Functioneel programmeren 2

NHLStenden hogeschool

Jan Julius de Lang

11 januari 2020

Inhoud

[Inleiding 3](#_Toc29665981)

[De keuzes 4](#_Toc29665982)

[Regels 5](#_Toc29665983)

[Programmeren 6](#_Toc29665984)

[Werking algoritme 6](#_Toc29665985)

[Kennis 6](#_Toc29665986)

[Opzet 6](#_Toc29665987)

[Gebruikte onderdelen 6](#_Toc29665988)

[Programma volgorde en algemene doorloop 7](#_Toc29665989)

[Resultaten 8](#_Toc29665990)

[Snelheid 8](#_Toc29665991)

[Filesize 8](#_Toc29665992)

[Conclusie 9](#_Toc29665993)

# Inleiding

Voor het vak functioneel programmeren 2 moet er een algoritme in drie verschillende programmeertalen geschreven worden.  
Een imperatieve taal, een semi-imperatieve taal en een functionele taal waarbij de functionele taal Haskell is.  
Wanneer dit gedaan is zal de werken en de uitwerkingen van de algoritmes in de verschillende talen vergeleken worden, hier wordt vervolgens verslaglegging van gedaan in dit document.

# De keuzes

In de tijd die ik programmeer heb ik eigenlijk alleen nog maar imperatieve talen aangeraakt, functionele en zelfs semi functionele talen heb ik buiten de opleiding vrijwel nooit gebruikt.

De keuze voor de imperatieve taal was makkelijk, een taal die ik graag gebruik is C# dus die keuze stond al vrij snel vast.  
Daarnaast wist ik wel van een semi functionele taal af die van Java afstamt, een taal die ik naast C# ook veel heb gebruikt, dat is Kotlin.  
Daarom heb ik Kotlin gekozen voor de semi-functionele taal.  
De functionele taal moest Haskell zijn, daarom heb ik Haskell gekozen.

Voor het algoritme heb ik Breadth-first-search gekozen.   
Dit is een algoritme wat ik al meerdere keren heb gemaakt en gebruikt en iets wat ik leuk vind om te maken.

|  |  |
| --- | --- |
| Imperatieve taal | Csharp |
| Semi-functionele taal | Kotlin |
| Functionele taal | Haskell |
| Algoritme | Breadth-first-search (BFS) |

# Regels

Ik heb een aantal regels opgezet zodat het resultaat (vooral voor de snelheid van de programma’s) zo accuraat mogelijk is.

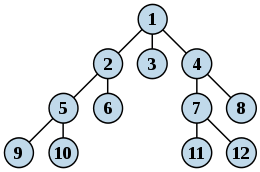
1. Er moet aan het begin van het programma gezegd worden dat het de bfs algoritme is, in die specifieke taal.
2. De nodes (of edges) moeten van tevoren gedefinieerd worden indien mogelijk, zo niet moet dat tijdens runtime gebeuren.
3. De start node moet vermeld worden
4. Het resultaat moet een lijstje zijn van alle bezochte nodes.

En tijdens het programmeren:

1. Maak objecten indien mogelijk.
2. Maak gebruik van componenten van de taal (bijvoorbeeld lists, tuples enz.).

# Programmeren

In alle drie talen is het algoritme geschreven, alles is gebaseerd op dezelfde boom.



De boom.

## Werking algoritme

Het algoritme werkt als volgt en wordt op dezelfde manier geprogrammeerd in de 3 verschillende talen.

* Defineer de nodes/vertexen en de aanliggende.
* Laat weten wat het programma doet bijvoorbeeld: BFS algorithm in C#\n
* Laat weten wat de begin node is, bijvoorbeeld: "Starting node is {src}"
* Gooi start node op een queue
* Terwijl de queue niet leeg is:
  + Haal Node van queue
  + Voeg deze toe aan een bezochte node lijst
  + Kijk of de node buren heeft
  + Zo ja, kijk wie deze buren zijn
  + Voor elke aanliggende node:
    - Kijk of deze al voorkomt in de lijst van bezochte nodes
      * Zo nee voeg deze toe aan de queue

Wanneer de queue leeg is zal de bezochte node lijst teruggegeven worden.

## Kennis

Mijn kennis van imperatieve talen is veel groter dan die van functionele of semi-functionele talen, hierdoor is mijn Kotlin applicatie meer aan de imperatieve zijde dan de functionele zijde.  
Ook kan de Haskell applicatie minder efficiënt zijn dan optimaal beschouwt zal worden aangezien mijn kennis daar niet ligt.

# Opzet

De drie verschillende talen roepen voor verschillende implementaties van het algoritme aangezien niet elke taal bepaalde functionaliteiten heeft.

In Csharp en Kotlin heb ik gebruik gemaakt van OOP, ik heb van de knopen een object gemaakt.  
Haskell heeft een vergelijkbaar systeem, hier kun je gebruik maken van het “data” keyword.

In alle talen heeft een knoop dezelfde informatie, zijn eigen nummer en de aanliggende nummers.

### Gebruikte onderdelen

Van verschillende talen kon ik verschillende al in gebouwde onderdelen gebruiken.

Ik heb het hier niet over datatypen zoals integers en arrays maar over dingen als queues en list, omdat deze onderdelen goede utiliteiten bieden voor het gekozen algoritme.

|  |  |
| --- | --- |
| Taal | Gebruikte onderdelen |
| Csharp | Collections  Collections.Generic  Enumerable  Linq  Diagnostics |
| Kotlin | Collections  \_Collections  \_Arrays  ArrayDeque |
| Haskell | Data.time |

Hieruit blijkt dat, wanneer je met Haskell bezig bent veel van de onderdelen die de andere talen bieden niet aanwezig zijn.  
Data.time in de Haskell applicatie is ook alleen gebruikt om de informatie van de duur van het programma te verkrijgen, dus dit heeft eigenlijk niets met het algoritme te maken.  
Waar bij Csharp en Kotlin de “Queue” en “ArrayDeque” gebruikt wordt, word dat in Haskell allemaal gedaan met recursieve methoden.

Dit heeft ook zijn nadelen voor de programmer, het uitprinten van de data objecten is een stuk gecompliceerder dan in de twee andere talen, voor Haskell moet je daar apart een functie voor schrijven terwijl je dit in de andere talen met ingebouwde functionaliteiten kan doen.

Bijvoorbeeld, in Csharp:

Console.WriteLine("Result: " + string.Join(", ", visisted));

En in Kotlin:

*println*(**"result: "** + result.*joinToString*(postfix = **","**))

Terwijl je daar in Haskell deze functie voor moet schrijven:

printGraph :: **Graph** -> **IO** ()

printGraph (Graph []) = putStrLn ""

printGraph (Graph (x:y)) = do

print x

printGraph (Graph y)

return ()

## Programma volgorde en algemene doorloop

De volgorde van het programma is heel belangrijk in een imperatieve taal, dat is bij een functionele taal niet zo belangrijk, ook gebruik je in een functionele veel meer functies dan in een imperatieve taal, eigenlijk hangt het gehele functionele programma samen met functies bij een imperatieve taal hangt dat meer samen met losse methoden die condities en loops bevatten.

# Resultaten

Snelheid:

|  |  |  |  |
| --- | --- | --- | --- |
| Taal | Tijd (in ms) | Project | Gekompileerd |
| Csharp |  | 20 | 8-11 |
| Kotlin |  | 24 - 27 | 15-19 |
| Haskell |  | - | 3.5-4 |

De resultaten van de snelheden zijn niet heel bijzonder, wel vond ik het interessant dat Haskell sneller is in vergelijking met de andere twee talen.  
Het blijkt dat Haskell zo snel is door dat er statische types zijn, de taal “lazy” is en omdat de compiler radicale veranderingen kan implementeren zonder de verwachtingen van de code te breken.

Filesize:

|  |  |  |
| --- | --- | --- |
| Taal | File size | Compiled size |
| Csharp | 3.03Kb | 7.5kb (.EXE) |
| Kotlin | 1.97kb | 3,527kb (.JAR) |
| Haskell | 2.30kb | 15,587kb (.EXE) |

Haskell’s filesize is een heel stuk groter dan de rest, dit komt omdat haskell alles statisch gelinkt heeft en dus ook de taal zijn runtime en standaard libraries meegeeft aan de executables, vandaar dat de binaries van Haskell een stuk(je) groter zijn dan de andere talen. Ook is een .jar gecomprimeerd en daarom iets kleiner.

# Conclusie

Alle talen hebben verschillende oplossingen voor het algoritme, elke taal doet dit op zijn eigen manier en de programmeur is heel vrij in het zelf bedenken van een oplossing in welke taal dan ook.  
Functionele programmeertalen zijn meer geschikt voor het maken van kleinere programma’s waar je je niet hoeft te houden aan het bewaren van objecten of het onthouden van de status van deze.  
Een imperatieve taal kan goed gebruikt worden als meerdere onderdelen dezelfde functionaliteit moeten hebben en deze tegelijk of los van elkaar moeten worden gemanipuleerd.

De conclusie in de verschillen van de talen die ik kan trekken na het maken van de algoritmes en het testen daarvan.

|  |  |
| --- | --- |
| Imperatief | Functioneel |
| For, foreach en while | Recursie of nesting |
| Methoden geven terug op basis van condities | Functies muteren |
| Aanpassing datastructuren bij het benaderen daarvan | Aanpassing datastructuren doormiddel van recursie en functies |
| Statische uitvoering (vaste volgorde) | Dynamische uitvoering (geen vaste volgorde) |
| Bij grote projecten het aanpassen van veel objecten | Doorbouwen op eigen gemaakte functies |

Beide soorten talen hebben verschillende voor- en nadele, dit ligt vooral aan de voorwaarden van de opdracht.