February 3, 2026

Contents

0.1 Module Functioneel Programmeren . . . 1
0.2 Inleiding . . . 2
0.3 Overzicht . . . 3
0.3.1 Waarden, expressies en namen . . . 3
0.3.2 Functies . . . 3
0.3.3 Types en waarden . . . 4
0.3.4 Functies als waarden . . . 4
0.3.5 Zelf-gedefinieerde data-types . . . 4
0.3.6 Recursie: data-types en functies . . . 4
0.3.7 Generieke functies (map enz.) . . . 5
0.4 Waarden en functies . . . 6
0.4.1 Waarden, namen en expressies . . . 6
0.4.2 Namen: definitie en gebruik . . . 6
0.4.3 Functies: definitie en toepassing . . . 7
0.5 Een voorproefje van functioneel programmeren in Haskell . . . 11
0.6 Typering en lijsten . . . 12
0.6.1 Typering van functies . . . 14
0.7 Functies als waarden . . . 15
0.7.1 Lambda-expressies: anonieme functie-waarden . . . 15
0.7.2 Functie als operator, operator als functie . . . 16
0.7.3 Functie als parameter: map, reduce, filter . . . 16
0.7.4 Functie als resultaat . . . 18
0.7.5 Functies in datastructuren . . . 19
0.8 Zelf-gedefinieerde datatypes . . . 20
0.8.1 Alternatieve waarden . . . 20
0.8.2 Alternatieven met attributen . . . 21
0.9 Recursieve datatypes . . . 25
0.9.1 Voorbeeld: expressies . . . 25
0.9.2 Voorbeeld: vormen . . . 26
0.10 Voor de docent . . . 28
0.10.1 Tips voor het gebruik van Jupyter Notebook . . . 28
0.11 Over deze module . . . 29

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

Hoofdstuk	onderdelen
1	waarden, expressies en namen
1	functies: definitie en toepassing
2	typering, samengestelde waarden, lijsten
2	typering van functies
3	functies als waarden; map, fold
4	zelf-gedefinieerde data-types; functies daarvoor
4	geparametriseerde types; generieke functies
5	recursieve data-types; recursieve functies daarvoor
6	functies op lijsten; map, fold
7	bomen en de toepassingen daarvan
7	zoeken en sorteren
8	grotere voorbeelden

0.3.1 Waarden, expressies en namen

begrippen: 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 heeft in Haskell de vorm: `f 3`
  - de naam van de functie met daarachter het argument waarop de functie toegepast wordt.
  - in plaats van `f(3)` (zoals gebruikelijk in veel andere programmeertalen, of in de wiskunde)
  - functie-toepassing bindt sterker dan operatoren als `+`, `*`, enz.

voorbeelden: enkele uitgewerkte berekeningen
opdrachten: uitwerken van enkele berekeningen

0.3.2 Functies

begrippen: 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

0.3.3 Types en waarden

begrippen:
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.)

0.3.4 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.)

0.3.5 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.)

0.3.6 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)

0.3.7 Generieke functies (map enz.)

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

0.4 Waarden en functies

(Inleiding)

0.4.1 Waarden, namen en expressies

Begrippen
waarde, operator, expressie, prioriteit van operatoren (of "sterkte van binding"), naam (-definitie, -gebruik), lokale namen (let-constructie).

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*, een *teken* (*character*), of een logische waarde (`False` of `True`).

Voorlopig gebruiken we alleen *gehele getallen.* Later zullen we kennismaken met tekens, logische waarden, floating point getallen, en *samengestelde waarden*, zoals strings en lijsten.

In een **expressie** 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.

Opmerking. 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.

0.4.2 Namen: 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.

Lokale namen: let

De namen `a` en `b` die we hierboven gedefinieerd hebben, hebben betekenis voor het hele programma. Het zijn *globale namen. Met de `let`-constructie kun je een lokale naam definiëren: deze heeft alleen betekenis in de let-expressie:

```
let <naam>= <waarde>in <expressie met naam>
```

Voorbeeld:

```
let x = 20 in x + 3

23
```

Deze lokale naam `x` heeft buiten de let-constructie geen betekenis:

```
x

Runtime error: error: "<repl>": line 10, col 3: undefined value: x
```

Omdat de lokale naam buiten de let-constructie geen betekenis heeft, hadden we ook een andere naam kunnen gebruiken:

```
let zzz = 20 in zzz + 3

23
```

0.4.3 Functies: definitie en toepassing

Begrippen

- functie-definitie
- functie-naam
- functie-parameter - als lokale naam
- functie-expressie ("body"), met daarin de parameter
- toepassing van een functie op een (argument)waarde

In de voorbeelden hierboven staan de namen `a` en `b` voor getallen. Maar, we kunnen een naam ook koppelen aan een *functie*.

- de definitie van een functie koppelt de functie-naam aan een functie-waarde (expressie)

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

$$f(x) = x * x \tag{1}$$

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 geen haakjes gebruiken, in tegenstelling tot de wiskundige defintie.

functie-toepassing. We kunnen een functie *toepassen op een waarde.*

De functie `sqr` toegepast op de waarde `3` schrijven we in Haskell als: `sqr 3`

De waarde waarop de functie toegepast wordt heet ook wel de argument-waarde.

Voor functie-toepassing (Engels: *function application*) gebruik je in Haskell geen haakjes, net als in de definitie. In veel andere programmeertalen schrijf je: `sqr(3)` - net is meestal in de wiskunde.

Functie-toepassing in Haskell bindt sterker dan de operatoren: `sqr 3 + 4` lees je dan als `(sqr 3) + 4`. (Ga dit na.) Dit is dus iets anders dan `sqr (3 + 4)`.

De toepassing van deze functie op een argument, bijvoorbeeld `sqr 7`, resulteert in de expressie (berekening) waarin de waarde van het afgument 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

sqr 7

49
```

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 - 10
```

```
-1
```

Als je een andere volgorde (prioriteit) wilt afdwingen gebruik je haakjes:

```
sqr (a - 10)
```

```
49
```

Mogelijke voorbeelden:

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

```
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
==     {rekenen}
    25
```

Merk op dat we hetzelde 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

Een voorbeeld van een functie met meerdere parameters:

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

Deze functie pas je toe door twee argument-waarden na de functienaam te plaatsen:

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

```
23
```

Merk op: geen haakjes, geen komma. Maar wel haakjes voor de expressie `4*5`, omdat *functie-applicatie sterker bindt dan vermenigvuldigen.* Zonder deze haakjes zou de betekenis zijn: `(add 3 4) * 5` (*ga dit na*)

0.5 Een voorproefje van functioneel programmeren in Haskell

Je kunt Haskell als een rekenmachine gebruiken: je kunt expressies invoeren in de gebruikelijke notatie.

Denk erom dat je, zoals in alle programmeertalen, vermenigvuldiging altijd moet uitschrijven met de *=operator.

```
3 + 4
```

```
7
```

```
3 + 4 * 5
```

```
(3 + 4) * 5
```

Je kunt een waarde ook een naam geven, en die naam gebruiken op de plek van een waarde in een expressie:

```
a = 3
```

```
a + 4
```

Naast getallen als waarde kun je ook tekens (*chars*) en *strings* gebruiken. (Later)

Een functie-definitie ziet er vrijwel net zo uit als in de wiskunde:

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

Als je nu de waarde van `sqr` opvraagt, krijg een foutmelding: je kunt een functie-waarde niet afdrukken.

```
sqr
```

De aanroep (de "applicatie" ofwel het gebruik) van een functie schrijf je als de naam van de functie direct gevolgd door het argument. Je hebt geen haakjes nodig.

```
sqr 3
```

0.6 Typering en lijsten

Tot nu toe hebben we alleen gewerkt met gehele getallen als waarden. Maar Haskell biedt meerdere soorten waarden, zoals floating point getallen, tekens (characters), strings, tupes., lijsten, enz.

Het *type* van een waarde geeft de soort aan. Het type van een waarde bepaalt welke operaties (functies, operatoren) er voor die waarde mogelijk zijn. Door deze typering kan het Haskell systeem controleren of in een berekening de waarden en de operaties bij elkaar passen. Je kunt bijvoorbeeld niet een string optellen bij een getal: optellen (`+`) is alleen gedefinieerd voor getallen.

```
3 + 'a'
```

```
Runtime error: error: "<repl>": line 8, col 3: Cannot satisfy constraint: (Num Char)
     fully qualified: (Data.Num.Num Primitives.Char)
```

We hebben in Haskell te maken met:

- elementaire waarden, zoals getallen, tekens, en boolean waarden;
- samengestelde waarden, zoals lijsten en tupels;
- functie-waarden;
- waarden van zelf-gedefinieerde *data-types*.

In dit hoofdstuk behandelen we de eerste twee. In de volgende hoofdstukken komen de andere types en waarden aan de orde.

Elementaire waarden.

- type: `Int` - voorbeeld: `31415926`
- type: `Float` - voorbeeld: `3.1415926`, `2.3e5`
- type `Char` - voorbeeld: `'A'`, `'#'`
- type `Bool` - voorbeelden: `False`, `True`

Als je een naam definieert, zoals in `a = 10`, dan bepaalt het Haskell-systeem het type van de naam op basis van het type van de waarde (expressie).

Samengestelde waarden.

Een samengestelde waarde bevat (mogelijk meerdere) waarden van andere types. Voorbeelden van samengestelde waarden zijn lijsten en tupels.

Lijsten.

- een lijst bevat 0 of meer waarden van *eenzelfde type*
- als `a` een type is, dan is `[a]` het type van een "lijst van a-waarden"
  - `[]` - de lege lijst
  - `[1, 3, 5]` - lijst van gehele getallen - type: `[Int]`
  - `[3.4, 120e10, -7.5]` - lijst van floating point getallen - type: `[Float]`
  - `['a', 'b', '*']` - lijst van tekens - type: `[Char]`

Deze laatste waarde kun je ook schrijven als `"ab*"`. Het type `[Char]` noemen we meestal `String`.

In deze voorbeelden hebben we alleen letterlijke waarden in de lijsten opgenomen. Maar je kunt de elementen van een lijst ook uitrekenen met expressies (van het juiste type), bijvoorbeeld: `[1, a*12, abs (b - 7)]`

```
['a', 'b', '*']
```

```
"ab*"
```

Verkorte notatie. Voor lijsten met opeenvolgende gehele getallen is er een speciale verkorte notatie: `[a..b]` staat voor een lijst met alle getallen van `a` tot en met `b`. Bijvoorbeeld: `[3..6] = [3,4,5,6]`.

Een andere beknopte manier om een lijst te genereren is met behulp van *list comprehension*, zie XXXX.

Lijst-constructor: cons. De operator `:` (spreek uit: *cons*) kun je gebruiken om een element op kop van een lijst toe te voegen, bijvoorbeeld: `3 : [5, 7]` geeft `[3, 5, 7]`.

Je kunt de lijst `[3, 5, 7]` dan zien als een verkorte notatie voor: `3 : (5 : (7 : []))` of korter: `3 : 5 : 7 : []`. (De lijst-constructor *cons* is *rechts-associatief*.)

```
[3..12]
```

```
[3,4,5,6,7,8,9,10,11,12]
```

Operaties op lijsten

Wat kun je met lijsten? Welke opdrachten/functies zijn mogelijk?

- aaneenrijgen (concatenatie): `++` - voorbeeld: `[1, 2] ++ [3, 4]` of `"aap" + "noot" + "mies"`
- kop-element van een lijst: `head "aap"`
- staart van een lijst: `tail "aap"`
- lengte van een lijst: `length "aap"`
- omkeren van een lijst: `reverse "aap"`

```
[3 ,4] ++ [1, 2]
```

```
[3,4,1,2]
```

```
4 : 5 : [9, 10]
```

```
[4,5,9,10]
```

```
"aap" ++ "noot" ++ "mies"
```

```
"aapnootmies"
```

```
head "aap"
```

```
'a'
```

```
tail "aap"
```

```
"ap"
```

```
length "aap"
```

```
3
```

```
reverse "aap"
```

```
"paa"
```

Tupels

Soms heb je twee of drie (of meer) waarden die bij elkaar horen, en eigenlijk één waarde vormen. Denk bijvoorbeeld aan de coördinaten in een 2-dimensionale ruimte. Hiervoor kun je een *tupel* gebruiken, in dit geval een paar of 2-tupel. Het aantal waarden ligt vast, en is klein, maar de waarden hoeven niet van eenzelfde type te zijn.

Een tupel schrijf je door de waarden tussen haakjes te schrijven, gescheiden door komma's. Voorbeelden:

- `(10, 20)` – heeft type `(Int, Int)`
- `('a', 75)` – heeft type `(Char, Int)`
- `(1.0, 4, 3.4)`– heeft type `(Float, Int, Float)`

Een waarde in een tupel kan ook een samengestelde waarde zijn. Je kunt op die manier tupels en strings combineren, bijvoorbeeld:

- een tupel met lijsten `([1, 7] , "aap")` – heeft type `([Int], [Char])`

- een lijst met tupels: `[('a', 7), ('b', 35)]`– heeft type `[(Char, Int)]`

We hebben gezien dat je met de `( , , )` - notatie een tupel *construeert* uit samenstellende waarden. Je kunt op de volgende manieren een tupel *de-construeren* tot de samenstellende waarden:

- via het benoemen van de elementen:
  - `let (a, b) = ('x', 37) in b * 2`
- via de functies `fst`, `snd` (alleen voor 2-tupels ofwel *paren*)
  - `let b = snd ('x', 37) in b * 2`

0.6.1 Typering van functies

Het type van een functie beschrijft de types van de parameter(s) en van het resultaat. Door het type bij een functie te vermelden verduidelijk je (voor een deel) hoe die functie gebruikt kan worden, en voorkom je sommige fouten in het gebruik van de functie.

De notatie: `double ::  Int ->Int` geeft aan:

- dat `double` functie is (door de pijl: `->`)
- met een `Int`-waarde als parameter (het type voor de pijl)
- en een `Int`-waarde als resultaat (het type na de pijl)

Anders gezegd: `double` is een functie *van* `Int` (domein) *naar* `Int` (bereik).

```
double :: Int -> Int
double x = x + x

double 8

16
```

Toepassing van de functie `double` op een niet-`Int` argument geeft een foutmelding:

```
double 'A'

Runtime error: error: "<repl>": line 12, col 10: Cannot satisfy constraint: (Char ~ Int)
     fully qualified: (Primitives.~ Primitives.Char Primitives.Int)
```

0.7 Functies als waarden

In dit hoofdstuk maak je kennis met functies als waarden. Een functie kan overal gebruikt worden waar je een "normale" waarde zoals een getal of een string kunt gebruiken:

- een functie kan optreden als *parameter* van een andere functie;
- een functie kan optreden als *resultaat* van een functie;
- een functie kan onderdeel zijn van een samengestelde datastructuur, zoals een lijst;
- je kunt "rekenen met functies", bijvoorbeeld door de *compositie van twee functies*.

Je kunt ook anonieme functie-waarden kunt hebben: een functie-waarde hoeft niet beslist een naam te hebben, net zomin als elke getalwaarde of string een naam hoeft te hebben.

0.7.1 Lambda-expressies: anonieme functie-waarden

Tot nu toe hebben we functie-definities gezien van de vorm: `sqr x = x * x`. Dit lijkt nog niet op de vorm van andere naam-waarde koppelingen, zoals `a = 42`. De waarde `42` is hier een anonieme waarde, die in deze definitie gekoppeld wordt aan een naam.

Een lambda-expressie beschrijft een *anonieme functie*: deze bestaat alleen uit de parameters en de definiërende expressie. Bijvoorbeeld:

```
\ x -> x * x
```

Het symbool `\` spreek je uit als *lambda*, zie ook de gelijkenis met de griekse letter λ. Het pijltje `->` kun je uitspreken als `naar` of `geeft`. Merk de overeenkomst op tussen deze lambda-expressie en de notatie voor een functie-type.

Wat kun je met zo'n anonieme functie-expressie (of eigenlijk: functie-waarde)? Deze kun je gebruiken overal waar je een functie-waarde verwacht: in functie-definities, parameters of resultaten, enz.

Hiermee kunnen we de definitie van een functie schrijven op dezelfde manier als een "normale" naam-waarde definitie:

```
sqr :: Int -> Int
sqr = \ x -> x * x
```

De toepassing van een functie op een argument kun zien als het vervangen van de `\` door `let`, met de parameter-naam gekoppeld aan het argument; en het pijltje `->` vervangen door `in`

```
sqr 7
==    {invullen waarde van sqr}
    (\ x -> x * x) 7
==    {functie -toepassing op argument 7)
    let x = 7 in x * x
==    {invullen waarde van  x}
    7 * 7
==    {rekenen}
    49
```

```
sqr :: Int -> Int
sqr = \ x -> x * x

sqr 7

49

(\ y -> y * y) 7

49
```

Opdracht Definieer de functie `double` met een lambda-expressie.

functies en lambda-expressies met meerdere parameters

Een lambda-expressie kan ook meerdere parameters hebben:

```
add :: Int -> Int -> Int
add = \ x y -> x + y
```

```
add :: Int -> Int -> Int
add = \ x y -> x + y

add 3 4
```

```
7
```

0.7.2 Functie als operator, operator als functie

Een functie met twee parameters, zoals `add` hierboven, kun je ook als (binaire) operator *tussen de parameters* noteren, zoals een `+` of `*` operator. Je gebruikt daarvoor "backticks" (“‘) rond de naam.

```
3 ‘add‘ 4 ‘add‘ 5
```

Alleen de notatie verandert, het principe van functie-aanroep blijft gelijk.

```
3 ‘add‘ 4 ‘add‘ 5
```

```
12
```

Operatoren als functie. Je kunt een operator, zoals `+` ook als functie gebruiken, door deze tussen haakjes te plaatsen: `(+)`. bijvoorbeeld: `(+) 3 4`

```
(+) 3 4
```

Bij ingewikkelder expressies moet je er dan rekening mee houden dat functie-toepassing "links-associatief" is. In de onderstaande expressie is daarom een extra haakjes-paar nodig voor de vermenigvuldiging.

```
(+) 3 ((*) 4 5)
```

```
23
```

0.7.3 Functie als parameter: map, reduce, filter

Soms wil je een functie toepassen op alle waarden in een datastructuu, zoals een lijst. Hiervoor heb je in Haskell (en andere talen) de functie `map` die zowel een functie als een lijst als parameter heeft: deze `map` past de betreffende functie toe op alle elementen van de lijst:

`map f [a, b, c, ...] = [f a, f b, fc, ...]`

Het type van `map` voor `Int`-lijsten kun je dan zien als: `map :: (Int ->Int) ->[Int] ->[Int]` De eerste parameter is een functie, de tweede is een Int-lijst, het resultaat is ook weer een Int-lijst.

```
sqr :: Int -> Int
sqr x = x * x
```

Voorbeeld

```
map sqr [1,3,7]
==    {"definitie" map invullen}
    [sqr 1, sqr 3, sqr 7]
==    {definitie sqr invullen}
    [let x=1 in x*x, let x=3 in x*x, let x=7 in x*x]
==    {definitie (waarde) van x invullen}
    [1*1, 3*3, 7*7]
==    {rekenen}
    [1, 9, 49]

map sqr [1, 3, 7]

[1,9,49]

double :: Int -> Int
double x = x + x

map double [1..10]

[2,4,6,8,10,12,14,16,18,20]
```

Je kunt ook een *anonieme functie* (lambda expressie) meegeven aan `map`:

```
map (\x -> x * x * x) [1..10]

[1,8,27,64,125,216,343,512,729,1000]
```

foldl, foldr Een andere veel voorkomende operatie is om een functie *tussen* alle elementen van een lijst uit te voeren, bijvoorbeeld het optellen van alle elementen.

`[1, 3, 7] ->1 + 3 + 7 ->11`

Als de lijst leeg is, is het resultaat 0 (voor optelling). Eigenlijk kun je bovenstaande dan schrijven als:

`[1, 3, 7] ->1 + 3 + 7 + 0 ->11`

(Wat is de waarde die je moet gebruiken voor het vermenigvuldigen van alle elementen van een lijst? Anders gezegd: welke bijdrage levert de lege lijst aan de totale vermenigvuldiging?)

De functie `foldl` (*fold left*) in Haskell past een operator `f` (functie met 2 parameters) toe *tussen* alle elementen van een lijst, waarbij het resultaat voor de lege lijst gelijk is aan de "nul-waarde" `z`:

```
foldl :: (Int -> Int) -> Int -> [Int] - Int
foldl f z [a, b, c, ...] = a `f` b `f` c `f` ... `f` z
```

Dit kun je ook zien als het vervangen van de const-constructor `:` door de functie (operator) `f`, waarbij de waarde voor de lege lijst gelijk is aan de "nul-waarde" `z`:

```
foldl f z (a : b : c ... : []) = a `f` b `f` c ... `f` z
```

Voorbeeld:

```
foldl (+) 0 (1 : 3 : 7 : []) = 1 + 3 + 7 + 0 = 11
```

De functie `foldl` vouwt de lijst wordt eigenlijk "opgevouwen" tot een enkele waarde met behulp van de functie `f`, vandaar de naam `foldl` (fold left).

Naast `foldl` heb je ook `foldr` (*fold right*). Het verschil is dat `foldl` van links naar rechts rekent (links-associatief), en foldr van rechts naar links (rechts-associatief). Voor het optellen van gehele getallen maakt dat geen verschil, maar bij andere functies kan dat een ander resultaat geven.

```
foldl f z (a : b : c ... : []) = ((((a `f` b) `f` c) ...) `f` z)
foldr f z (a : b : c ... : []) = (a `f` (b `f` (c `f` (... `f` z))))
```

reduce In andere programmeertalen heet de functie `foldl` vaak *reduce*: deze reduceert een lijst, of een andere datastructuur, tot een enkele waarde.

De combinatie **map/reduce** (en soms ook *filter*) komt tegenwoordig in veel programmeertalen voor: dit maakt het mogelijk om potentieel parallellisme uit te drukken, omdat de volgorde van de afzonderlijke operaties er niet toe doet.

```
foldl (\x y -> x + y) 0 [1..10]
```

```
55
```

```
foldl (\x y -> x + y) 0 []
```

```
0
```

Operatoren als functie. Je kunt een operator, zoals `+` ook als functie gebruiken, door deze tussen haakjes te plaatsen: `(+)`. Bovenstaand voorbeeld kun je dan ook schrijven als:

0.7.4 Functie als resultaat

Je kunt een functie-waarde ook gebruiken als resultaat van een functie. Bestudeer het volgende voorbeeld:

```
addx :: Int -> (Int -> Int)
addx x = \ y -> x + y

addtwo = addx 2
...
addtwo 9
```

De functie-toepassing `addx 2` heeft als resultaat de functie-waarde `\ y ->2 + y`

```
addx 2
```

== {definitie van addx invullen} let x = 2 in \ y ->x + y == {definitie van x invullen} \ y ->2 + y

We kunnen de definitie van `addx` ook schrijven als geneste lambda-expressies:

```
addx :: Int -> (Int -> Int)
addx = \ x -> (\ y -> x + y)
```

De haakjes rond het resultaat-type en de haakjes rond de geneste lambda-expressie mogen we weglaten, omdat de pijl `->` *rechts-associatief* is:

```
addx :: Int -> Int -> Int
addx = \ x -> \ y -> x + y
```

De functie `addx` is nu eigenlijk een gewone functie met twee parameters. Als je alleen het eerste argument opgeeft, krijg je een functie die je later op het tweede argument kunt toepassen.

Alle functies met twee of meer parameters kunnen we beschouwen als *geneste lambda-expressies*. We kunnen de functies *partieel parametriseren*, door deze toe te passen op 1 argument: we krijgen dan een functie die we later op de volgende argumenten kunnen toepassen.

Het verschijnsel dat je functies met meerdere parameters kunt beschouwen als geneste lambda-expressies, die je partieel kunt parametriseren, heet Currying, naar de logicus Haskell (!) Curry. Je mag die dus nu bij zijn voornaam noemen.

```
addx :: Int -> (Int -> Int)
addx x = \ y -> x + y

addtwo = addx 2

addtwo 3

5

addx 2 3

5
```

Deze Currying kunnen we toepassen op alle functies en operatoren met twee of meer parameters, zoals ook bijvoorbeeld de normale reken-operatoren:

```
succ = (+) 1

succ 4

5
```

We kunnen dit ook gebruiken om gespecialiseerde lijst-functies te maken, op basis van algemene lijstfuncties zoals `map` of `foldl`.

```
sum = foldl (+) 0

sum [1..10]

55
```

0.7.5 Functies in datastructuren

lijst met functies.

0.8 Zelf-gedefinieerde datatypes

Begrippen

- type-synoniem (alternatieve naam)
- data-type definitie: opsomming van alternatieve waarden
- alternatieven: constructors
- alternatieven met attributen

0.8.1 Alternatieve waarden

De datatypes die we tot nu toe gezien hebben, zowel de elementaire types zoals getallen, als de samengestelde types lijsten en tupels, zijn voorgedefinieerd. Maar, in Haskell kunnen we ook zelf datatypes definiëren.

De eenvoudigste vorm van een datatype-definitie geeft een *opsomming van de namen* van de verschillende waarden. Deze waarde-namen heten ook wel *constructors* - waarom, zal straks duidelijker worden. Deze alternatieven kunnen ook van extra waarden (attributen) voorzien zijn, zoals we verderop zullen zien.

De namen van types en de namen van constructors beginnen in Haskell met een hoofdletter.

Voorbeeld: kleuren

```
data Color = Red | Orange | Yellow | Green | Blue | Indigo | Violet deriving (Show)
```

De namen van de alternatieven beginnen altijd met een hoofdletter. Zo'n naam heet een *constructor*, omdat je daarmee een waarde van het betreffende data-type maakt.

De toevoeging `deriving (Show)` betekent dat de `show`-functie voor deze waarden automatisch gedefinieerd wordt; deze laat de waarde zien zoals je deze in een Haskell-programma noteert. Deze `show`-functie wordt impliciet gebruikt om het resultaat van een cel te tonen.

We kunnen een kleur-waarde toekennen aan een naam, bijvoorbeeld:

```
myColor = Yellow

show myColor

"Yellow"

myColor

Yellow
```

We kunnen functies definiëren om met deze waarden te "rekenen". Eigenlijk moeten we alle vormen van rekenen met deze waarden nu zelf definiëren. De functies die we zo definiëren moeten voor elke Color-waarde een resultaat definiëren. We kunnen dat in Haskell doen door een definitie per Color-alternatief te geven:

```
nextColor :: Color -> Color
nextColor Red = Orange
nextColor Orange = Yellow
nextColor Yellow = Green
nextColor Green = Blue
nextColor Indigo = Violet
nextColor Violet = Violet
```

De structuur van deze functie volgt de structuur van het datatype van de parameter.

```
nextColor (nextColor myColor)
```

```
Blue
```

We hebben hier de functie `nextColor` gedefinieerd door een definitie voor elke mogelijke vorm van `Color`. Deze constructie heet ook wel *pattern matching*.

Een alternatief is om een "case analysis" binnen de functie-definitie uit te voeren, zoals in dit voorbeeld:

```
nextColor :: Color -> Color
nextColor c =
    case c of
    Red -> Orange
    Orange -> Yellow
    Yellow -> Green
    Green -> Blue
    Indigo -> Violet
    Violet -> Violet
```

Bool als data-type

Een aantal types zijn voorgedefinieerd in de Haskell "standard prelude": de standaard-library van Haskell. Het type Bool is daarin bijvoorbeeld gedefinieerd als:

```
data  Bool  =  False | True
```

0.8.2 Alternatieven met attributen

Voorbeeld: vormen

In het eenvoudige data-type `Color` spreken de waarden voor zich. Maar bij complexere types kunnen de alternatieven *attributen* hebben.

Een voorbeeld hiervan is het type `Shape` (voor een 2-dimensionale geometrische vorm). We onderscheiden (in eerste instantie) cirkels en rechthoeken:

- een cirkel heeft een middelpunt (punt) en de straal (Float)
- een rechthoek een positie (punt, voor de linker-bovenhoek), een breedte en een hoogte (Floats)
- een punt is een *2-tupel* van Floats: x- en y-coördinaten.

Als afkorting voeren we het type `Point` in: een tupel 2 getallen: de x- en y-coördinaat van het punt.

```
type Point = (Float, Float)
```

> Dit is een voorbeeld van een **type-synoniem**: overal waar de naam `Point` gebruikt wordt, kun je ook `(Float, Float)` schrijven of lezen.

We onderscheiden twee vormen: een cirkel, met middelpunt en straal; en een rechthoek, met positie, breedte en hoogte.

```
data Shape = Circle Point Float | Rect Point Float Float
```

Uitzoeken *(Hebben we geen andere manier om deze eigenschappen te benoemen en te documenteren?)*

Uitzoeken: attributen zoals lijndikte, lijnkleur en vulkeur zijn eigenlijk gemeenschappelijk voor alle vormen. Hoe kun je dat het best uitdrukken in Haskell? Een vorm van "overerving"?

Later zullen we toevoegen als vormen:

- lijnstuk (met 2 coördinaten: begin- en eindpunt)
- pad (een lijst van coördinaten; reeks aaneengesloten lijnstukken)
- tekst

We kunnen nu een functie definiëren voor het uitrekenen van de oppervlakte van een (gesloten) figuur. We moeten in die functie de verschillende soorten vormen onderscheiden. In Haskell kan dat door de functie voor elk alternatief afzonderlijk te definiëren:

```
area :: Shape -> Float
area (Circle centre radius) = pi * radius * radius
area (Rect position width height) = width * height
```

We demonstreren dit met twee voorbeeld-vormen:

```
shape1 = Circle (50, 50) 20
shape2 = Rect (20, 30) 100 20

area shape1

1256.6371

area shape2

2000.0
```

We kunnen nog meer functies definiëren voor deze vormen

- `translate :: Point ->Shape ->Shape`
- `scale :: Float ->Shape ->Shape`
- `tosvg :: Shape ->String`

Als voorbeeld werken we deze laatste functie uit: `tosvg`, om de SVG-string voor deze vorm te bepalen. Deze kunnen we dan kopiëren in een SVG-figuur, om deze te tonen. (Zie het svg-display notebook.)

De attributen van een SVG-element hebben allemaal dezelfde structuur: `name="value"`. Hier worden dezelfde dubbele quote-tekens gebruikt als voor Haskell-strings Dat betekent dat we deze niet zomaar in een string kunnen opnemen: we hebben een *escape*-notatie nodig. De volgende functie maakt deze structuur aan:

```
attr :: String -> Float -> String
attr name value = name ++ "=\"" ++ (show value) ++ "\" "
```

de functie `show 2.3` zet het getal om in een string, hier `"2.3"`

```
attr "cx" 12.34

"cx=\"12.34\" "
```

```
tosvg :: Shape -> String
tosvg (Circle (mx, my) r) = "<circle " ++ (attr "cx" mx) ++ (attr "cy" my) ++ (attr "r" r) ++
tosvg (Rect (mx, my) w h) = "<rect " ++ (attr "x" mx) ++ (attr "y" my) ++ (attr "width" w) ++

tosvg shape1

"<circle cx=\"50.0\" cy=\"50.0\" r=\"20.0\"  /> \n"
```

Om dit te kunnen gebruiken in een SVG-figuur, moeten we deze string-waarde in de uitvoer-vorm hebben, in plaats van in de Haskell-notatie; met andere woorden, zonder de quote-tekens, en met de escape-tekens zooals `'\n'` en `'\"'` verwerkt. Hiervoor gebruiken we de functie `putStr`:

```
putStr (tosvg shape1)

<circle cx="50.0" cy="50.0" r="20.0"  />

putStr (tosvg shape2)

<rect x="20.0" y="30.0" width="100.0" height="20.0"  />
```

Dit resultaat kopiëren we naar de lege regel in de cel met `svgimage=...`, in het svg-display notebook. (Zie de handleiding daar.) Voor een figuur met `shape1` en `shape2` geeft dit:

Opdrachten

- toevoegen van andere (SVG) vormen, bijvoorbeeld *ellipse* en *line*.
  - je moet dan per vorm het alternatief toevoegen aan het type; en de bijbehorende functies uitbreiden.
- toevoegen van extra attributen aan de SVG-vormen, bijvoorbeeld de vul-kleur

Voorbeeld: expressies

Als tweede voorbeeld van zelf-gedefineerde datatypes behandelen we *expressies*. We willen expressies als data kunnen opbouwen. We maken hier een eerste begin, en zullen zien dat deze vorm nogal beperkt is. Voor algemene expressies hebben we *recursie* nodig: dat komt in het volgende hoofdstuk aan bod.

We willen een expressie kunnen beschrijven in de vorm van een datatype. We kunnen deze expressies als waarde gebruiken, en op verschillende manieren manipuleren. Als operatoren gebruiken we in eerste instantie alleen optelling (`Plus`) en vermenigvuldiging (`Times`), en een elementaire waarde (`Num`). Als operanden voor de operatoren gebruiken we voorlopig getallen: dat geeft al direct de beperking aan. In het volgende hoofdstuk zullen we dat generaliseren.

```
data Expr = Num Float | Plus Float Float | Times Float Float deriving (Show)
```

Enkele voorbeelden van waarden van dit datatype:

```
pi = Num 3.1415629
twopi = Times 2.0 3.1415629
expr3 = Plus 2 21

pi
```

```
Runtime error: error: "<repl>": line 18, col 31: Cannot satisfy constraint: (Float ~ Expr)
     fully qualified: (Primitives.~ Primitives.Float Inline.Expr)
```

```
twopi
```

```
expr3
```

We kunnen nu een functie `eval` maken, voor het evalueren van een expressie:

```
eval :: Expr -> Float
eval (Num val) = val
eval (Plus l r) = l + r
eval (Times l r) = l * r
```

```
eval pi
```

We kunnen de operatoren ook in *postfix* afdrukken. Daarbij krijgen we eerst de operanden, en daarna de operator. (Sommige rekenmachines werken met die notatie. Een voordeel is dat je dan geen haakjes nodig hebt.)

```
postfix :: Expr -> String
postfix (Num val) = show val
postfix (Plus l r) = (show l) ++ " " ++ (show r) ++ " " ++ "+"
postfix (Times l r) = (show l) ++ " " ++ (show r) ++ " " ++ "+"
```

```
postfix expr3
```

Recursie als volgende stap. We hebben *recursie* nodig: eigenlijk willen we kunnen schrijven:

```
twopi = Times 2 pi
```

Daarvoor moet de waarde van een operand (attribuut) van `Times` een `Num Float` waarde kunnen zijn, of in het algemeen: een `Expr` waarde. Daar gaan we in het volgende hoofdstuk mee aan de slag.

0.9 Recursieve datatypes

0.9.1 Voorbeeld: expressies

We hebben in het vorige hoofdstuk gezien dat we voor algemene expressies recursie nodig hebben: een argument van een operator-knoop kan weer een expressie zijn.

Dit kunnen we weergeven in de volgende definitie: (later voegen we meer operatoren toe)

```
data Expr = Add Expr Expr
          | Mul Expr Expr
          | Val Float
```

Merk op:

- in de definitie van `Expr` komt `Expr` weer voor in de definiërende termen in de rechterkant: dit is *recursie*. (Letterlijk: komt weer voor)
- het alternatief `Val Float` is niet recursief. In een recursieve (data)definitie moet tenminste één alternatief niet-recursief zijn.

Met behulp van deze definitie kunnen we nu waarden construeren:

```
expr1 = Add (Val 1) (Val 2.5)

expr2 = Add (Mul (Val 2) (Val 7.5)) (Val 3)
```

We kunnen nu functies definiëren op `Expr`-waarden. Een dergelijke functie moet een alternatieve definitie hebben voor elk alternatief in de data-definitie. Anders gezegd: een functie die werkt op `Expr`-waarden volgt de structuur van het `Expr` data-type.

Als voorbeeld geven we een functie om een `Expr`-waarde uit te rekenen. Merk op dat we haakjes om de parameters van `eval` moeten schrijven, omdat `eval Add a b` gelezen wordt als `(eval Add) a b`

```
eval :: Expr -> Float
eval (Add a b) = (eval a) + (eval b)
eval (Mul a b) = (eval a) * (eval b)
eval (Val a) = a

eval expr1

3.5

eval expr2

18.0
```

We schrijven hier de uitwerking van deze laatste uitdrukking uit:

```
eval expr2
=   {def. expr2}
  eval (Add (Mul (Val 2) (Val 7.5)) (Val 3)
=   {def. eval, voor Mul -alternatief}
  (eval (Mul (Val 2) (Val 7.5)) + (eval (Val 3))
=   {def. eval, voor Add -alternatief}
  ((eval (Val 2)) * (eval (Val 7.5))) + (eval (Val 3))
=   {def. eval, voor alle Val -alternatieven}
  ((2) * (7.5)) + (3)
=   {rekenen}
  (15.0) + (3)
=   {rekenen}
  18.0
```

Voorbeeld. Met behulp van de functie `postfix` zetten we een `Expr`-waarde om in een string, in *postfix-formaat*: eerst de argumenten, dan de operator.

```
postfix :: Expr -> String
postfix (Add a b) = (postfix a) ++ (postfix b) ++ " + "
postfix (Mul a b) = (postfix a) ++ (postfix b) ++ " * "
postfix (Val a) = show a ++ " "

postfix expr2

"2.0 7.5  * 3.0  + "
```

Opdracht Maak een functie om een `Expr`-waarde om te zetten in prefix-formaat: eerst de operator, dan de argumenten.

Variant: probeer de string in Haskell-formaat te maken, waarbij je de operatoren als functies beschouwt. (Gebruik de Haskell-notatie: `(+) 3 4` - met de operatoren tussen haakjes).

0.9.2 Voorbeeld: vormen

We voegen aan de grafische figuren *groepering* toe, waarbij we een lijst van grafische elementen samenvoegen tot één element. Bovendien voegen we *tranformaties* toe aan deze groepen.

Groepering is in tekenprogramma's een gebruikelijke operatie. De elementen in een groep kun je dan tegelijk verplaatsen, en bijvoorbeeld ook tegelijk kopiëren. Als je de afzonderlijke elementen wilt bewerken, kun je deze groepering (tijdelijk) opheffen.

In SVG kun je in een groep ook de vormgeving van de elementen in de groep bepalen, zoals de vulkeur of de pendikte.

(Wij voegen hier een positie toe aan een groep: dat komt overeen met een *translate* van de elementen in de groep.)

```
type Point = (Float, Float)

data Shape = Circle Point Float
           | Rect Point Float Float
           | Text Point String
           | Line Point Point
           | Grp Point [Shape]
             deriving (Show)

shape1 = Circle (10, 10) 20
shape2 = Rect (20, 10) 100 20

shape3 = Line (10, 10) (100, 100)

attr :: String -> Float -> String
attr name value = name ++ "=\"" ++ (show value) ++ "\" "

tosvg :: Shape -> String
tosvg (Circle (mx, my) r) = "<circle " ++ (attr "cx" mx) ++ (attr "cy" my) ++ (attr "r" r) ++
tosvg (Rect (mx, my) w h) = "<rect " ++ (attr "x" mx) ++ (attr "y" my) ++ (attr "width" w) ++
tosvg (Text (mx, my) s) = "<text x=" ++ (attr "x" mx) ++ (attr "y" my) ++ ">" ++ s ++ "</text>
tosvg (Line (ax, ay) (bx, by)) = "<line " ++ (attr "x1" ax) ++ (attr "y1" ay) ++ (attr "x2" bx
tosvg (Grp (mx, my) lst) = "<g transform=\"translate(" ++ (show mx) ++ " " ++ (show my) ++ ")\
                           where svg_lst = map tosvg lst
                                 elems = foldr (++) "" svg_lst

putStr (tosvg shape1)
```

```
<circle cx="10.0" cy="10.0" r="20.0"  />

shape4 = Grp (10, 20) [shape1, Grp (50, 50) [shape1, shape2, shape3], shape2]

shape4

Grp (10.0,20.0) [Circle (10.0,10.0) 20.0,Grp (50.0,50.0) [Circle (10.0,10.0) 20.0,Rect (20.0,1

putStr (tosvg shape4)

<g transform="translate(10.0 20.0)" >
<circle cx="10.0" cy="10.0" r="20.0"  />
<g transform="translate(50.0 50.0)" >
<circle cx="10.0" cy="10.0" r="20.0"  />
<rect x="20.0" y="10.0" width="100.0" height="20.0" />
<line x1="10.0" y1="10.0" x2="100.0" y2="100.0" />
</g>
<rect x="20.0" y="10.0" width="100.0" height="20.0" />
</g>
```

Dit geeft de volgende figuur:

Opdracht

- maak een figuur die een huisje of een poppetje voorstelt (en geef deze een naam)
- maak een figuur die bovenstaande 3-maal naast elkaar herhaalt.

0.10 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.

0.10.1 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*).
-

0.11 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?)