January 30, 2026

Contents

0.1 Module Functioneel Programmeren

0.2 Inleiding

- het doel van de module is kennismaking met Functioneel Programmeren, als programmeer-paradigma
- 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?)