|  |
| --- |
| Algoritmes, paradigma’s en programmeertalen\_  Blackjack |

Khaled El Samman

1652064

3 oktober 2025

Versie 1.0

INHOUDSOPGAVE

[1 Onderzoek 4](#_Toc210397741)

[1.1 Haskell 4](#_Toc210397742)

[1.1.1 Recursie 4](#_Toc210397743)

[1.1.2 Higher Order Functions 4](#_Toc210397744)

[1.1.3 Pattern Matching 4](#_Toc210397745)

[1.1.4 Lazy Evaluation 5](#_Toc210397746)

[1.1.5 First Class Functions 5](#_Toc210397747)

[2 Challenge 5](#_Toc210397748)

[3 Implementatie 6](#_Toc210397749)

[3.1 Basis 6](#_Toc210397750)

[3.2 BlackjackAI 6](#_Toc210397751)

[3.3 BlackjackSim 7](#_Toc210397752)

[4 Reflectie 8](#_Toc210397753)

[4.1 Functionele Concepten 8](#_Toc210397754)

[4.1.1 Immutability 8](#_Toc210397755)

[4.1.2 Recursie 8](#_Toc210397756)

[4.1.3 Higher Order Functions 9](#_Toc210397757)

[4.1.4 Pattern Matching 9](#_Toc210397758)

[4.1.5 Pure Functions 9](#_Toc210397759)

[4.2 Functional vs Imperative 9](#_Toc210397760)

[4.3 Wat nog te doen 10](#_Toc210397761)

[5 conclusie 10](#_Toc210397762)

[6 Bronnen 10](#_Toc210397763)

INLEIDING

Voor deze opdracht moesten we een zelfbedachte challenge maken in een functionele programmeertaal om zo kennis te maken met het functionele paradigma. Ik heb gekozen om een Blackjack kaartenteller te bouwen in Haskell. Met dit programma kan het systeem bijhouden welke kaarten al gespeeld zijn en berekenen of de speler een voordeel heeft.

Het doel van deze opdracht is om functionele concepten zoals pure functies, immutability, recursie en higher order functies in de praktijk te gebruiken. Door het ontwikkelen van dit systeem leer ik hoe functioneel programmeren werkt en wat de voordelen zijn ten opzichte van andere programmeerstijlen.

# Onderzoek

In dit onderzoek beschrijf ik de gekozen programmeertaal en de belangrijkste functionele concepten die in deze taal centraal staan. Ik heb gekozen voor Haskell, omdat het een van de meest bekende functionele programmeertaal is. En dus, kan een grijp in Haskell, mij een bredere toekomstige functie bieden.

## Haskell

Haskell is een functionele programmeertaal die bekend staat om het gebruik van pure. Een aantal concepten staat centraal bij Haskell: recursie, higher order functions en pattern matching.

### Recursie

In Haskell gebruik je recursie in plaats van traditionele loops. Een functie roept zichzelf opnieuw aan totdat een eindconditie is bereikt.

Voorbeeld:

*sumList :: [Int] -> Int*

*sumList [] = 0*

*sumList (x:xs) = x + sumList xs*

Hier wordt **sumList** telkens opnieuw aangeroepen met de rest van de lijst **xs**, totdat de lijst leeg is.

### Higher Order Functions

In Haskell kun je functies doorgeven als argument of teruggeven als resultaat.

Voorbeeld: **map** neemt een functie en een lijst, en past de functie toe op elk element.  
*double :: Int -> Int*

*double x = x \* 2*

*map double [1,2,3]*

*-- resultaat: [2,4,6]*

### Pattern Matching

Met pattern matching kun je data uit elkaar halen op een eenvoudige manier. Dat doe je door verschillende gevallen voor de data te schrijven.

Voorbeeld:

*hiLoCard :: Int -> Int*

*hiLoCard v = case v of*

*2 -> 1*

*3 -> 1*

*10 -> -1*

*\_ -> 0*

Hier worden verschillende kaatwaardes direct herkend door patronen in de input. Voor de kaarten 2 en 3 geeft de functie de waarde 1 terug. Voor de 10 geeft die -1 terug. En in alle andere gevallen geeft die de waarde 0 terug.

### Lazy Evaluation

Haskell voert berekeningen pas uit wanneer het resultaat echt nodig is. Dit kan werk besparen bij grote bomen, omdat niet elke tak meteen wordt doorgerekend.

Voorbeeld:  
*evens :: [Int]*

*evens = [0,2,4….] -- oneindig*

*firstFive :: [Int]*

*firstFive = take 5 evens -- resultaat: [0,2,4,6,8]*

### First Class Functions

In Haskell zijn functies first class: je kunt ze net als waarden doorgeven, opslaan en teruggeven. Dit maakt benodigdheden zoals ‘ pas een bewerking toe op alle elementen’ . Het resultaat is kortere, herbruikbare code, zonder expliciete for loops.

Voorbeeld:

*applyTwice :: (a -> a) -> a -> a*

*applyTwice f x = f (f x)*

*plus1 :: Int -> Int*

*plus1 = (+1)*

*demo1 :: Int*

*demo1 = applyTwice plus1 10 -- resultaat:12*

# Challenge

Voor deze opdracht moest ik een eigen challenge bedenken. In eerste instantie dacht ik aan Rock Paper Scissors Lizard Spock, maar daar heb ik uiteindelijk niet voor gekozen. Dit spel heeft geen ultieme winnende strategie en blijft grotendeels gebaseerd op toeval en geluk, ook al kun je verschillende strategieën implementeren.

Daarna dacht ik aan Tic Tac Toe, maar dat vond ik niet uitdagend genoeg. Er is bovendien veel hulp en kant en klare code online te vinden, waardoor het minder leerzaam zou zijn.

Uiteindelijk heb ik gekozen voor een Blackjack kaartenteller. Dit vond ik een echte uitdaging, omdat ik eerst zelf de basis van kaartentellen moest leren. Daarnaast is er weinig hulp online te vinden voor een directe Blackjack implementatie in Haskell. Dit maakte het een interessant en leerzaam project, waarin ik functionele concepten goed kon toepassen.

# Implementatie

## Basis

De modules **Cards**, **GameFloor** en **IOFloor** heb ik grotendeels gebaseerd op een bestaand tutorial uit het boek Magical Haskell (Antich, 2025). Ze vormen samen de basis van mijn Blackjack spel.

**Cards**: bevat alle kaarten, het deck en de spelers/dealer als data. Hier staan ook functies om kaarten te maken, waarden te berekenen en handen te beoordelen.

**GameFloor**: regelt het spelverloop. Hier zit de logica voor inzetten, kaarten delen, acties van de speler en de dealer, en het bijhouden van de speltoestand.

**IOFloor**: zorgt voor invoer en uitvoer. Dit betekent dat het de gebruiker om input vraagt, de inzet leest en het deck schudt.

Omdat deze modules vooral gaan over data, invoer/uitvoer en het bijhouden van de speltoestand, zitten er weinig echte functionele onderdelen in. Er wordt veel gewerkt met IO acties en het aanpassen van de toestand (state), wat meer op imperative programming lijkt. Ook functies zoals **dealCardsToPlayer** en **dealCardsToDealer** zijn niet puur: ze veranderen het spel in plaats van alleen een nieuwe waarde terug te geven.

Deze drie modules maken het spel speelbaar, maar laten niet goed zien wat functioneel programmeren echt kan. De functionele onderdelen zitten vooral in mijn eigen modules **BlackjackAI** en **BlackjackSim**, waar ik de berekeningen en strategieën heb gemaakt.

**BlackjackAI** en **BlackjackSim** zijn zelf geschreven modules. In deze modules heb ik bewust functioneel programmeren toegepast. Ze vormen het ‘intellegentie’ van het spel: hier wordt het kaartentellen, kansberekenen en strategie bepalen gedaan.

## BlackjackAI

Hierin bereken ik de kaarten­telling met het Hi-Lo systeem. Dit systeem geeft lage kaarten een positieve waarde, hoge kaarten een negatieve waarde en neutrale kaarten de waarde 0. Zo kan het programma inschatten of het gunstig is om een hogere inzet te doen.

Voorbeelden:

* **hiLoCard** bepaalt de waarde van elke kaart met een case expressie (pattern matching).
* **runningCount** telt alle waardes bij elkaar op met **sum . map hiLoCard**.
* **trueCount** berekent de echte telling door **runningCount** te delen door het aantal resterende decks (**remainingDecks**).

Alles in deze module is pure functies: dezelfde input geeft altijd dezelfde output en er wordt nergens state aangepast.

## BlackjackSim

In **BlackjackSim** bereken ik wat de slimste keuze is: hit of stand. Deze module simuleert hoe het spel zou verlopen met alle mogelijke kaarten en kansen.

Voorbeelden:

* **handTotal** berekent altijd het totaal van een hand met **handValue**.
* **dealerResultDist** simuleert alle mogelijke kaarten die de dealer kan trekken door recursief de functie weer aan te roepen met nieuwe deck(s).
* **evHitDepth** berekent de verwachte waarde van meerdere slagen vooruit.
* **decideHitOrStand** vergelijkt de verwachte waarde van hitten en staan en geeft het advies terug.

**evHit**(figuur 1) rekende eerste **evHit pHand dHand deck** zonder limiet alle mogelijke hits door. Elke extra kaart vergroot de boom enorm met een exponentiele groei (O(n!). Dit betekent dat de rekentijd heel snel toeneemt naarmate het boom groter wordt. Door **evHitDepth** **k** te implementeren heb ik deze nesting beperkt. Met een vaste diepte wordt de rekentijd linear (o(n-k)), waardoor het programma veel snel reageert.

Afbeelding met tekst, schermopname, Lettertype

Door AI gegenereerde inhoud is mogelijk onjuist.

Figuur 1 evHit

|  |  |  |  |
| --- | --- | --- | --- |
| **Aantal kaarten (n)** | **Zonder limiet (n!)** | **k=1(n)** | **k=2(n)** |
| 5 | 120 | 5 | 10 |
| 10 | 3628800 | 10 | 20 |
| 20 | 2432902008176640000 | 20 | 40 |
| 52 | 8,07\*1067 | 52 | 104 |

Deze module gebruikt pure functies, recursie, higher-order functies en pattern matching om op een overzichtelijke manier kansen en strategieën te berekenen.

# Reflectie

## Functionele Concepten

### Immutability

In plaats van variabelen aan te passen, worden nieuwe waardes gemaakt.

* In **dealerResultDist** maak ik een nieuwe hand met **(h ++ [c])** in plaats van de bestaande hand te wijzigen.
* **remainingDecks** berekent op basis van het deck een waarde, maar verandert het deck zelf niet.

In mijn code houd ik alles immutabel: niets wordt aangepast. Als de dealer een kaart trekt, maak ik gewoon een nieuwe hand **(h ++ [c])** en reken ik daarmee verder. Het deck zelf pas ik in de pure functies niet aan, een functie zoals **remainingDecks** kan alleen lezen.

### Recursie

Functies roepen zichzelf aan om een probleem in kleinere stappen op te lossen. Dit vervangt loops.

* **dealerResultDist** roept zichzelf aan totdat de dealer 17 of meer heeft of bust gaat.
* **evHitDepth** berekent de verwachte waarde van hitten door telkens een kaart toe te voegen en opnieuw aan te roepen.

Voor het doorrekenen van het spel gebruik ik recursie, omdat dat perfect past bij de regels, trek tot 17+ of bust. In **dealerResultDist** roep ik de functie steeds opnieuw aan met de nieuwe hand, tot we bij een eindtoestand zijn (bust of >=17). Voor de speler gebruik ik **evHitDepth** **k**: met **k** bepaal ik hoe ver ik vooruitkijk**. k=0** betekent: nu stoppen en beoordelen; hogere **k** kijkt verder vooruit maar kost meer rekentijd.

### Higher Order Functions

Functies die andere functies als parameter nemen of teruggeven.

* **runningCount = sum . map hiLoCard** -> **map** past **hiLoCard** toe op elke kaart
* In **dealerResultDist** gebruik ik **map** en **foldr** om kansen samen te voegen

Met higher order functions schrijf ik de kans en tel logica**. runningCount = sum . map hiLoCard** zegt letterlijk: “geef elke kaart een waarde en tel op”. In het combineren van uitkomsten gebruik ik een **foldr** in **combine** om gelijke resultaten samen te voegen en hun kansen op te tellen.

### Pattern Matching

Met pattern matching splits ik de logica netjes op in gevallen, waardoor de code overzichtelijker wordt.

* **hiLoCard** gebruik **case v of** om elk kaarttype een waarde te geven.
* **DealerResult** heeft twee vormen: **DealerBust** en **DealerTotal int**. Deze woorden in **combine** en **dealerResultDist** met pattern matching afgehandeld.

Dit zorgt voor geen lange if-loops, maar per regel precies wat er in dat geval moet gebeuren.

### Pure Functions

Functies zoals **hiLoCard**, **runningCount**, **dealerResultDist**, **evStand** zijn pure: dat wil zeggen dat de dezelfde input altijd dezelfde output geeft. Daardoor kan ik bijvoorbeeld **evHit** en **evStand** vergelijken.

## Functional vs Imperative

Functioneel programmeren werkt hier goed omdat het voorspelbaar is. Functies doen altijd hetzelfde bij dezelfde input (geen bijwerkingen), dus je kunt keuzes eerlijk vergelijken. De spelregels lijken op een beslisboom, daarom is recursie logisch: je rekent stap voor stap door tot een eindpunt (17+ of bust). Voor de snelheid kijk je niet oneindig ver vooruit, maar zet je een grens. Met higher order functies (zoals ‘ doe dit voor elk element’ en ‘ tel alles op’ ) blijft de code kort, en pattern matching zet elk geval (bust, stop,) duidelijk op een eigen regel. Vergeleken met imperatief programmeren voorkom je zo gedoe met veranderende variabelen en door elkaar lopende logica. Als ik blackjack imperatief had moeten doen, zou ik in plaats van recursie, while/for loops gebruiken. Mutabel variabelen voor hand, deck, en kansen. De dealer uitkomst zal ik waarschijnlijk uitrekenen met een eigen stack. Dit zou voor meer boilerplate en vermenging van logica zorgen dan mijn functionele aanpak.

## Wat nog te doen

* Spelregels realistischer maken: split en double down toevoegen, en echte uitbetalingen gebruiken (bijv. blackjack 3:2).
* Sneller vooruitrekenen: diepte adaptief maken (dieper bij twijfel, minder diep bij duidelijke situaties) eventueel door tussenuitkomsten te onthouden (memoization).

# conclusie

Functioneel programmeren past hier goed: je denkt in stappen door een beslisstructuur, zonder bijwerkingen of stiekeme wijzigingen. Daardoor blijven uitkomsten voorspelbaar en kun je keuzes goed vergelijken. De kern is duidelijk: tellen, kansen inschatten en kiezen op basis van verwachte waarde.

Belangrijkste leerpunten uit dit project:

* Zuiverheid geeft rust: dezelfde situatie levert steeds hetzelfde resultaat op, wat vergelijken en testen eenvoudig maakt.
* Immutability geeft rust: omdat waarden niet onverwacht veranderen, hoef je minder te houden en wordt het gedrag van de code voorspelbaarder.
* Recursief past goed bij een beslisboom, maar je moet wel grenzen aan de diepte stellen voor de snelheid.

# Bronnen

<https://learnyouahaskell.com/chapters>

<https://www.haskell.org/>

<https://wiki.haskell.org/A_brief_introduction_to_Haskell>

<https://stackoverflow.com/questions/65241264/haskell-recursion-pitfall-whats-is-slowing-it-down>

https://wiki.haskell.org/tail\_recursion?utm\_source=chatgpt.com

<https://stackoverflow.com/questions/2225774/haskell-pattern-matching-what-is-it>

<https://wiki.haskell.org/index.php?title=Lazy_pattern_match>

<https://www.blackjackapprenticeship.com/how-to-count-cards/>

<https://hackage.haskell.org/package/base-4.21.0.0/docs/Data-Foldable.html>

<https://wizardofodds.com/games/blackjack/appendix/9/1ds17r4/>

https://wiki.haskell.org/index.php?title=Memoization

Antich, A. (2025). Magical Haskell: A Friendly Approach to Moden Functional Programming, Type Theory, and Artifical Intelligence.

