

January 30, 2026

Contents

0.1	Module Functioneel Programmeren	1
0.2	Inleiding	2
0.3	Overzicht	3
0.3.1	Eerste kennismaking	3
0.3.2	Waarden, expressies en namen	3
0.3.3	Functies	3
0.3.4	Types en waarden	3
0.3.5	Functies als waarden	4
0.3.6	Zelf-gedefinieerde data-types	4
0.3.7	Recursie: data-types en functies	4
0.3.8	Generieke functies (map enz.)	4
0.4	Voor de docent	5
0.4.1	Tips voor het gebruik van Jupyter Notebook	5
0.5	Over deze module	6
0.6	Waarden en functies	7
0.6.1	Waarden, namen en expressies	7
0.6.2	Namen: definitie en gebruik	7
0.6.3	Functies: definitie en gebruik	8

0.1 Module Functioneel Programmeren

0.2 Inleiding

- het doel van de module is kennismaking met Functioneel Programmeren, als programmeerparadigma
- we gebruiken hierbij Haskell als voorbeeld-taal;
 - deze module is geen inleiding in Haskell, veel aspecten van deze taal laten we buiten beschouwing
-

0.3 Overzicht

0.3.1 Eerste kennismaking

0.3.2 Waarden, expressies en namen

concepten: waarde, expressie, operator, operator-prioriteit (precedentie?), naam-definitie, naamgebruik, rekenen/herschrijven (berekening)

- een expressie *beschrijft* een berekening. Wanneer deze berekening uitgevoerd wordt, is het resultaat van deze berekening een *waarde* (getal, string, enz.)
- een expressie bestaat uit waarden (voorlopig alleen gehele getallen), operatoren (en functies? bijv. div, mod?, abs, neg?, succ, pred)
- functie-toepassing (applicatie, aanroep) heeft in Haskell de vorm: `f 3` - de naam van de functie met daarachter het argument.

- in plaats van `f(3)`

voorbeelden: enkele uitgewerkte berekeningen

opdrachten: uitwerken van enkele berekeningen

0.3.3 Functies

concepten: functie-definitie, functie-naam, parameters, functie-toepassing (applicatie, aanroep), argumenten

- een functie kun je zien als de *abstractie* van een berekening: de interface beschrijft het effect(?), de expressie (body) de implementatie: de berekening voor het bereiken van dat effect.
 - in het geval van functies kun je het effect beschrijven als een relatie tussen de parameters en het resultaat.
 - een voorbeeld is `ggd x y` - het *effect* is: het grootste getal dat zowel een deler is van `x` als van `y`; dit zegt nog niets over de manier waarop je dit kunt berekenen. (Dit is overigens een klassiek informatica-voorbeeld.)
 - een ander voorbeeld: `sort xs` - het *effect* is:
- een functie kun je (ook) zien als een *uitgestelde berekening*, vanwege het ontbreken van de waarden van de parameters. Zodra deze bekend zijn, bij functie-applicatie, kun je de berekening uitvoeren.

voorbeelden: `sqr` (definitie en toepassing), `succ` (en functies met twee parameters?)

opdrachten: uitwerken van enkele berekeningen; def. van double

0.3.4 Types en waarden

concepten:

voorbeelden:

Elementaire types

concepten: typering van waarden; typering van functies

Getallen, tekens, boolean

voorbeelden:

Samengestelde types

Strings, lijsten; tupels

Typering van functies

(In het bijzonder ook: functies van meerdere parameters.)

0.3.5 Functies als waarden

- anonieme functie-waarden: lambda-expressies
 - (overeenkomst tussen lambda-expressie en regel voor functie-applicatie: `let`)
- map
- foldl, foldr, (reduce)
- filter
- zip (tupels en lijsten)
- functies met meerdere parameters; partiële parametrisatie (Currying)

(De uitdaging van het gebruik van map, foldl, filter enz. is om te denken in complete lijsten, niet in de afzonderlijke elementen.)

(Bij dit hoofdstuk moeten we een redelijk groot aantal oefeningen hebben, om leerlingen vertrouwd te maken met de verschillende begrippen en de manier waarop je die gebruikt. Een aantal kleine voorbeelden moet met de hand uitgewerkt worden, voor een beter begrip.)

0.3.6 Zelf-gedefinieerde data-types

- constructors (met parameters)
- functie-definities met *pattern matching* (eigenlijk: “de-constructie”)
- voorbeelden:
 - grafische (geometrische vormen) - cirkel, rechthoek, driehoek, trapezium, ster?
 - * bepalen van oppervlakte
 - *
 - expressie - operator met operanden, (bijv. `+`, `-`, `*`, `. mod`, `div`); functie met argumenten? (De voorbeelden in dit hoofdstuk bereiden voor op de “logische” uitbreiding naar groepering, en daarmee naar recursie.)

(Eigenlijk zou ik de vormen zo willen definiëren dat je deze ook kunt tekenen; dan moeten we ook met coördinaten werken, bijvoorbeeld in de vorm van tupels.)

0.3.7 Recursie: data-types en functies

- recursieve data-types
 - grafische vormen - met groepering
 - expressies - met sub-expressies
- recursieve functies voor recursieve data
 - grafische vormen, bijv. berekenen van oppervlak (of: omzetten in SVG)
 - expressies, bijv. evaluatie
- binaire bomen
- generieke bomen (met willekeurig aantal kinderen)
- bomen in de informatica
 - recursieve structuren (o.a file system; en eerdere genoemde voorbeelden)
 - zoekbomen
- lijsten - en functies op lijsten (sum, max, `lst_sqr`)

0.3.8 Generieke functies (map enz.)

- definities van map, foldl (en foldr)
- (en nog enkele andere functies)

0.4 Voor de docent

- **PRIMM** Als didactische aanpak gebruiken we, waar mogelijk, de PRIMM methode: Predict, Read, Investigate, Modify, Make. Leerlingen beginnen met het lezen en begrijpen van bestaande code.
 - het is dan belangrijk om goede voorbeelden te kiezen.
- **Gebruik voor implementatie.** In het algemeen proberen we voorbeelden van het gebruik te laten zien, voordat we ingaan op de implementatie.
- – **Uitwerken met de hand.** Om een goed begrip te krijgen van de verschillende constructies, is het belangrijk om een aantal voorbeelden met de hand uit te werken.
- **Kleine stappen, controleren cq. testen.** We gebruiken Jupyter Notebook (of -Lab) om programma's op te kunnen bouwen in kleine stappen, waarbij elke stap getest kan worden met een of twee kleine voorbeelden.
 - Welke voorbeelden kies je om te testen? Zorg in elk geval dat je de randgevallen apart test, zoals bijvoorbeeld de lege lijst. En daarnaast een of twee "normale" gevallen.

0.4.1 Tips voor het gebruik van Jupyter Notebook

- gebruik Jupyter Notebook om te experimenteren met de voorbeelden: pas de voorbeeld-data aan, en pas de voorbeeld-functies aan.
- de volgorde van de berekening is belangrijk, hiermee bouw je een historie op: na een aantal experimenten kan het zijn dat de actuele inhoud van de cellen niet meer klopt met de opgebouwde berekening-historie (van definities e.d.). Het is handig om regelmatig de kernel opnieuw te starten (bijv. *Restart Kernel and Clear...* of *Restart Kernel and Run up to Selected Cell*).
-

0.5 Over deze module

De aanpak van het thema *functioneel programmeren* is wat anders dan gebruikelijk, met als bedoeling de essentiële begrippen op een logische en geleidelijke manier in te voeren.

Eén van de uitgangspunten is “gebruik voor definitie”: voor het gebruik heb je voldoende aan een specificatie, die soms wat informeel kan zijn. Bij de implementatie komen dan alle formele details aan bod, maar op dat moment is de noodzaak van die details vaak wel duidelijk.

Functioneel programmeren gaat voor een belangrijk deel over *abstractie*. Maar het is belangrijk om zo concreet mogelijk te beginnen, en de abstractie daarna in te voeren als een volgende, logische stap.

Functies. Het eerste onderdeel betreft het functiebegrip: een functie als *uitgestelde berekening*. Er zijn twee redenen om die berekening uit te stellen:

- (a) je kent de invoer (argumenten) van de functie nog niet;
- (b) je hebt het resultaat van de functie nog niet nodig.

Ad (a): als je een functie toepast op één of meerdere argument-waarden, kun je de berekening van de functie (uit de functie-definitie) uitvoeren.

Ad (b): je hoeft de functie niet uit te voeren, ook als de invoer (argumenten) wel beschikbaar zijn in een functie-toepassing; dit kun je uitstellen totdat het resultaat van deze functie-toepassing nodig is. Dit *lazy evaluation* principe wordt in sommige functionele programmeertalen, zoals ook Haskell, gebruikt. In onze voorbeelden zullen we echter vrijwel altijd een *stricte* evaluatie gebruiken: we rekenen een functie-toepassing uit zodra de argumenten beschikbaar zijn.

Het belangrijkste van dit hoofdstuk is het onderscheid tussen een *functie*, als rekenvoorschrift of uitgestelde berekening, en de toepassing (of “aanroep”) van die functie, waarbij die berekening uitgevoerd wordt. (*wanneer de omvattende berekening uitgevoerd wordt*) (Ook: *denken in termen van het effect van de functie, in plaats van de implementatie*).

We introduceren een eenvoudig rekenmodel, waarbij een expressie via het invullen van waarden stapsgewijs uitgerekend wordt tot een enkele waarde.

Ook: functie als *abstractie*, kunnen werken met een functie op basis van de *specificatie* van die functie, zonder kennis van de definitie (implementatie).

Belangrijke begrippen hierbij zijn: waarden, namen (gekoppeld aan waarden), functie-specificatie (interface): parameters, resultaat; functie-toepassing (“aanroep”), op bepaalde argument-waarden; functie-definitie. (functie als afbeelding?)

Hogere-orde functies.

Belangrijke begrippen: lambda-expressie (anonieme functies). Functie als parameter. Functie als resultaat. `map`, `fold` (of `reduce`), `filter`; lijst-waarden; partiële evaluatie, Currying.

Zelf-gedefinieerde data-types.

Structuur het een functie volgt de structuur van het data-type (van het argument).

Recursieve data-types en recursieve functies.

Structuur van de (recursieve) functie volgt de structuur van het data-type.

Een recursieve definitie, van een data-type of van een functie, heeft naast de recursieve alternatieven tenminste één niet-recursief alternatief, om de “recursie te eindigen”.

Lijst als recursief data-type.

Haskell-lijst als recursief data-type; geparametriseerde (generieke) data-types. definitie (implementatie) van (generieke) functies als `map`, `fold` en `filter`.

Bomen

Binaire bomen, zoekbomen, afdrukken van bomen; flatten, opbouwen van bomen; (B-trees?)

0.6 Waarden en functies

(Inleiding)

0.6.1 Waarden, namen en expressies

Begrippen

(enkelvoudige) waarde, operator, expressie, prioriteit (of “sterkte van binding”), naam (-definitie, -gebruik)

Als eerste stap in de kennismaking met functioneel programmeren beginnen we erg eenvoudig: met *waarden*, zoals gehele getallen, en *expressies* waarin je (invoer)waarden gebruikt om nieuwe (uitvoer)waarden te berekenen.

Een **enkelvoudige waarde** is een *getal* of een *teken (character)*. Voorlopig gebruiken we in de voorbeelden alleen *gehele getallen*; maar zwevende komma getallen (*floating point*) zijn ook mogelijk in Haskell.

Later zullen we kennismaken met *samengestelde waarden*, zoals strings en lijsten.

In **expressies** kunnen we waarden combineren met *operatoren* en *functies*, om daarmee nieuwe waarden uit te rekenen. De gebruikelijke operatoren zijn: + - * voor optellen, aftrekken en vermenigvuldigen. Voor het delen van gehele getallen hebben we de functies `div` en `mod`. Vermenigvuldigen en delen hebben een grotere prioriteit dan optellen en aftrekken, alweer zoals we gewend zijn. We kunnen haakjes gebruiken om de prioriteit (volgorde van berekenen) expliciet aan te geven.

(Operatoren blijken in Haskell ook gewoon functies te zijn; daar komen we later op terug.)

Voorbeelden van expressies

`3 + 4 * 5`

`23`

`(3 + 4) * 5`

`35`

Opdracht. Geef voor bovenstaande expressies aan wat de waarden zijn en wat de operatoren.

0.6.2 Namens: definitie en gebruik

We kunnen een waarde een **naam** geven (*definiëren*), en die naam later *gebruiken* waar we de waarde nodig hebben. Bij het uitrekenen van een expressie vervangen we een naam dan door de waarde die aan die naam gekoppeld is.

In een functionele taal als Haskell mag een naam precies één keer gedefinieerd zijn: je kunt in eenzelfde programma die naam geen andere waarde geven. `a` is dus geen variabele in de zin van “gewone” programmeertalen, het is meer zoals een naam in de wiskunde die een enkele waarde voorstelt.

let op: namen van waarden en van functies moeten in Haskell met een kleine letter beginnen.

`a = 3` -- definitie van `a`

`a + 4 * 5` -- gebruik van `a`

`23`

`b = 4 * 5` -- definitie van `b`

`a + b` -- gebruik van `a` en `b`

23

Opmerking. Wanneer reken je de expressie in de definitie van **b** uit? In onze voorbeelden gaan we ervan uit dat je eerst de expressie uitrekent, en de resultaat-waarde gebruikt als definitie van **b**. Maar: in Haskell maakt dat niet uit, omdat je een naam maar éénmaal een waarde mag geven: je zou de berekening ook later kunnen uitvoeren (*lazy evaluation*).

Een expressie beschrijft een berekening. Deze berekening kun je met de hand uitvoeren, waarbij je elke stap uitlegt wat je doet.

Voorbeeld:

```
2 * a + b
==      {vervang a door de bijbehorende waarde: 3}
2 * 3 + b
==      {rekenen}
6 + b
==      {vervang b door 20}
6 + 20
==      {rekenen}
26
```

Dit blijkt te kloppen met het Haskell-resultaat

```
2 * a + b
26
```

Bovenstaande uitwerking lijkt erg omslachtig, maar we zullen later voorbeelden zien waarin zulke berekeningen kunnen helpen om programma's te begrijpen.

0.6.3 Functies: definitie en gebruik

In de voorbeelden hierboven staan de namen **a** en **b** voor enkelvoudige waarden (gehele getallen). We kunnen een naam ook koppelen aan een *functie*.

- de definitie van een functie koppelt de functie-naam aan een functie-waarde
- het gebruik van een functie(-naam) met een argument-waarde betekent de toepassing van de functie op die waarde.

functie-toepassing. In een expressie kunnen we ook functies gebruiken: we spreken dan functietoepassing (functie-applicatie) of functie-aanroep. De functie **abs** *toegepast op het argument 5* schrijf je in Haskell als: **abs 5** - de naam van de functie met daarna het argument.

In de wiskunde en in veel programmeertalen zou je schrijven: **abs(5)**, maar in Haskell gebruik je voor functie-toepassing geen haakjes.

Je hebt soms wel haakjes nodig om de rekenvolgorde te beschrijven (prioriteit of binding): functietoepassing “bindt sterker” dan operatoren zoals **+** en *****. **abs a + b** betekent dan: **(abs a) + b**, als je het anders bedoelt moet je dit schrijven als: **abs (a+b)**,

```
abs (a - 10)
```

7

```
abs a - 10
```

-7

functie-definitie. Als eerste voorbeeld van een functie definiëren we de functie `sqr`, voor het kwadrateren van een getal. De gebruikerlijke wiskundige definitie is:

$$f(x) = x * x \quad (1)$$

In Haskell wordt dit:

```
sqr x = x * x
```

Deze regel definieert de naam `sqr` als de naam van de functie, met één *parameter*, die we hier `x` noemen. De definitie koppelt de naam `sqr` aan de geparametriseerde expressie, `x * x`.

De *naam* van de parameter `x` heeft alleen betekenis in de functie-definitie. We kunnen deze definitie ook schrijven als: `sqr y = y * y` of als `sqr abc = abc * abc`.

Merk op dat we hier, net als bij functie-toepassing, geen haakjes schrijven.

De toepassing van deze functie op een argument, bijvoorbeeld `sqr 7`, resulteert in de expressie (berekening) waarin de waarde van het argument ingevuld is in de definiërende expressie van de functie, hier `sqr`:

```
sqr 7
== {functie -toepassing sqr: vervang de naam door de definiërende expressie}
let x = 7 in x * x
== {vervang x door waarde: 7}
7 * 7
== {rekenen}
49
```

Met de `let`-constructie in Haskell kun je een lokale naam definiëren, die buiten deze `let` geen betekenis heeft. Op die manier voorkomen je naamsconflicten.

```
let z = 12 in z + z
```

24

De berekening in Haskell geeft hetzelfde resultaat als hierboven:

```
sqr 7
```

49

Begrippen

- functie-definitie (vs. functie-toepassing of -gebruik)
- functienaam
- (formele) parameter - als lokale naam
- (geen haakjes)
- scope van een lokale naam; “uniek” ten opzichte van andere namen in programma
- “body” - of expressie? - waarin deze parameter voorkomt, “geparametriseerd”

We kunnen deze functie *toepassen* op het *argument* 3. Dit is een voorbeeld van functie-applicatie ofwel functie-toepassing.

```
sqr 3
```

Mogelijke voorbeelden:

- double
- succ
- isEven
- (omzetten van hoofd- in kleine letters of omgekeerd)

Voorbeeld. In het onderstaande voorbeeld gebruiken we een *expressie* als argument voor de `sqr`-functie. In dit geval moeten we haakjes gebruiken voor deze expressie, omdat functie-applicatie sterker “bindt” dan optelling (of andere operatoren). De notatie `sqr a + 2` moet je dan lezen als `(sqr a) + 2`. *Ga dit na.*

```
sqr (a + 2)
```

25

We rekenen deze expressie uit door *eerst de argument-expressie `a + 2` uit te rekenen*:

```
sqr (a + 2)
==      {invullen waarde van a}
      sqr (3 + 2)
==      {rekenen}
      sqr 5
==      {invullen definitie van sqr}
      let x = 5 in x * x
==      {invullen waarde van x}
      5 * 5
==      {rekenen}
      25
```

We kunnen de volgorde van de verschillende stappen ook anders kiezen, bijvoorbeeld:

```
sqr (a + 2)
== -- invullen definitie van sqr
      let x = a + 2 in x * x
== -- invullen van waarde van x
      (a + 2) * (a + 2)
== -- invullen van waarde van a
      (3 + 2) * (3 + 2)
== -- rekenen
      5 * 5
== -- rekeken
      25
```

Merk op dat we hetzelfde resultaat krijgen als eerder. In een functionele taal maakt de volgorde van de stappen geen verschil voor het resultaat, omdat de waarde bij een naam nooit verandert. Maar deze volgorde kan wel verschil maken voor de *hoeveelheid rekenwerk*. In dit laatste voorbeeld moesten we bijvoorbeeld de uitdrukking `3 + 2` tweemaal uitrekenen.

In het vervolg hanteren we steeds de regel: eerst de argumenten uitreken, daarna de functie-definitie invullen. Deze manier van het gebruik van functies heet *strict evaluation*. Dit is de aanpak die de meeste programmeertalen gebruiken, ook veel functionele talen.

Haskell zelf gebruikt een andere aanpak: een expressie wordt pas uitgerekend als het resultaat nodig is, bijvoorbeeld omdat dit afgedrukt moet worden. Deze aanpak heet *lazy evaluation*. We zullen daar later voorbeelden van laten zien, en gevallen waar dit extra gemak geeft.

Functies met meerdere parameters

```
add x y = x + y
```

```
add 3 (4*5)
```

23

Merk op: geen haakjes, geen komma. Maar wel haakjes voor de expressie $4*5$, omdat *functieapplicatie sterker bindt dan vermenigvuldigen*. Zonder deze haakjes zou de betekenis zijn: $(\text{add } 3 \ 4) * 5$ (*controleer dit*)