Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

College 2010-2011
2. Notatie, Recursie, Lijsten

Doaitse Swierstra

Utrecht University

September 19, 2010

Opfrissing: Algemene vorm functiedefinitie

Elke functie-definitie heeft dus de volgende vorm:

- de naam van de functie
- de namen van eventuele parameters
- een $=$-teken
- een expressie waar de parameters, standaardfuncties en zelf-gedefinieerde functies in mogen voorkomen.

Where ...

In de definitie van de functie *abcFormule* komen de expressies $sqrt\ (b * b - 4.0 * a * c)$ en $(2.0 * a)$ twee keer voor. Dus beter:

$$
\begin{aligned}
abcFormule'\ a\ b\ c = [&(-b + d)\ /\ n \\
,&(-b - d)\ /\ n \\
]& \\
\textbf{where}\ &d = sqrt\ (b * b - 4.0 * a * c) \\
&n = 2.0 * a
\end{aligned}
$$

Where ...

In de definitie van de functie *abcFormule* komen de expressies $sqrt\ (b * b - 4.0 * a * c)$ en $(2.0 * a)$ twee keer voor. Dus beter:

$$
\begin{aligned}
abcFormule'\ a\ b\ c = [&(-b + d)\ /\ n \\
&,(-b - d)\ /\ n \\
&] \\
&\textbf{where}\ d = sqrt\ (b * b - 4.0 * a * c) \\
&\qquad\quad n = 2.0 * a
\end{aligned}
$$

```
? abcFormule' 1.0 1.0 0.0
[0.0, -1.0]
(18 reductions, 66 cells)
? abcFormule 1.0 1.0 0.0
[0.0, -1.0]
(24 reductions, 84 cells)
```

Gevalsonderscheid in Functiedefinities

Soms is het nodig om in de definitie van een functie meerdere gevallen te onderscheiden Dat gebeurt bijvoorbeeld in de definitie van de functie *signum*:

$$
\begin{array}{lrcr}
\mathit{signum}\ x & \mid x > 0 & = & 1 \\
& \mid x \equiv 0 & = & 0 \\
& \mid x < 0 & = & -1
\end{array}
$$

De definities voor de verschillende gevallen worden ‘bewaakt’ door Boolese expressies, die dan ook *guards* worden genoemd.

Patronen

Je kunt een functie definiëren met verschillende patronen als formele parameter:

$$
\begin{aligned}
som\ [\,] &= 0 \\
som\ [x] &= x \\
som\ [x,y] &= x+y \\
som\ [x,y,z] &= x+y+z
\end{aligned}
$$

Patronen

Je kunt een functie definiëren met verschillende patronen als formele parameter:

$$
\begin{aligned}
som\ [\,] &= 0 \\
som\ [x] &= x \\
som\ [x,y] &= x+y \\
som\ [x,y,z] &= x+y+z
\end{aligned}
$$

Bij aanroep van de functie kijkt de interpreter of de parameter 'past' op een van de definities; de aanroep $som\ [3,4]$past bijvoorbeeld op de derde regel van de definitie.

Patronen

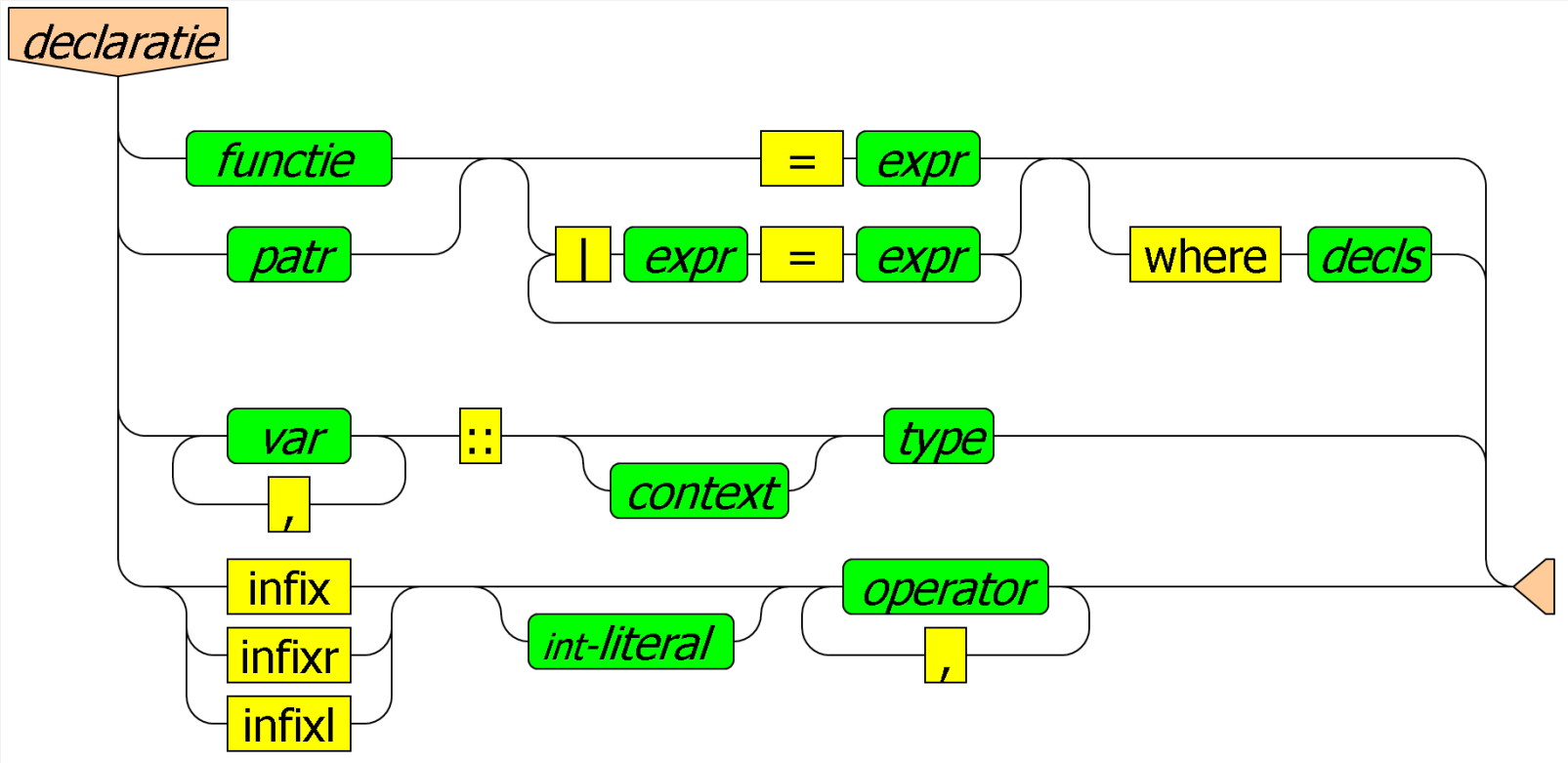
Je kunt een functie definiëren met verschillende patronen als formele parameter:

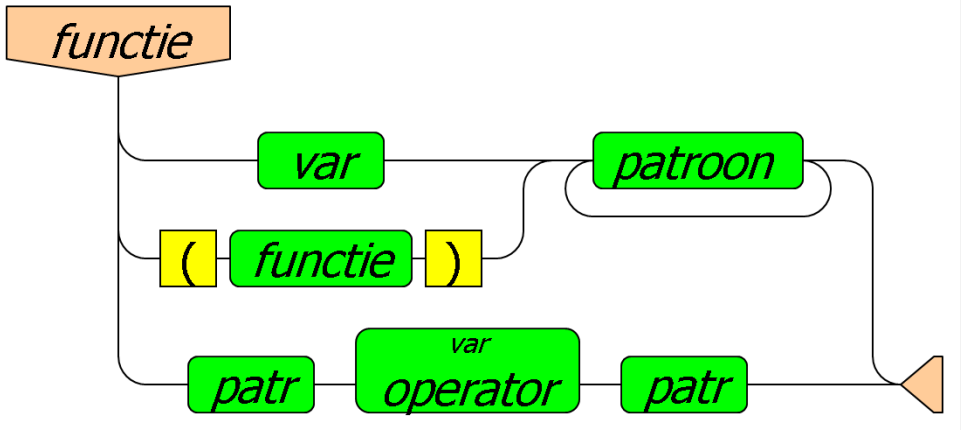
$$
\begin{aligned}
som\ [\,] &= 0 \\
som\ [x] &= x \\
som\ [x,y] &= x+y \\
som\ [x,y,z] &= x+y+z
\end{aligned}
$$

Bij aanroep van de functie kijkt de interpreter of de parameter 'past' op een van de definities; de aanroep $som\ [3,4]$ past bijvoorbeeld op de derde

Let op: als het argument een lijst is met meer dan 3 elementen gaat dit niet goed!

Syntax Diagram





Soorten van patronen

De volgende constructies zijn toegestaan als patroon:

- Getallen (bijvoorbeeld 3)
- Constanten zoals *True*, *False* en $[\,]$;
- Namen (bijvoorbeeld x);
- Lijsten, waarvan de elementen ook weer patronen zijn (bijvoorbeeld $[1, x, y]$);
- De operator : met patronen links en rechts, en haakjes buitenom! (bijvoorbeeld $(x : xs)$);

Voorbeelden uit de Prelude

Met behulp van patronen zijn een aantal belangrijke functies te definiëren:

$$
\begin{aligned}
\mathit{True} \wedge \mathit{True} &= \mathit{True} \\
\_ \wedge \_ &= \mathit{False} \\
\mathit{False} \vee \mathit{False} &= \mathit{False} \\
\_ \vee \_ &= \mathit{True}
\end{aligned}
$$

Voorbeelden uit de Prelude

Met behulp van patronen zijn een aantal belangrijke functies te definiëren:

$$
\begin{aligned}
\mathit{True} \wedge \mathit{True} &= \mathit{True} \\
\_ \wedge \_ &= \mathit{False} \\
\mathit{False} \vee \mathit{False} &= \mathit{False} \\
\_ \vee \_ &= \mathit{True}
\end{aligned}
$$

Met de operator $:$ kunnen lijsten worden opgebouwd. gebruikt in patronen kunnen nu de standaardfuncties geschreven worden die lijsten weer afbreken:

$head$ en $tail$

$$
\begin{aligned}
\mathit{head}\ (x:y) &= x \\
\mathit{tail}\ \ (x:y) &= y
\end{aligned}
$$

Voorbeelden uit de Prelude

Met behulp van patronen zijn een aantal belangrijke functies te definiëren:

$$
\begin{aligned}
\mathit{True} \wedge \mathit{True} &= \mathit{True} \\
\_ \wedge \_ &= \mathit{False} \\
\mathit{False} \vee \mathit{False} &= \mathit{False} \\
\_ \vee \_ &= \mathit{True}
\end{aligned}
$$

Met de operator $:$ kunnen lijsten worden opgebouwd. gebruikt in patronen kunnen nu de standaardfuncties geschreven worden die lijsten weer afbreken:

head en *tail*

Lijsten mogen dus niet leeg zijn!

$$
\begin{aligned}
\mathit{head}\ (x:y) &= x \\
\mathit{tail}\ \ (x:y) &= y
\end{aligned}
$$

Recursie, en daar gaat het om ...

Recursieve functies (d.w.z. functies die zichzelf aanroepen, zijn echter zinvol onder de volgende twee voorwaarden:

- de parameter van de recursieve aanroep is eenvoudiger (bijvoorbeeld: numeriek kleiner, of een kortere lijst) dan de parameter van de te definiëren functie;
- voor een basis-geval is er een niet-recursieve definitie.

Recursie, en daar gaat het om ...

Recursieve functies (d.w.z. functies die zichzelf aanroepen, zijn echter zinvol onder de volgende twee voorwaarden:

- de parameter van de recursieve aanroep is eenvoudiger (bijvoorbeeld: numeriek kleiner, of een kortere lijst) dan de parameter van de te definiëren functie;
- voor een basis-geval is er een niet-recursieve definitie.

Een recursieve definitie van de faculteit-functie is de volgende:

$$\begin{aligned} fac\ n \mid & n \equiv 0 = 1 \\ \mid & n > 0 = n * fac\ (n-1) \end{aligned}$$

Recursie gaat goed met lijsten samen ..

Vrijwel alle functies op lijsten volgen een speciaal patroon:

$$\begin{aligned} som\ [\,] &= 0 \\ som\ (kop : staart) &= kop + som\ staart \end{aligned}$$

Recursie gaat goed met lijsten samen ..

Vrijwel alle functies op lijsten volgen een speciaal patroon:

$$\begin{aligned} som\ [\,] &= 0 \\ som\ (kop : staart) &= kop + som\ staart \end{aligned}$$

Tip: als je zelf zo'n functie schrijft, begin dan altijd met het lege geval, controleer het type, en kijk of dat je aanstaat.

Niet alle recursie gaat goed ...

De volgende functie is niet erg zinvol:

$$f\ x = f\ x$$

Niet alle recursie gaat goed ...

De volgende functie is niet erg zinvol:

$$f\ x = f\ x$$

Gaat zoiets altijd fout?

Niet alle recursie gaat goed ...

De volgende functie is niet erg zinvol:

$$f\ x = f\ x$$

Gaat zoiets altijd fout? Belangrijk in geval van zo'n identieke aanroep is dat een functie productief is, d.w.z. dat er iets geproduceerd is voordat er dieper in een expressie een recursieve aanroep plaatsvindt:

Niet alle recursie gaat goed ...

De volgende functie is niet erg zinvol:

$$f\ x = f\ x$$

Gaat zoiets altijd fout? Belangrijk in geval van zo'n identieke aanroep is dat een functie productief is, d.w.z. dat er iets geproduceerd is voordat er dieper in een expressie een recursieve aanroep plaatsvindt:

repeat

$$repeat\ x = x : repeat\ x$$

Is dit zinvol?

In tegenstelling tot wat je zou denken is dit geen onzin:

Is dit zinvol?

In tegenstelling tot wat je zou denken is dit geen onzin:

$$head :: [a] \to a$$
$$head\ (x : xs) = x$$
$$head\ (repeat\ 3) \Rightarrow head\ (3 : repeat\ 3) \Rightarrow 3$$

Bedenk dat in Haskell een expressie niet verder wordt uitgerekend dan nodig is! Voor de pattern match in de definitie van *head* is het voldoende om de definitie van *repeat* één keer uit te vouwen, en zo de constructor : "zichtbaar" te maken, zodat pattern matchingslaagt.

Is dit zinvol?

In tegenstelling tot wat je zou denken is dit geen onzin:

$$
\begin{aligned}
&head :: [a] \to a \\
&head\ (x : xs) = x \\
&head\ (repeat\ 3) \Rightarrow head\ (3 : repeat\ 3) \Rightarrow 3
\end{aligned}
$$

Bedenk dat in Haskell een expressie niet verder wordt uitgerekend dan nodig is! Voor de pattern match in de definitie van *head* is het voldoende om de definitie van *repeat* één keer uit te vouwen, en zo de constructor : "zichtbaar" te maken, zodat pattern matchingslaagt. Het is de pattern matching in Haskell die de eveluatie aandrijft!

Normale recursie

Recursieve functies zijn wèl zinvol onder de volgende voorwaarden:

- de parameter van de recursieve aanroep is eenvoudiger (bijvoorbeeld: numeriek kleiner, of een kortere lijst) dan de parameter van de te definiëren functie;
- voor een basis-geval is er een niet-recursieve definitie.
- of: de functie is productief! Dit laatste is uniek voor lazy geëvalueerde talen.

Normale recursie

Recursieve functies zijn wèl zinvol onder de volgende voorwaarden:

- de parameter van de recursieve aanroep is eenvoudiger (bijvoorbeeld: numeriek kleiner, of een kortere lijst) dan de parameter van de te definiëren functie;
- voor een basis-geval is er een niet-recursieve definitie.
- of: de functie is productief! Dit laatste is uniek voor lazy geëvalueerde talen.

Een recursieve definitie van de faculteit-functie is de volgende:

$$\begin{aligned} fac\ n \mid n \equiv 0 &= 1 \\ \mid n > 0 &= n * fac\ (n-1) \end{aligned}$$

Aansluiten bij bekende notatie

De wiskundige definitie van machtsverheffen kan bijvoorbeeld vrijwel letterlijk als Haskell-functie worden gebruikt:

$$x \uparrow 0 = 1$$
$$x \uparrow n = x * x \uparrow (n - 1)$$

Aansluiten bij bekende notatie

De wiskundige definitie van machtsverheffen kan bijvoorbeeld vrijwel letterlijk als Haskell-functie worden gebruikt:

$$x \uparrow 0 = 1$$
$$x \uparrow n = x * x \uparrow (n - 1)$$

Een recursieve definitie waarin voor het gevalsonderscheid patronen worden gebruikt (in plaats van Boolese expressies) wordt daarom ook wel een inductieve definitie genoemd.

Alternatieve schrijfwijzen

We kunnen schrijven:

$$
\begin{aligned}
som\ lijst \mid null\ lijst &= 0 \\
\mid otherwise &= head\ lijst + som\ (tail\ lijst)
\end{aligned}
$$

Alternatieve schrijfwijzen

We kunnen schrijven:

$$\begin{aligned} som\ lijst \mid null\ lijst &= 0 \\ \mid otherwise &= head\ lijst + som\ (tail\ lijst) \end{aligned}$$

Maar meestal scrhijven we:

$$\begin{aligned} som\ [\,] &= 0 \\ som\ (kop : staart) &= kop + som\ staart \end{aligned}$$

omdat dit duidelijker is.

Anonieme waarden

De standaardfunctie *length*, die het aantal elementen in een lijst bepaalt, kan ook inductief worden gedefinieerd:

$$\begin{aligned} length\ [\,] &= 0 \\ length\ (kop : staart) &= 1 + length\ staart \end{aligned}$$

Anonieme waarden

D
b

Op deze manier wordt ervoor gezorgd dat iets wat je niet wilt gebruiken, ook niet kunt gebruiken

$$\begin{aligned} length\ [\,] &= 0 \\ length\ (kop : staart) &= 1 + length\ staart \end{aligned}$$

In patronen is het toegestaan om in dit soort gevallen het teken _ te gebruiken in plaats van een naam:

$$\begin{aligned} length\ [\,] &= 0 \\ length\ (\_ : staart) &= 1 + length\ staart \end{aligned}$$

Layout!!

Anders dan in andere programmeertalen is een regelovergang niet helemaal zonder betekenis. Bekijk bijvoorbeeld de volgende twee **where**-constructies:

where

$$\begin{aligned} v &= f\ x\ y \\ w &= g\ z \end{aligned}$$

where

$$\begin{aligned} v &= f\ x \\ y\ w &= g\ z \end{aligned}$$

Layout!!

Anders dan in andere programmeertalen is een regelovergang niet helemaal zonder betekenis. Bekijk bijvoorbeeld de volgende twee **where**-constructies:

where

$$v = f\ x\ y$$
$$w = g\ z$$

where

$$v = f\ x$$
$$y\ w = g\ z$$

De plaats van de regelovergang (tussen x en y, of tussen y en w) maakt nogal wat uit.

Layout regels (zo ongeveer)

In een rij definities gebruikt Haskell de volgende methode om te bepalen wat bij elkaar hoort:

- een regel die precies even ver is ingesprongen als de vorige, wordt als nieuwe definitie beschouwd;
- is de regel verder ingesprongen, dan hoort hij bij de vorige regel;
- is de regel minder ver ingesprongen, dan hoort hij niet meer bij de huidige lijst definities.

Layout regels (zo ongeveer)

In een rij definities gebruikt Haskell de volgende methode om te bepalen wat bij elkaar hoort:

- een regel die precies even ver is ingesprongen als de vorige, wordt als nieuwe definitie beschouwd;
- is de regel verder ingesprongen, dan hoort hij bij de vorige regel;
- is de regel minder ver ingesprongen, dan hoort hij niet meer bij de huidige lijst definities.

Dat laatste is van belang als een **where**-constructie binnen een andere **where**-constructie voorkomt. Bijvoorbeeld in

$$
\begin{aligned}
f\ x\ y &= g\ (x + w) \\
&\quad \textbf{where}\ g\ u = u + v \\
&\qquad\qquad \textbf{where}\ v = u * u \\
&\qquad w = \ \ 2 + y
\end{aligned}
$$

In de praktijk volstaat meestal:

In de praktijk gaat alles vanzelf goed als je één ding in het oog houdt:

> *gelijkwaardige definities moeten even ver worden ingesprongen*

Dit betekent ook dat alle globale functiedefinities even ver moeten worden ingesprongen (bijvoorbeeld allemaal nul posities).

Commentaar

Er zijn in Haskell twee soorten commentaar:

- met de symbolen `--` begint commentaar dat tot het eind van de regel doorloopt;
- met de symbolen `{-` begint commentaar dat doorloopt tot de symbolen `-}`.

Commentaar

Er zijn in Haskell twee soorten commentaar:

- met de symbolen `--` begint commentaar dat tot het eind van de regel doorloopt;
- met de symbolen `{-` begint commentaar dat doorloopt tot de symbolen `-}`.

Uitzondering op de eerste regel is het geval waarbij `--` deel uitmaakt van een operator, bijvoorbeeld `<-->`. Een losse `--` kan echter geen operator zijn.

Typering en type inferentie

Als een functie-definitie niet aan de vorm-eisen voldoet, krijg je daarvan een melding zodra deze functie geanalyseerd wordt:

$$isNul\ x = x = 0$$

Ziet iemand een fout?

Typering en type inferentie

Als een functie-definitie niet aan de vorm-eisen voldoet, krijg je daarvan een melding zodra deze functie geanalyseerd wordt:

$$isNul\ x = x = 0$$

Ziet iemand een fout? Bij de analyse van deze functie meldt de ghci interpreter:

```
Prelude> let isNul x = x = 0
interactive:1:16: parse error on input '='
```

Nog meer type pouten

Andere typerings-fouten treden bijvoorbeeld op bij het toepassen van de functie *length* op iets anders dan een lijst, zoals in *length* 3:

```
Prelude> length 3
interactive:1:7:
    No instance for (Num [a])
      arising from the literal '3' at <interactive>:1:
    Possible fix: add an instance declaration for (Num
    In the first argument of 'length', namely '3'
    In the expression: length 3
    In the definition of 'it': it = length 3
```

Opvragen van het type van een expressie

Het type van een expressie kan bepaald worden met de interpreter-opdracht `:t` (afkorting van 'type'). Achter `:t` staat de expressie die getypeerd moet worden. Bijvoorbeeld:

```
?  :t True && False
True && False :: Bool
```

Het symbool `::` kan gelezen worden als 'heeft het type'. De expressie wordt met de `:type`-opdracht niet uitgerekend; alleen het type wordt bepaald.

Basis types

Er zijn aantal basis-types:

- *Int*: het type van de gehele getallen (integer numbers of integers), tot een maximum van ruim 2 miljard;
- *Integer*: het type van gehele getallen, zonder praktische begrenzing
- *Float*: het type van de floating-point getallen;
- *Bool*: het type van de Boolese waardes *True* en *False*;
- *Char*: het type van letters, cijfers en symbolen op het toetsenbord (*characters*)

Lijsten kunnen verschillende types hebben

```
? :t ['a', 'b', 'c']
['a','b','c'] :: [Char]
? :t [True,False]
[True,False] :: [Bool]
? :t [ [1,2], [3,4,5] ]
[[1,2],[3,4,5]] :: [[Int]]
```

Lijsten kunnen verschillende types hebben

```
? :t ['a', 'b', 'c']
['a','b','c'] :: [Char]
? :t [True,False]
[True,False] :: [Bool]
? :t [ [1,2], [3,4,5] ]
[[1,2],[3,4,5]] :: [[Int]]
```

Alle waarden in een lijst moeten hetzelfde type hebben.

Zo niet, dan verschijnt er een melding van een typerings-fout:

```
interactive:1:1:
    No instance for (Num Bool)
      arising from the literal ‘2’ at interactive:1:1
    Possible fix: add an instance declaration for (Num
    In the expression: 2
    In the expression: [2, True]
```

Ook functies hebben een type

Het type van een functie wordt bepaald door het type van de parameter en het type van het resultaat:

```
? :t sum
sum :: [Int] -> Int
```

Ook functies hebben een type

Het type van een functie wordt bepaald door het type van de parameter en het type van het resultaat:

```
? :t sum
sum :: [Int] -> Int
```

De functie *sum* neemt als argument een lijst van integer waarden ($[Int]$) en levert een enkele integer op.

Alles kan!

Omdat functies (net als getallen, Boolese waarden en lijsten) een type hebben, is het mogelijk om functies in een lijst op te nemen:

```
? :t [sin,cos,tan]
[sin,cos,tan] :: [Float -> Float]
```

Expliciet types opgeven

Een functiedefinitie met een expliciet opgegeven type:

$$
\begin{aligned}
sum\ &:: [Int] \rightarrow Int \\
sum\ [\,]\ &= 0 \\
sum\ (x : xs)\ &= x + sum\ xs
\end{aligned}
$$

Voordelen:

- er wordt gecontroleerd of de functie inderdaad het type heeft dat je ervoor hebt gedeclareerd;
- de declaratie maakt het voor een menselijke lezer eenvoudiger om een functie te begrijpen.
- als je alle types aan het begin zet heb je een soort inhoudsopgave

Polymorfie!

Voor sommige functies op lijsten maakt het niet uit wat het type van de elementen van die lijst is.

Het type van de functie *length* wordt dan ook als volgt genoteerd:

$$length :: [a] \rightarrow Int$$

Polymorfie!

Voor sommige functies op lijsten maakt het niet uit wat het type van de elementen van die lijst is.

Het type van de functie *length* wordt dan ook als volgt genoteerd:

$$length :: [a] \rightarrow Int$$

Het type van deze elementen wordt aangegeven door een *typevariabele*, in het voorbeeld a. Typevariabelen worden, in tegenstelling tot de vaste types als Int en $Bool$, **met een kleine letter** geschreven.

Voorbeelden

$$head :: [a] \rightarrow a$$
$$tail :: [a] \rightarrow [a]$$

Voorbeelden

$$head :: [a] \to a$$
$$tail :: [a] \to [a]$$

Polymorfe functies, zoals *length* en *head*, hebben met elkaar gemeen dat ze alleen de structuur van de lijst gebruiken. Een niet-polymorfe functie, zoals *sum*, gebruikt ook eigenschappen van de elementen van de lijst, zoals 'optelbaarheid'.

Voorbeelden

$$head :: [a] \rightarrow a$$
$$tail :: [a] \rightarrow [a]$$

Polymorfe functies, zoals *length* en *head*, hebben met elkaar gemeen dat ze alleen de structuur van de lijst gebruiken. Een niet-polymorfe functie, zoals *sum*, gebruikt ook eigenschappen van de elementen van de lijst, zoals 'optelbaarheid'. In meer theoretisch verhalen schrijft men ook wel $forall\ a \circ [a] \rightarrow a$, en maakt daarmee expliciet dat a vrij gekozen kan worden.

Exploiteer Polymorfie

Niet alleen functies op lijsten kunnen polymorf zijn. De eenvoudigste polymorfe functie is de identiteits-functie (de functie die zijn parameter onveranderd oplevert):

$$
\begin{aligned}
id &:: a \to a \\
id\ x &= x
\end{aligned}
$$

De functie id kan op elementen van willekeurig type werken (en het resultaat is dan van hetzelfde type).

Exploiteer Polymorfie

Niet alleen functies op lijsten kunnen polymorf zijn. De eenvoudigste polymorfe functie is de identiteits-functie (de functie die zijn parameter onveranderd oplevert):

$$
\begin{aligned}
id &:: a \to a \\
id\ x &= x
\end{aligned}
$$

De functie id kan op elementen van willekeurig type werken (en het resultaat is dan van hetzelfde type). De functie kan zelfs worden toegepast als in $id\ sqrt$, of op functies van lijsten van integers naar integers: $id\ sum$.

Testje!

Wat denk je van:

> $id\ id$
> $(id\ id)\ (id\ id\ id)$

Testje!

Wat denk je van:

> $id\ id$
> $(id\ id)\ (id\ id\ id)$

Zoals het type al aangeeft kan de functie worden toegepast op parameters van een willekeurig type. De parameter mag dus ook het type $a \to a$ hebben, zodat de functie id ook op zichzelf kan worden toegepast: $id\ id$.

Testje!

Wat denk je van:

> $id\ id$
> $(id\ id)\ (id\ id\ id)$

Zoals het type al aangeeft kan de functie worden toegepast op parameters van een willekeurig type. De parameter mag dus ook het type $a \rightarrow a$ hebben, zodat de functie id ook op zichzelf kan worden toegepast: $id\ id$.
Welke functie is dit?

Meerdere parameters

De functie *abcFormule* heeft drie floating-point getallen als parameter en een lijst met floating-point getallen als resultaat. De type-declaratie luidt daarom:

$$abcFormule :: Float \rightarrow Float \rightarrow Float \rightarrow [Float]$$

Meerdere parameters

De functie *abcFormule* heeft drie floating-point getallen als parameter en een lijst met floating-point getallen als resultaat. De type-declaratie luidt daarom:

$$abcFormule :: Float \rightarrow Float \rightarrow Float \rightarrow [Float]$$

En de functie *map*

$$map :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]$$

De eerste parameter van *map* is een functie tussen willekeurige types (a en b), die niet eens hetzelfde hoeven te zijn. De tweede parameter van *map* is een lijst, waarvan de elementen hetzelfde type (a) moeten hebben als de parameter van de functie-parameter.

Testje

Denk je dat de volgend expressie goed getypeerd is, en zo ja wat is dan het type:

$$[sin, id]$$

Testje

Denk je dat de volgend expressie goed getypeerd is, en zo ja wat is dan het type:

$[sin, id]$

Antwoord:

```
Prelude> :t sin
sin :: Float -> Float
Prelude> :t id
id :: a -> a
Prelude> :t [sin, id]
[sin, id] :: [Float -> Float]
```

Testje

Denk je dat de volgend expressie goed getypeerd is, en zo ja wat is dan het type:

$$[length, head]$$

Testje

Denk je dat de volgend expressie goed getypeerd is, en zo ja wat is dan het type:

> $[length, head]$

Antwoord:

```
Prelude> :t [length,head]
[length,head] :: [[Int] -> Int]
Prelude> :t length
length :: [a] -> Int
Prelude> :t head
head :: [a] -> a
Prelude> :t [length,head]
[length,head] :: [[Int] -> Int]
Prelude>
```

Overloading

Het type van $+$ kan zowel $Int \rightarrow Int \rightarrow Int$ als $Float \rightarrow Float \rightarrow Float$ zijn. Toch is $+$ niet echt een polymorfe operator. Als het type $a \rightarrow a \rightarrow a$ zou zijn, zou de operator namelijk ook op bijvoorbeeld $Bool$ parameters moeten werken, hetgeen niet mogelijk is. Zo'n functie die 'een beetje' polymorf is, wordt een *overloaded* functie genoemd.

Klassen

Om toch een type te kunnen geven aan een overloaded functie of operator, worden types ingedeeld in klassen (*classes*):

- *Num* is de klasse van types waarvan de elementen opgeteld, afgetrokken, vermenigvuldigd en gedeeld kunnen worden (numerieke types);

Klassen

Om toch een type te kunnen geven aan een overloaded functie of operator, worden types ingedeeld in klassen (*classes*):

- *Num* is de klasse van types waarvan de elementen opgeteld, afgetrokken, vermenigvuldigd en gedeeld kunnen worden (numerieke types);
- *Ord* is de klasse van types waarvan de elementen geordend kunnen worden (ordenbare types);

Klassen

Om toch een type te kunnen geven aan een overloaded functie of operator, worden types ingedeeld in klassen (*classes*):

- *Num* is de klasse van types waarvan de elementen opgeteld, afgetrokken, vermenigvuldigd en gedeeld kunnen worden (numerieke types);
- *Ord* is de klasse van types waarvan de elementen geordend kunnen worden (ordenbare types);
- *Eq* is de klasse van types waarvan de elementen met elkaar vergeleken kunnen worden (equality types).

Klassen

Om toch een type te kunnen geven aan een overloaded functie of operator, worden types ingedeeld in klassen (*classes*):

- *Num* is de klasse van types waarvan de elementen opgeteld, afgetrokken, vermenigvuldigd en gedeeld kunnen worden (numerieke types);
- *Ord* is de klasse van types waarvan de elementen geordend kunnen worden (ordenbare types);
- *Eq* is de klasse van types waarvan de elementen met elkaar vergeleken kunnen worden (equality types).
- *Integral* is de klasse van types waarvan de elementen gehele getallen zijn, dus *Int* en *Integer*

Klassen

Om toch een type te kunnen geven aan een overloaded functie of operator, worden types ingedeeld in klassen (*classes*):

- *Num* is de klasse van types waarvan de elementen opgeteld, afgetrokken, vermenigvuldigd en gedeeld kunnen worden (numerieke types);
- *Ord* is de klasse van types waarvan de elementen geordend kunnen worden (ordenbare types);
- *Eq* is de klasse van types waarvan de elementen met elkaar vergeleken kunnen worden (equality types).
- *Integral* is de klasse van types waarvan de elementen gehele getallen zijn, dus *Int* en *Integer*

De operator $+$ heeft nu het volgende type:

$$(+) :: Num\ a \Rightarrow a \rightarrow a \rightarrow a$$

Nog meer operatoren

Andere voorbeelden van overloaded operatoren zijn:

$$(<) :: Ord\ a \Rightarrow a \rightarrow a \rightarrow Bool$$
$$(\equiv) :: Eq\ a \Rightarrow a \rightarrow a \rightarrow Bool$$

Nog meer operatoren

Andere voorbeelden van overloaded operatoren zijn:

$$(<) :: Ord\ a \Rightarrow a \rightarrow a \rightarrow Bool$$
$$(\equiv) :: Eq\ a \Rightarrow a \rightarrow a \rightarrow Bool$$

Zelf-gedefinieerde functies kunnen ook overloaded zijn. Bijvoorbeeld de functie

$$kwadraat\ x = x * x$$

heeft het type

$$kwadraat :: Num\ a \Rightarrow a \rightarrow a$$

doordat de operator $*$ die erin gebruikt wordt overloaded is.

Het moet niet gekker worden

Ook constanten zijn overloaded. Bijvoorbeeld:

```
? :t 3
3 :: Num a => a
```

Dat betekent dat de constante 3 overal gebruikt kan worden waar een numeriek type wordt verwacht. Deze overloading strekt zich uit tot lijsten:

```
:t [1,2,3]
[1,2,3] :: Num a => [a]
```

Overloading versus Polymorfie!!

Polymorfie

Een polymorfe functie is één functie, die je op verschillende manieren kunt gebruiken. Het type systeem *specialiseert* de functie als dat nodig is.

Overloading versus Polymorfie!!

Polymorfie

Een polymorfe functie is één functie, die je op verschillende manieren kunt gebruiken. Het type systeem *specialiseert* de functie als dat nodig is.

Overloading

Er is sprake van overloading als we *verschillende functies dezelfde naam* geven. Het type systeem kiest de juiste definitie op grond van beschikbare type informatie.

Operator sections

- een operator tussen haakjes gedraagt zich als de overeenkomstige functie;

```
Prelude> 2 + 3
5
Prelude> (+) 2 3
5
```

Operator sections

- een operator tussen haakjes gedraagt zich als de overeenkomstige functie;

```
Prelude> 2 + 3
5
Prelude> (+) 2 3
5
```

- een functie tussen *back quotes* gedraagt zich als de overeenkomstige operator.

```
Prelude> div 8 4
2
Prelude> 8 `div` 4
2
```

Prioriteiten

In totaal zijn er negen nivo's van prioriteiten:

nivo 9	$\circ$ *(.) en* !!
nivo 8	$\uparrow$
nivo 7	$*$, /, *'div'*, *'rem' en 'mod'*
nivo 6	$+$ *en* $-$
nivo 5	:, $+\!\!+$ *en* $\backslash\backslash$
nivo 4	$\equiv$, $\not\equiv$,$<$, $\leqslant$, $>$, $\geqslant$, $\in$ (*'*`elem`*'*) *en* $\notin$ (*'*`notElem`*'*)
nivo 3	$\wedge$
nivo 2	$\|\|$
nivo 1	*(niet gebruikt in de prelude)*

Associativiteit

- de operator $\oplus$ *associeert naar links*, dat wil zeggen $a \oplus b \oplus c$ is equivalent aan $(a \oplus b) \oplus c$;
- de operator $\oplus$ *associeert naar rechts*, dat wil zeggen $a \oplus b \oplus c$ is equivalent aan $a \oplus (b \oplus c)$;
- de operator $\oplus$ is *associatief*, dat wil zeggen het maakt niet uit in welke volgorde $a \oplus b \oplus c$ wordt uitgerekend;
- de operator $\oplus$ is *non-associatief*, dat wil zeggen dat het verboden is om $a \oplus b \oplus c$ te schrijven; je moet dus altijd met haakjes aangeven wat de bedoeling is.

Linksassociatief

De volgende operatoren associëren naar **links**:

- de 'onzichtbare' operator *functie-applicatie*, dus $f\ x\ y$ moet gelezen worden als $(f\ x)\ y$
- de operator !!
- de operator $-$, dus de waarde van $8 - 5 - 1$ is 2
- de operator $/$ en de verwante operatoren div, rem en mod.

Rechtsassociatief

De volgende operatoren associëren naar **rechts**:

- de operator $\uparrow$ (machtsverheffen), dus de waarde van $2 \uparrow 2 \uparrow 3$ is $2^8 = 256$ en niet $4^3 = 64$;
- de operator : ('zet op kop van'), zodat de waarde van $1 : 2 : 3 : x$ een lijst is die begint met de waarden 1, 2 en 3.

Niet associerend

De volgende operatoren zijn **non**-associatief:

- de operator $\backslash\backslash$;
- de vergelijkings-operatoren $\equiv$, $<$ enzovoort: het heeft meestal toch geen zin om $x \equiv y \equiv z$ te schrijven. Probeer je dat toch dan krijg je de volgende melding:

```
Fail: ambiguous use
      of   non-associative operator == at (1,13)
      with non-associative operator == at (1,16)
```

Niet associerend

De volgende operatoren zijn **non**-associatief:

- de operator $\backslash\backslash$;
- de vergelijkings-operatoren $\equiv$, $<$ enzovoort: het heeft meestal toch geen zin om $x \equiv y \equiv z$ te schrijven. Probeer je dat toch dan krijg je de volgende melding:

```
Fail: ambiguous use
      of   non-associative operator == at (1,13)
      with non-associative operator == at (1,16)
```

Dus niet $2 < x < 8$ maar $2 < x \wedge x < 8$.

Associatief

De volgende operatoren zijn (vaak) associatief:

- de operatoren $*$ en $+$ (deze operatoren worden overeenkomstig wiskundige traditie links-associërend uitgerekend);
- de operatoren $+\!\!+$, $\wedge$ en $\vee$ (deze operatoren worden rechts-associërend uitgerekend omdat dat iets efficiënter is);
- de operator voor functiecompositie: $\circ$.

Prioriteitsdefinities

Machtsverheffen is rechts-associatief (**infixr**):

infixr $8\uparrow$

Prioriteitsdefinities

Machtsverheffen is rechts-associatief (**infixr**):

> **infixr** $8 \uparrow$

Voor operatoren die naar links associëren dient het gereserveerde woord **infixl**, en voor non-associatieve operatoren het woord **infix**:

> **infixl** $6 +, -$
> **infix** $4 \equiv, \not\equiv, \in$

Prioriteitsdefinities

Machtsverheffen is rechts-associatief (**infixr**):

> **infixr** $8\uparrow$

Voor operatoren die naar links associëren dient het gereserveerde woord **infixl**, en voor non-associatieve operatoren het woord **infix**:

> **infixl** $6+, -$
> **infix** $4\equiv, \not\equiv, \in$

En je kunt bijvoorbeeld zelf definieren::

> **infix** 5!^!,‘*boven*‘

zodat je $a+b$!^! $c+d$ kan schrijven.

Currying

In Haskell mag je ook minder argumenten aan een functie meegeven. Als *plus* maar één argument krijgt, bijvoorbeeld zoals in *plus* 1, dan houdt je een functie over die nog een argument verwacht:

$$
\begin{aligned}
&\mathit{plus} &&:: \mathit{Int} \to \mathit{Int} \to \mathit{Int} \\
&\mathit{opvolger} &&:: \mathit{Int} \to \mathit{Int} \\
&\mathit{opvolger} &&= \mathit{plus}\ 1
\end{aligned}
$$

Currying

In Haskell mag je ook minder argumenten aan een functie meegeven. Als *plus* maar één argument krijgt, bijvoorbeeld zoals in *plus* 1, dan houdt je een functie over die nog een argument verwacht:

$$
\begin{aligned}
&\mathit{plus} &&:: \mathit{Int}\rightarrow\mathit{Int}\rightarrow\mathit{Int}\\
&\mathit{opvolger} &&:: \mathit{Int}\rightarrow\mathit{Int}\\
&\mathit{opvolger} &&= \mathit{plus}\ 1
\end{aligned}
$$

Een tweede toepassing van een partieel geparametriseerde functie is dat deze als argument kan dienen voor een andere functie.

```
? map (plus 5) [1,2,3]
[6, 7, 8]
```

Currying

Dus *plus* 1 is 'de functie die ergens 5 bij optelt'.

In Haskell mag je ook minder argumenten aan een functie meegeven. Als *plus* maar één argument krijgt, bijvoorbeeld zoals in *plus* 1, dan houdt je een functie over die nog een argument verwacht:

$$
\begin{aligned}
&\mathit{plus} &&:: \mathit{Int} \to \mathit{Int} \to \mathit{Int} \\
&\mathit{opvolger} &&:: \mathit{Int} \to \mathit{Int} \\
&\mathit{opvolger} &&= \mathit{plus}\ 1
\end{aligned}
$$

Een tweede toepassing van een partieel geparametriseerde functie is dat deze als argument kan dienen voor een andere functie.

```
? map (plus 5) [1,2,3]
[6, 7, 8]
```

Functie applicatie: nogmaals

Staan er in een expressie een hele rij letters op een rij, dan moet de eerste daarvan een functie zijn die de andere achtereenvolgens als parameter opneemt:

$$f\ k\ l\ VaIdm\ n$$

wordt opgevat als

$$((((f\ k)\ l)\ VaIdm)\ n)$$

Functie applicatie: nogmaals

Staan er in een expressie een hele rij letters op een rij, dan moet de eerste daarvan een functie zijn die de andere achtereenvolgens als parameter opneemt:

$$f\ k\ l\ ValIdm\ n$$

wordt opgevat als

$$((((f\ k)\ l)\ ValIdm)\ n)$$

Als k type K heeft, l type L enzovoort, en f uiteindelijk iets van type O oplevert, dan is het type van f:

$$f :: K \to L \to M \to N \to O$$

Functie applicatie: nogmaals

Staan er in een expressie een hele rij letters op een rij, dan moet de eerste daarvan een functie zijn die de andere achtereenvolgens als parameter opneemt:

$$f\ k\ l\ VaIdm\ n$$

wordt opgevat als

$$((((f\ k)\ l)\ VaIdm)\ n)$$

Als k type K heeft, l type L enzovoort, en f uiteindelijk iets van type O oplevert, dan is het type van f:

$$f :: K \to L \to M \to N \to O$$

of, als je alle haakjes zou schrijven:

$$f :: K \to (L \to (M \to (N \to O)))$$

Operator-secties

Voor het partieel parametriseren van operatoren zijn twee speciale notaties beschikbaar:

- met $(\oplus x)$ wordt de operator $\oplus$ partieel geparametriseerd met x als *rechter* parameter;
- met $(x \oplus)$ wordt de operator $\oplus$ partieel geparametriseerd met x als *linker* parameter.

Voorbeelden van secties

Met operator-secties kunnen een aantal functies gedefinieerd worden:

$$
\begin{aligned}
\mathit{opvolger} &= (+1) \\
\mathit{verdubbel} &= (2*) \\
\mathit{halveer} &= (/2.0) \\
\mathit{omgekeerde} &= (1.0/) \\
\mathit{kwadraat} &= (\uparrow 2) \\
\mathit{tweeTotDe} &= (2\uparrow) \\
\mathit{eencijferig} &= (\leqslant 9) \\
\mathit{isNul} &= (\equiv 0)
\end{aligned}
$$

Voorbeelden van secties

Met operator-secties kunnen een aantal functies gedefinieerd worden:

$$
\begin{aligned}
\mathit{opvolger} &= (+1) \\
\mathit{verdubbel} &= (2*) \\
\mathit{halveer} &= (/2.0) \\
\mathit{omgekeerde} &= (1.0/) \\
\mathit{kwadraat} &= (\uparrow 2) \\
\mathit{tweeTotDe} &= (2\uparrow) \\
\mathit{eencijferig} &= (\leqslant 9) \\
\mathit{isNul} &= (\equiv 0)
\end{aligned}
$$

Een subtiliteit is dat (-2) gewoon de waarde -2 is, dus:

$$
\begin{aligned}
\mathit{subtract}\ x\ y &= y - x \\
\mathit{trek\_twee\_af} &= (2\text{`}\mathit{subtract}\text{`})
\end{aligned}
$$

Het gebruik van secties

De belangrijkste toepassing van operator-secties is het meegeven van zo'n partieel geparametriseerde operator aan een functie:

```
? map (2*) [1,2,3]
[2, 4, 6]
```

Case study: De zeef van Erathostenes

Een beroemd algoritme, toegeschreven aan Erathostenes, berekent de priemgetallen:

1. neem een lijst van alle getallen: $[2..]$.

Case study: De zeef van Erathostenes

Een beroemd algoritme, toegeschreven aan Erathostenes, berekent de priemgetallen:

1. neem een lijst van alle getallen: $[2..]$.
2. streep hier alle 2-vouden uit weg, en onthoudt dat 2 een priemgetal is.

Case study: De zeef van Erathostenes

Een beroemd algoritme, toegeschreven aan Erathostenes, berekent de priemgetallen:

1. neem een lijst van alle getallen: $[2..]$.
2. streep hier alle 2-vouden uit weg, en onthoudt dat 2 een priemgetal is.
3. het kleinste getal dat nu nog voorkomt is 3, dus streep alle 3-tallen weg en onthoudt dat 3 een priemgetal is.

Case study: De zeef van Erathostenes

Een beroemd algoritme, toegeschreven aan Erathostenes, berekent de priemgetallen:

1. neem een lijst van alle getallen: $[2..]$.
2. streep hier alle 2-vouden uit weg, en onthoudt dat 2 een priemgetal is.
3. het kleinste getal dat nu nog voorkomt is 3, dus streep alle 3-tallen weg en onhoudt dat 3 een priemgetal is.
4. het kleinste getal dat nu nog voorkomt is 5, dus ...

Verder met zeven

$$zeef\ n\ lijst = filter\ ((\not\equiv 0) \circ (`mod` n))\ lijst$$

Verder met zeven

$$zeef\ n\ lijst = filter\ ((\not\equiv 0) \circ (`mod` n))\ lijst$$

En nu herhaaldelijk toepassen, en priemgetallen doorlaten:

$$zeven\ (x : xs) = x : zeven\ (zeef\ x\ xs)$$

Verder met zeven

$$zeef\ n\ lijst = filter\ ((\not\equiv 0) \circ (`mod` n))\ lijst$$

En nu herhaaldelijk toepassen, en priemgetallen doorlaten:

$$zeven\ (x : xs) = x : zeven\ (zeef\ x\ xs)$$

En nu maar aan het werk op alle kandidaten:

$$priemgetallen = zeven\ [2..]$$

Verder met zeven

$$zeef\ n\ lijst = filter\ ((\not\equiv 0) \circ (`mod` n))\ lijst$$

En nu herhaaldelijk toepassen, en priemgetallen doorlaten:

$$zeven\ (x : xs) = x : zeven\ (zeef\ x\ xs)$$

En nu maar aan het werk op alle kandidaten:

$$priemgetallen = zeven\ [2..]$$

```
Programs> take 4 priemgetallen
[2,3,5,7]
```