoegd en wordt de gehele html geschreven in het html-bestand die is opgegeven.

## Functionele concepten

**Pattern matching:** Een goed voorbeeld van pattern matching is te zien bij de conversie van de hoofdstukken. Hier zie je dat de functie ‘convertSingleLine’ langs elk hoofdstuk type van markdown gaat om zo te kijken welke overeenkomt met de meegegeven regel tekst.

Afbeelding met tekst, schermopname, Lettertype

Door AI gegenereerde inhoud is mogelijk onjuist.

Afbeelding 1: Heading conversions

**Recursie:** Een voorbeeld van recursie is bij het converteren van dik- en schuingedrukte tekst. Om ervoor te zorgen dat de converter verder gaat nadat er dikgedrukte letters zijn gevonden in een regel tekst, wordt de functie weer opnieuw aangeroepen de rest van de zin kan worden gecontroleerd en omgezet. Op deze manier is het dus mogelijk om meerdere keren in een zin gebruik te maken van dik- of schuingedrukte letters en hoef je niet elke keer een nieuwe regel hiervoor aan te maken.

Afbeelding met tekst, schermopname, Lettertype

Door AI gegenereerde inhoud is mogelijk onjuist.

Afbeelding 2: Emphasis conversion

**Pure functies:** Functies zoals degene aangegeven in afbeelding 2 zijn zuiver. Dezelfde input geeft altijd dezelfde output en er zijn verder geen side effects. De enige functie in mijn applicatie die niet zuiver is, is de main functie aangezien die bestanden moet lezen en schrijven.

**Higher-order functies:** Ik heb gebruik gemaakt van functies als map en span die andere functies als argumenten kunnen nemen. Dit heb ik bijvoorbeeld gebruikt in de ‘convertLines’ functie om met span de regels die items zijn uit een lijst op te splitsen van de rest van de regels.

Afbeelding met tekst, schermopname, Lettertype

Door AI gegenereerde inhoud is mogelijk onjuist.

Afbeelding 3: List conversion

Verder zijn de overige functionele concepten standaard deel van Haskell. In Haskell zijn functies altijd **First-class functies** aangezien je ze kan doorgeven en opslaan in variabelen. In afbeelding 3 zie je bijvoorbeeld dat ik de functie ‘toLi’ doorgeef aan de functie ‘map’. Ook zijn waarden standaard **immutable**, aangezien je altijd een nieuwe variabel aanmaakt in plaats van hem veranderen (zie afbeelding 3). Tenslotte maakt Haskell altijd gebruik van **lazy evaluation**, maar heb ik hier zelf niet bewust mee gewerkt.

# Reflectie

Het lastigste aan deze opdracht was voor mij de switch van object-georiënteerd programmeren naar functioneel programmeren. In eerste instantie wist ik niet eens hoe ik zou moeten beginnen, ook voornamelijk omdat het een nieuwe taal was. Om een beginnetje te maken had ik aan chat gevraagd hoe een conversie precies zou kunnen werken, en hierdoor kwam ik gelijk in aanraking met het eerste functionele concept, pattern matching. (ChatGPT - Markdown naar HTML Haskell, sd)

Hierdoor leerde ik werken met pattern matching en de mogelijkheden die je hiermee had, zo kon je het bijvoorbeeld gebruiken als een gigantische if/else statement die controleert of de pattern van je waarde overeenkomt met degene die je wilt.

In tegenstelling tot klassen en objecten werkt functioneel programmeren met functies en het verwerken van data hiermee. Hierdoor heb ik leren werken met de denkwijze dat je data moest doorgeven en functies daarop moest toepassen, in plaats van dat je een object kan manipuleren.

Het programmeren voelde best straight-to-the-point aangezien het dus doorgeven van variabelen was. Hierdoor vind ik de functies ook een stuk leesbaarder.

Ik vond het wel naar dat je elke keer opnieuw de applicatie moest opstarten om iets te testen. Ook was het lastig om te werken zonder herkenbare constructies als for loops. Het kostte even tijd om een vervanging hiervoor te zoeken, wat in mijn geval uiteindelijk recursie was.

Ik heb dus geleerd dat functioneel programmeren vraagt om een andere mindset en dat door combinatie van pure functies en pattern matching het complexe datastructuren overzichtelijk kan maken.

# Conclusie

In conclusie vraag functioneel programmeren naar een andere manier van denken, maar kan dit voor veel voordelen zorgen zoals overzichtelijke code, het eenvoudiger structuren van complexe algoritmes, het vermijden van onverwachte bijwerkingen door bijvoorbeeld pure functies en betere performance. Ongeacht het feit dat de overstap van object-georiënteerd naar functioneel programmeren uitdagend en verwarrend was, ben ik er bewust van dat functionele programmeertalen als Haskell snelle en leesbare oplossingen kan opleveren.

# Bibliografie

*A brief introduction to Haskell*. (2024, Mei 3). Opgehaald van HaskellWiki: https://wiki.haskell.org/A\_brief\_introduction\_to\_Haskell

*Basic Syntax | Markdown Guide*. (sd). Opgehaald van markdownguide.org: https://www.markdownguide.org/basic-syntax/

*ChatGPT - Markdown naar HTML Haskell*. (sd). Opgehaald van ChatGPT: https://chatgpt.com/share/68df936e-a5f0-8011-9967-875deb4d05f7

haskell.org. (sd). *Haskell Language*. Opgehaald van haskell: https://www.haskell.org/

*Higher order function*. (2010, September 29). Opgehaald van HaskellWiki: https://wiki.haskell.org/Higher\_order\_function