Universiteit Utrecht

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

College 2010-2011
3. Lijstfuncties en Basic IO

Doaitse Swierstra

Utrecht University

September 21, 2010

Functies hebben alle burgerrechten!

Functies zijn first-class citizens want:

- functies hebben een type

Functies hebben alle burgerrechten!

Functies zijn first-class citizens want:

- functies hebben een type
- functies kunnen door andere functies worden opgeleverd als resultaat (waarvan met Currying veel gebruik wordt gemaakt);

Functies hebben alle burgerrechten!

Functies zijn first-class citizens want:

- functies hebben een type
- functies kunnen door andere functies worden opgeleverd als resultaat (waarvan met Currying veel gebruik wordt gemaakt);
- functies kunnen als argument aan andere functies worden meegegeven.

map

Definition (*map*)

$$
\begin{array}{lll}
map & :: & (a \to b) \to [a] \to [b] \\
map\ f\ [\,] & = & [\,] \\
map\ f\ (x : xs) & = & f\ x : map\ f\ xs
\end{array}
$$

filter

Voorbeelden van het gebruik van *filter* zijn:

```
? filter even [1..10]
[2, 4, 6, 8, 10]
? filter (>10) [2,17,8,12,5]
[17, 12]
```

filter

Voorbeelden van het gebruik van *filter* zijn:

```
? filter even [1..10]
[2, 4, 6, 8, 10]
? filter (>10) [2,17,8,12,5]
[17, 12]
```

Definition (*filter*)

$$
\begin{array}{lll}
filter & :: & (a \to Bool) \to [a] \to [a] \\
filter\ p\ [\,] & = & [\,] \\
filter\ p\ (x : xs) \mid p\ x & = & x : filter\ p\ xs \\
\quad \mid otherwise & = & \quad filter\ p\ xs
\end{array}
$$

filter

Voorbeelden van het gebruik van *filter* zijn:

```
? filter even [1..10]
[2, 4, 6, 8, 10]
? filter (>10) [2,17,8,12,5]
[17, 12]
```

Definition (*filter*)

$$
\begin{aligned}
&\mathit{filter} \quad :: \ (a \to \mathit{Bool}) \to [a] \to [a] \\
&\mathit{filter}\ p\ [\,] = [\,] \\
&\mathit{filter}\ p\ (x : xs) = (\textbf{if}\ p\ x\ \textbf{then}\ (x:)\ \textbf{else}\ \mathit{id})\ (\mathit{filter}\ p\ xs)
\end{aligned}
$$

Veel functie lijken erg op elkaar qua structuur

$$
\begin{array}{lll}
sum & [\,] & = 0 \\
sum & (x : xs) & = x \; + \; sum \; xs \\
product & [\,] & = 1 \\
product & (x : xs) & = x \; * \; product \; xs \\
and & [\,] & = True \\
and & (x : xs) & = x \; \wedge \; and \; xs
\end{array}
$$

Zoek de verschillen en de overeenkomsten!

Veel functie lijken erg op elkaar qua structuur

Zoek de verschillen en de overeenkomsten!

$$
\begin{array}{lll}
sum & [\,] & = 0 \\
sum & (x : xs) & = x + sum\ xs
\end{array}
$$

Veel functie lijken erg op elkaar qua structuur

Zoek de verschillen en de overeenkomsten!

$$
\begin{array}{lll}
sum & [\,] & = 0 \\
sum & (x : xs) & = x + sum\ xs \\
product & [\,] & = 1 \\
product & (x : xs) & = x * product\ xs
\end{array}
$$

Veel functie lijken erg op elkaar qua structuur

Zoek de verschillen en de overeenkomsten!

$$
\begin{array}{lll}
sum & [\,] & = 0 \\
sum & (x : xs) & = x + sum\ xs \\
product & [\,] & = 1 \\
product & (x : xs) & = x * product\ xs \\
and & [\,] & = True \\
and & (x : xs) & = x \wedge and\ xs
\end{array}
$$

Veel functie lijken erg op elkaar qua structuur

Zoek de verschillen en de overeenkomsten!

$$
\begin{array}{lll}
sum & [\,] & = 0 \\
sum & (x : xs) & = x + sum\ xs \\
product & [\,] & = 1 \\
product & (x : xs) & = x * product\ xs \\
and & [\,] & = True \\
and & (x : xs) & = x \wedge and\ xs
\end{array}
$$

Veel functie lijken erg op elkaar qua structuur

Zoek de verschillen en de overeenkomsten!

$$
\begin{array}{lll}
sum & [\,] & = 0 \\
sum & (x : xs) & = x + sum\ xs \\
product & [\,] & = 1 \\
product & (x : xs) & = x * product\ xs \\
and & [\,] & = True \\
and & (x : xs) & = x \wedge and\ xs
\end{array}
$$

Door te abstraheren kunnen we de gemeenschappelijk zaken uitfactoriseren. De verschillen drukken we dan uit middels verschillende argumenten.

foldr

Definition (*foldr*)

$$
\begin{aligned}
&\mathit{foldr} :: (a \to b \to b) \to b \to [a] \to b \\
&\mathit{foldr}\ \mathit{op}\ e\ [\,] \quad\ = e \\
&\mathit{foldr}\ \mathit{op}\ e\ (x : xs) = x\ `\mathit{op}`\ \mathit{foldr}\ \mathit{op}\ e\ xs
\end{aligned}
$$

foldr

Definition (*foldr*)

$$
\begin{aligned}
&\mathit{foldr} :: (a \to b \to b) \to b \to [a] \to b \\
&\mathit{foldr}\ \mathit{op}\ e\ [\,] \quad\ = e \\
&\mathit{foldr}\ \mathit{op}\ e\ (x : xs) = x\ `\mathit{op}`\ \mathit{foldr}\ \mathit{op}\ e\ xs
\end{aligned}
$$

En nu:

$$
\begin{aligned}
\mathit{sum} &= \mathit{foldr}\ (+)\ 0 \\
\mathit{product} &= \mathit{foldr}\ (*)\ 1 \\
\mathit{and} &= \mathit{foldr}\ (\wedge)\ \mathit{True}
\end{aligned}
$$

foldr

Definition (*foldr*)

$$
\begin{aligned}
&\mathit{foldr} :: (a \to b \to b) \to b \to [a] \to b \\
&\mathit{foldr}\ \mathit{op}\ e\ [\,] \quad\ = e \\
&\mathit{foldr}\ \mathit{op}\ e\ (x : xs) = x\ `\mathit{op}`\ \mathit{foldr}\ \mathit{op}\ e\ xs
\end{aligned}
$$

En nu:

$$
\begin{aligned}
\mathit{sum} &= \mathit{foldr}\ (+)\ 0 \\
\mathit{product} &= \mathit{foldr}\ (*)\ 1 \\
\mathit{and} &= \mathit{foldr}\ (\wedge)\ \mathit{True}
\end{aligned}
$$

De functie *foldr* is in veel meer gevallen bruikbaar; daarom is hij als standaardfunctie in de prelude gedefinieerd.

Uitleg van *foldr*

De waarde van

$$foldr\ (+)\ e\ (w : (x : (y : (z : [\,]))))$$

is gelijk aan de waarde van de expressie

$$(w + (x + (y + (z + e))))$$

Uitleg van *foldr*

De waarde van

$$\mathit{foldr}\ (+)\ e\ (w : (x : (y : (z : [\,]))))$$

is gelijk aan de waarde van de expressie

$$(w + (x + (y + (z + e))))$$

De functie *foldr* 'vouwt' de lijst ineen tot één waarde:

- door alle "op-kop-van" voorkomens te vervangen door de gegeven operator (hier $(+)$),

Uitleg van *foldr*

De waarde van

$$\mathit{foldr}\ (+)\ e\ (w : (x : (y : (z : [\,]))))$$

is gelijk aan de waarde van de expressie

$$(w + (x + (y + (z + e))))$$

De functie *foldr* 'vouwt' de lijst ineen tot één waarde:

- door alle "op-kop-van" voorkomens te vervangen door de gegeven operator (hier $(+)$),
- en de lege lijst ($[\,]$) die helemaal aan de rechter kant staat te vervangen door de startwaarde (hier e).

Uitleg van *foldr*

De waarde van

$$\mathit{foldr}\ (+)\ e\ (w : (x : (y : (z : [\,]))))$$

is gelijk aan de waarde van de expressie

$$(w + (x + (y + (z + e))))$$

De functie *foldr* 'vouwt' de lijst ineen tot één waarde:

- door alle "op-kop-van" voorkomens te vervangen door de gegeven operator (hier $(+)$),
- en de lege lijst ($[\,]$) die helemaal aan de rechter kant staat te vervangen door de startwaarde (hier e).

Uitleg van *foldr*

Er bestaat ook een functie *foldl* die aan de linkerkant begint!

De waarde van

$$foldr\ (+)\ e\ (w : (x : (y : (z : [\,]))))$$

is gelijk aan de waarde van de expressie

$$(w + (x + (y + (z + e))))$$

De functie *foldr* 'vouwt' de lijst ineen tot één waarde:

- door alle "op-kop-van" voorkomens te vervangen door de gegeven operator (hier $(+)$),
- en de lege lijst ($[\,]$) die helemaal aan de rechter kant staat te vervangen door de startwaarde (hier e).

Testje

Kun je *filter* ook schrijven m.b.v. *foldr*? Dus:

$$filter\ p\ l = foldr\ op\ ?\ e\ ?\ \ l$$

Testje

Kun je *filter* ook schrijven m.b.v. *foldr*? Dus:

$$filter\ p\ l = foldr\ op\ ?\ e\ ?\ \ l$$

Bekijk altijd eerste het lege geval!!

$$\begin{aligned} filter\ p\ [\,] &= foldr\ op\ e\ [\,] \\ [\,] &= e \end{aligned}$$

Dus we kiezen voor e de lege lijst $[\,]$.

Testje

Kun je *filter* ook schrijven m.b.v. *foldr*? Dus:

$$filter\ p\ l = foldr\ op\ ?\ e\ ?\ \ l$$

Nu:

$$filter\ p\ (x : xs) = foldr\ op\ [\,]\ (x : xs)$$
$$\textbf{if}\ p\ x\ \textbf{then}\ x : filter\ p\ xs\ \textbf{else}\ filter\ p\ xs = x\ `op`\ foldr\ op\ e\ xs$$

Testje

Kun je *filter* ook schrijven m.b.v. *foldr*? Dus:

$$filter\ p\ l = foldr\ op\ ?\ e\ ?\ \ l$$

Nu:

$$filter\ p\ (x : xs) = foldr\ op\ [\,]\ (x : xs)$$
$$\textbf{if}\ p\ x\ \textbf{then}\ x : filter\ p\ xs\ \textbf{else}\ filter\ p\ xs = x\ `op`\ foldr\ op\ e\ xs$$

We herschijven dit een beetje:

$$filter\ p\ (x : xs) = \textbf{if}\ p\ x\ \textbf{then}\ x : rest$$
$$\textbf{else}\ rest$$
$$\textbf{where}\ rest = filter\ p\ xs$$

Testje

Kun je *filter* ook schrijven m.b.v. *foldr*? Dus:

$$filter\ p\ l = foldr\ op\ ?\ e\ ?\ l$$

We herschijven dit een beetje:

$$filter\ p\ (x : xs) = \textbf{if}\ p\ x\ \textbf{then}\ x : rest\ \textbf{else}\ rest\ \textbf{where}\ rest = filter\ p\ xs$$

En "vinden nu de operator *op* uit":

$$filter\ p\ (x : xs) = x\ `op`\ rest\ \textbf{where}\ x\ `op`\ rest = \textbf{if}\ p\ x\ \textbf{then}\ x : rest\ \textbf{else}\ rest$$
$$rest = filter\ p\ xs$$

Testje

Kun je *filter* ook schrijven m.b.v. *foldr*? Dus:

$$filter\ p\ l = foldr\ op\ ?\ e\ ?\ \ l$$

En "vinden nu de operator *op* uit":

$$filter\ p\ (x : xs) = x\ `op`\ rest$$
$$\textbf{where}\ x\ `op`\ rest = \textbf{if}\ p\ x\ \textbf{then}\ x : rest\ \textbf{else}\ rest$$
$$rest = filter\ p\ xs$$

Definition (*filter* m.b.v. *foldr*)

$$filter\ p\ l = foldr\ op\ [\,]\ l$$
$$\textbf{where}\ x\ `op`\ rest = \textbf{if}\ p\ x\ \textbf{then}\ x : rest\ \textbf{else}\ rest$$

Iteratie

Iteratie is goed te beschrijven met een hogere-orde functie. De functie *until* heeft als type:

$$until :: (a \to Bool) \to (a \to a) \to a \to a$$

Iteratie

Iteratie is goed te beschrijven met een hogere-orde functie. De functie *until* heeft als type:

$$until :: (a \to Bool) \to (a \to a) \to a \to a$$

De parameters van *until* zijn

1. de eigenschap waar het eindresultaat aan moet voldoen (een functie van type $a \to Bool$),

Iteratie

Iteratie is goed te beschrijven met een hogere-orde functie. De functie *until* heeft als type:

$$until :: (a \to Bool) \to (a \to a) \to a \to a$$

De parameters van *until* zijn

1. de eigenschap waar het eindresultaat aan moet voldoen (een functie van type $a \to Bool$),
2. de functie die steeds wordt toegepast (een functie van type $a \to a$), en

Iteratie

Iteratie is goed te beschrijven met een hogere-orde functie. De functie *until* heeft als type:

$$until :: (a \to Bool) \to (a \to a) \to a \to a$$

De parameters van *until* zijn

1. de eigenschap waar het eindresultaat aan moet voldoen (een functie van type $a \to Bool$),
2. de functie die steeds wordt toegepast (een functie van type $a \to a$), en
3. een startwaarde van type a

Iteratie

Iteratie is goed te beschrijven met een hogere-orde functie. De functie *until* heeft als type:

> $until :: (a \to Bool) \to (a \to a) \to a \to a$

De parameters van *until* zijn

1. de eigenschap waar het eindresultaat aan moet voldoen (een functie van type $a \to Bool$),
2. de functie die steeds wordt toegepast (een functie van type $a \to a$), en
3. een startwaarde van type a

De aanroep $until\ p\ f\ x$ kan gelezen worden als:
"pas net zo lang f toe op x totdat het resultaat voldoet aan p".

until

De definitie van *until* is weer recursief:

Definition (*until*)

$$\begin{aligned} until\ p\ f\ x \mid p\ x \quad &= x \\ \mid otherwise &= until\ p\ f\ (f\ x) \end{aligned}$$

of gebruik makend van **if**::

$$until\ p\ f\ x = \textbf{if}\ p\ x\ \textbf{then}\ x\ \textbf{else}\ until\ p\ f\ (f\ x)$$

Als de startwaarde x meteen al aan de eigenschap p voldoet, dan is de startwaarde tevens de eindwaarde.

Voorbeelden van *until*

$$until\ p\ f\ x = \textbf{if}\ p\ x\ \textbf{then}\ x\ \textbf{else}\ until\ p\ f\ (f\ x)$$

Onderstaande expressie berekent bijvoorbeeld de eerste macht van twee die groter is dan 1000 (begin met 1, en verdubbel dan net zo lang tot het resultaat groter is dan 1000):

```
? until (>1000) (2*) 1
1024
```

Voorbeelden van *until*

$$until\ p\ f\ x = \textbf{if}\ p\ x\ \textbf{then}\ x\ \textbf{else}\ until\ p\ f\ (f\ x)$$

Onderstaande expressie berekent bijvoorbeeld de eerste macht van twee die groter is dan 1000 (begin met 1, en verdubbel dan net zo lang tot het resultaat groter is dan 1000):

```
? until (>1000) (2*) 1
1024
```

Anders dan bij eerder besproken recursieve functies, is de parameter van de recursieve aanroep van *until* niet 'kleiner' dan de formele parameter. Daarom levert *until* niet altijd een resultaat op.

Een andere definitie van *until*

We kunnen *until* ook definiëren met behulp van andere fucties:

$$iterate\ f\ x = x : iterate\ f\ (f\ x)$$
$$until\ p\ f\ x = head\ (filter\ p\ (interate\ f\ x)$$

Een andere definitie van *until*

We kunnen *until* ook definiëren met behulp van andere fucties:

$$iterate\ f\ x = x : iterate\ f\ (f\ x)$$
$$until\ p\ f\ x = head\ (filter\ p\ (interate\ f\ x)$$

Of gebruik makend van functie compositie $(\circ)$

$$(\circ)\ f\ g\ x = f\ (g\ x)$$
$$until\ p\ f\ x = (head \circ filter\ p \circ interate\ f)\ x$$

Een andere definitie van *until*

We kunnen *until* ook definiëren met behulp van andere fucties:

$$iterate\ f\ x = x : iterate\ f\ (f\ x)$$
$$until\ p\ f\ x = head\ (filter\ p\ (interate\ f\ x)$$

Of gebruik makend van functie compositie $(\circ)$

$$(\circ)\ f\ g\ x = f\ (g\ x)$$
$$until\ p\ f\ x = (head \circ filter\ p \circ interate\ f)\ x$$

Of nog korter:

$$until\ p\ f = head \circ filter\ p \circ iterate\ f$$

Een andere definitie van *until*

We kunnen *until* ook definiëren met behulp van andere fucties:

$$iterate\ f\ x = x : iterate\ f\ (f\ x)$$
$$until\ p\ f\ x = head\ (filter\ p\ (interate\ f\ x)$$

Of gebruik makend van functie compositie $(\circ)$

$$(\circ)\ f\ g\ x = f\ (g\ x)$$
$$until\ p\ f\ x = (head \circ filter\ p \circ interate\ f)\ x$$

Of nog korter:

$$until\ p\ f = head \circ filter\ p \circ iterate\ f$$

Voor wie er niet genoeg van kan krijgen: :

$$until\ p = ((head \circ filter\ p)\circ) \circ iterate$$

Inmiddels bekende notaties voor “waarden”

Voor vrijwel alle types kunnen we waarden direct opschrijven:

- 1, 2.0 voor *Int*'s en *Float*'s
- *True* en *False* voor Boolean waarden
- `'a'` voor *Char*'s
- $[1,2,3]$ voor lijsten
- `"aap noot mies"` voor *String*'s ($[$*Char*$]$)

Inmiddels bekende notaties voor “waarden”

Voor vrijwel alle types kunnen we waarden direct opschrijven:

- 1, 2.0 voor *Int*'s en *Float*'s
- *True* en *False* voor Boolean waarden
- `'a'` voor *Char*'s
- $[1,2,3]$ voor lijsten
- `"aap noot mies"` voor *String*'s ($[Char]$)

De vraag die nu opkomt is:

Bestaat er ook zoiets voor functies?

Wanneer hebben we zoiets nodig?

Vaak is de functie die je als parameter meegeeft aan een andere functie vaak ontstaan door partiële parametrisatie:

$$map\ (+5)\ [1..10]$$
$$map\ (*2)\ [1..10]$$

Wanneer hebben we zoiets nodig?

Vaak is de functie die je als parameter meegeeft aan een andere functie vaak ontstaan door partiële parametrisatie:

$$map\ (+5)\ [1\,..\,10]$$
$$map\ (*2)\ [1\,..\,10]$$

Soms kan de functie die als parameter wordt meegegeven geconstrueerd worden door andere functies samen te stellen:

$$filter\ (\neg \circ even)\ [1\,..\,10]$$

Notatie voor “waarden van type $a \to \ldots$”

Maar soms is het te ingewikkeld om de functie op die manier te maken, bijvoorbeeld als we $x^2 + 3x + 1$ willen uitrekenen voor alle x in een lijst.

Het is dan altijd mogelijk om de functie apart te definiëren in een **where**-clausule:

$$ys = map\ f\ [1..10]$$
$$\quad \textbf{where}\ f\ x = x * x + 3 * x + 1$$

Notatie voor “waarden van type $a \to \ldots$”

Maar soms is het te ingewikkeld om de functie op die manier te maken, bijvoorbeeld als we $x^2 + 3x + 1$ willen uitrekenen voor alle x in een lijst.

Het is dan altijd mogelijk om de functie apart te definiëren in een **where**-clausule:

$$
\begin{aligned}
&ys = map\ f\ [1\,..\,10] \\
&\quad \textbf{where}\ f\ x = x * x + 3 * x + 1
\end{aligned}
$$

Als dit veel voorkomt is het echter een beetje vervelend dat je steeds een naam moet verzinnen voor de functie, en die dan achteraf definiëren.

Notatie voor “waarden van type $a \rightarrow \ldots$”

We voeren nu een notatie in voor functiewaarden:

$$\lambda < patroon > \rightarrow < expression >$$

Notatie voor “waarden van type $a \rightarrow \ldots$”

We voeren nu een notatie in voor functiewaarden:

$$\lambda <patroon> \rightarrow <expression>$$

Deze notatie staat bekend als de *lambda-notatie* (naar de Griekse letter λ; het symbool \ is de beste benadering voor die letter die op het toetsenbord beschikbaar is. . .)

Notatie voor “waarden van type $a \rightarrow \ldots$”

We voeren nu een notatie in voor functiewaarden:

$$\lambda <patroon> \rightarrow <expression>$$

Deze notatie staat bekend als de *lambda-notatie* (naar de Griekse letter λ; het symbool \ is de beste benadering voor die letter die op het toetsenbord beschikbaar is. . .)
Een voorbeeld van de lambda-notatie is de functie:

$$\lambda x \rightarrow x * x + 3 * x + 1$$

Notatie voor "waarden van type $a \to \ldots$"

We voeren nu een notatie in voor functiewaarden:

$$\lambda <patroon> \to <expression>$$

Deze notatie staat bekend als de *lambda-notatie* (naar de Griekse letter λ; het symbool \ is de beste benadering voor die letter die op het toetsenbord beschikbaar is...)
Een voorbeeld van de lambda-notatie is de functie:

$$\lambda x \to x * x + 3 * x + 1$$

"de functie die bij x de waarde $x^2 + 3x + 1$ oplevert".

Notatie voor "waarden van type $a \rightarrow \ldots$"

We voeren nu een notatie in voor functiewaarden:

$$\lambda <patroon> \rightarrow <expression>$$

Deze notatie staat bekend als de *lambda-notatie* (naar de Griekse letter λ; het symbool \ is de beste benadering voor die letter die op het toetsenbord beschikbaar is...)
Een voorbeeld van de lambda-notatie is de functie:

$$\lambda x \rightarrow x * x + 3 * x + 1$$

We kunnen nu schrijven:

$$ys = map\ (\lambda x \rightarrow x * x + 3 * x + 1)\ [1..100]$$

Syntactic Sugar

Syntactic Sugar

Onder syntactic sugar verstaan we een notatie die niets essentieels toevoegt aan de programmeertaal, maar die we kunnen gebruiken om programma's er fraaier uit te laten zien, of om ze gemakkelijker op te schrijven.

Syntactic Sugar

Syntactic Sugar

Onder syntactic sugar verstaan we een notatie die niets essentieels toevoegt aan de programmeertaal, maar die we kunnen gebruiken om programma's er fraaier uit te laten zien, of om ze gemakkelijker op te schrijven.

Zoals met alle suiker: een beetje is wel lekker, maar je moet er niet te veel van hebben.

Verschillende schrijfwijzen

Deze nieuwe notatie geeft ons wat nieuwe manieren om zaken op te schrijven:

Verschillende schrijfwijzen

Deze nieuwe notatie geeft ons wat nieuwe manieren om zaken op te schrijven:

$$f\ x = x * x + 3 * x + 1$$

staat eigenlijk voor:

$$f = \lambda x \rightarrow x * x + 3 * x + 1$$

Verschillende schrijfwijzen

Deze nieuwe notatie geeft ons wat nieuwe manieren om zaken op te schrijven:

$$f\ x = x * x + 3 * x + 1$$

staat eigenlijk voor:

$$f = \lambda x \to x * x + 3 * x + 1$$

Het volgende is allemaal hetzelfde:

$$\begin{array}{lllll} f\ x\ y\ z = & & & & x + y + z \\ f = & \lambda x \to & \lambda y \to & \lambda z \to & x + y + z \\ f = & \lambda x & y \to & z \to & x + y + z \\ f\ x\ y = & \lambda & & z \to & x + y + z \end{array}$$

Nog even terug

We hebben afgeleid dat:

$$filter\ p\ l = foldr\ op\ [\,]\ l$$
$$\quad \textbf{where}\ x\ `op`\ rest = \textbf{if}\ p\ x\ \textbf{then}\ x : rest\ \textbf{else}\ rest$$

Nog even terug

We hebben afgeleid dat:

$$filter\ p\ l = foldr\ op\ [\,]\ l$$
$$\quad \textbf{where}\ x\ `op`\ rest = \textbf{if}\ p\ x\ \textbf{then}\ x : rest\ \textbf{else}\ rest$$

In de nieuwe notatie kunnen we nu dus schrijven:

$$filter\ p\ l = foldr\ (\lambda x\ rest \rightarrow \textbf{if}\ p\ x\ \textbf{then}\ x : rest\ \textbf{else}\ rest)\ [\,]\ l$$

Nog even terug

We hebben afgeleid dat:

$$filter\ p\ l = foldr\ op\ [\,]\ l$$
$$\quad \textbf{where}\ x\ `op`\ rest = \textbf{if}\ p\ x\ \textbf{then}\ x : rest\ \textbf{else}\ rest$$

In de nieuwe notatie kunnen we nu dus schrijven:

$$filter\ p\ l = foldr\ (\lambda x\ rest \rightarrow \textbf{if}\ p\ x\ \textbf{then}\ x : rest\ \textbf{else}\ rest)\ [\,]\ l$$

Of zelfs:

$$filter\ p = foldr\ (\lambda x\ rest \rightarrow \textbf{if}\ p\ x\ \textbf{then}\ x : rest\ \textbf{else}\ rest)\ [\,]$$

Of zelfs (maar dat maakt het alleen maar onduidelijker ;-}):

$$filter\ p = foldr\ (\lambda x \rightarrow \textbf{if}\ p\ x\ \textbf{then}\ (x:)\ \textbf{else}\ id)\ [\,]$$

Let-expressies

In plaats van de **where**-constructie om lokaal een aantal namen te introduceren kunnen we ook gebruik maken van een **let** ... **in** ... constructie:

$$
\begin{aligned}
f\ x\ y = \textbf{let}\ & grootste = x\ `max`\ y \\
& kleinste\ = x\ `min`\ y \\
\textbf{in}\ & grootste - kleinste
\end{aligned}
$$

Let-expressies

In plaats van de **where**-constructie om lokaal een aantal namen te introduceren kunnen we ook gebruik maken van een **let** ... **in** ... constructie:

$$
\begin{aligned}
f\ x\ y = \textbf{let}\ & grootste = x\ `max`\ y \\
& kleinste\ = x\ `min`\ y \\
\textbf{in}\ & grootste - kleinste
\end{aligned}
$$

Let er op:

- dat de definities in een **let** ... **in** ... weer netjes in dezelfde kolom beginnen!
- dat je de **in** recht onder de **let** zet.
- dat namen die je in een let introduceert meer globale namen afschermen. Deze worden dus tijdelijk onzichtbaar.

Nog een keer foldr

Met behulp van de **let** kunnen we de functie *foldr* ook anders schrijven. Soms is dit een beetje efficiënter (maar dat hangt ook van de vertaler af).

$$
\begin{aligned}
\mathit{foldr}\ \mathit{op}\ e = \textbf{let}\ \ & \mathit{combine}\ (x : xs) = x\ `\mathit{op}`\ \mathit{combine}\ xs \\
& \mathit{combine}\ [\,] \quad\ = e \\
\textbf{in}\ & \mathit{combine}
\end{aligned}
$$

Nog een keer foldr

Met behulp van de **let** kunnen we de functie *foldr* ook anders schrijven. Soms is dit een beetje efficiënter (maar dat hangt ook van de vertaler af).

$$
\begin{aligned}
\mathit{foldr}\ \mathit{op}\ e = \textbf{let}\ \ & \mathit{combine}\ (x : xs) = x\ `\mathit{op}`\ \mathit{combine}\ xs \\
& \mathit{combine}\ [\,] \quad\ = e \\
\textbf{in}\ & \mathit{combine}
\end{aligned}
$$

Het idee is dat je op deze manier aangeeft dat de parameters *op* en *e* vast zijn, en dat het lijst argument varieert.

Input en Output

Tot dusverre:

- bestond een programma uit een aantal functie definities
- en kon je de waarde van een expressie laten uitrekenen door hem in de interpretator in te tikken

Input en Output

Tot dusverre:

- bestond een programma uit een aantal functie definities
- en kon je de waarde van een expressie laten uitrekenen door hem in de interpretator in te tikken

We willen in Haskell ook echt programma's schrijven, zoals:

- Quake
- Haskell compilers
- Grafische editors
- Belastingbiljetinvulprogramma's
- Webservers, Wiki systemen
- ...

Wat is het probleem

Haskell kent zogenaamde *referential transparency*, d.w.z. dat je op de plaats van een variabele ook altijd de rechterkant van de definitie op mag schrijven:

Wat is het probleem

Haskell kent zogenaamde *referential transparency*, d.w.z. dat je op de plaats van een variabele ook altijd de rechterkant van de definitie op mag schrijven: Dus

> **let** $x = <expr>$ **in** $\ldots x \ldots x \ldots$

mag niet wat anders betekenen dan:

> $\ldots <expr> \ldots <expr> \ldots$

Wat is het probleem

Haskell kent zogenaamde *referential transparency*, d.w.z. dat je op de plaats van een variabele ook altijd de rechterkant van de definitie op mag schrijven: Dus

> **let** $x = <expr>$ **in** $\ldots x \ldots x \ldots$

mag niet wat anders betekenen dan:

> $\ldots <expr> \ldots <expr> \ldots$

Zo moet ook:

> $(\lambda x \rightarrow <expr1>) <expr2>$

altijd gelijk zijn aan:

> **let** $x = <expr2>$ **in** $<expr1>$

Side-effects ;-{{

Stel nu dat we in Haskell een functie *random* zouden hebben die telkens een andere waarde oplevert –en dus onder water gebruik maakt van iets van toestand, om er voor te zorgen dat niet telkens dezelfde waarde wordt opgeleverd–

Side-effects ;-{{

Stel nu dat we in Haskell een functie *random* zouden hebben die telkens een andere waarde oplevert –en dus onder water gebruik maakt van iets van toestand, om er voor te zorgen dat niet telkens dezelfde waarde wordt opgeleverd– dan zouden dus:

> **let** $x = random$ **in** $x \equiv x$

en

> $random \equiv random$

hetzelfde moeten betekenen.

Side-effects ;-{{

Hetzelfde geldt voor the inlezen van een character:

> **let** $x = getChar$ **in** $x \equiv x$

en

> $getChar \equiv getChar$

zouden hetzelfde moeten betekenen, maar het is moeilijk in te zien hoe dat moet.

Acties

Dit probleem wordt in Haskell opgelost door een apart type $IO\ \_$ te introduceren:

$IO\ \_$

Waarden van type

$$IO\ ()$$

staan voor een rij input/output acties.

Acties

Dit probleem wordt in Haskell opgelost door een apart type $IO\ \_$ te introduceren:

$IO\ \_$

Waarden van type

$$IO\ ()$$

staan voor een rij input/output acties.

Een "normaal" Haskell programma bevat een functie $main :: IO\ ()$.

Acties

Dit probleem wordt in Haskell opgelost door een apart type $IO\ \_$ te introduceren:

$IO\ \_$

Waarden van type

$$IO\ ()$$

staan voor een rij input/output acties.

Een "normaal" Haskell programma bevat een functie $main :: IO\ ()$. Deze functie wordt standaard aangeroepen als je het programma opstart, en heeft als gevolg dat de rij acties die het resultaat zijn van het uitrekenen van de expressie die aan $main$ gebonden is, wordt uitgevoerd.

putChar

Kijk eens naar:

```
Prelude> 'a'
'a'
Prelude> :t putChar
putChar :: Char -> IO ()
Prelude> putChar 'a'
a
```

putChar

Kijk eens naar:

```
Prelude> 'a'
'a'
Prelude> :t putChar
putChar :: Char -> IO ()
Prelude> putChar 'a'
a
```

We doen hier twee verschillende dingen:

1. de waarde is niet van type $IO\ ()$: voeg afdrukken impliciet toe

putChar

Kijk eens naar:

```
Prelude> 'a'
'a'
Prelude> :t putChar
putChar :: Char -> IO ()
Prelude> putChar 'a'
a
```

We doen hier twee verschillende dingen:

1. de waarde is niet van type $IO\ ()$: voeg afdrukken impliciet toe

putChar

Kijk eens naar:

```
Prelude> 'a'
'a'
Prelude> :t putChar
putChar :: Char -> IO ()
Prelude> putChar 'a'
a
```

We doen hier twee verschillende dingen:

1. de waarde is niet van type $IO\ ()$: voeg afdrukken impliciet toe
2. wel van type $IO\ ()$: voer de acties uit

Samenstellen van acties

We kunnen acties samenstellen, d.w.z. achter elkaar hangen, gebruik makend van de **do** constructie:

```
tweeLetters :: IO ()
tweeLetters = do putChar 'H'
                 putChar 'i'
```

Samenstellen van acties

We kunnen acties samenstellen, d.w.z. achter elkaar hangen, gebruik makend van de **do** constructie:

```
tweeLetters :: IO ()
tweeLetters = do putChar 'H'
                 putChar 'i'
```

Let er op dat ook hier de indentatie er weer toe doet. We zullen zien dat de **do**-notatie op zich weer een vorm van syntactische suiker is, die ingevoerd is om je programma er een beetje imperatief uit te laten zien.

Recursieve acties

De meeste functies die iets met een lijst doen zijn recursief. Dit kunnen we ook nu weer toepassen:

```
putStr :: String→IO ()
putStr (x : xs) = do putChar x
                     putStr  xs
```

Recursieve acties

putStr

```
putStr :: String→IO ()
putStr (x : xs) = do putChar x
                     putStr  xs
putStr []       = return ()
```

Hierbij is *return* () de lege rij acties. We zullen straks zien wat het nut van de parameter van *return* is.

return

```
return :: a→IO a
```

Waarden doorgeven

We hebben nu wel gezien hoe we acties kunnen gebruiken, en samenstellen, maar de vraag die nu opkomt is:

> *Hoe kunnen we resultaten van eerdere acties gebruiken bij het constueren van verdere acties?*

Waarden doorgeven

We hebben nu wel gezien hoe we acties kunnen gebruiken, en samenstellen, maar de vraag die nu opkomt is:

> *Hoe kunnen we resultaten van eerdere acties gebruiken bij het constueren van verdere acties?*

Daarvoor gebruiken we het volgende voorbeeld:

```
echo :: IO ()
echo = do x ← getChar
          putChar x
```

Waarden doorgeven

We hebben nu wel gezien hoe we acties kunnen gebruiken, en samenstellen, maar de vraag die nu opkomt is:

> *Hoe kunnen we resultaten van eerdere acties gebruiken bij het constueren van verdere acties?*

Daarvoor gebruiken we het volgende voorbeeld:

$$
\begin{aligned}
&echo :: IO\ () \\
&echo = \textbf{do}\ x \leftarrow getChar \\
&\qquad\qquad putChar\ x
\end{aligned}
$$

```
Test> :t getChar
getChar :: IO Char
Test> echo
a
a
Test>
```

Ik verklap nog niets ..

Wat denk je dat het resultaat is van:

```
tweeLetters' :: IO ()
tweeLetters' = do putChar 'H'
                  return   "doet helemaal niets"
                  putChar 'i'
```

Ik verklap nog niets ..

Wat denk je dat het resultaat is van:

```
tweeLetters' :: IO ()
tweeLetters' = do putChar 'H'
                  return   "doet helemaal niets"
                  putChar 'i'
```

```
Test> tweeLetters'
Hi
Test>
```

Ik verklap nog niets ..

Wat denk je dat het resultaat is van:

```
tweeLetters' :: IO ()
tweeLetters' = do putChar 'H'
                  return   "doet helemaal niets"
                  putChar 'i'
```

```
Test> tweeLetters'
Hi
Test>
```

Dus *return* krijgt een argument mee, dat voor de rest van de berekening beschikbaar komt, maar helaas hier wordt weggegooid.

Onderliggende notatie

Wat denk je dat het resultaat is van:

```
tweeLetters' :: IO ()
tweeLetters' = do _ ← putChar 'H'
                  _ ← return   "doet helemaal niets"
                  _ ← putChar 'i'
```

Onderliggende notatie

Wat denk je dat het resultaat is van:

```
tweeLetters' :: IO ()
tweeLetters' = do _ ← putChar 'H'
                  _ ← return   "doet helemaal niets"
                  _ ← putChar 'i'
```

De *return* aanroep levert weliswaar een waarde op, maar die krijgt geen naam, en is verderop dus niet beschikbaar.

Vergelijk dit eens met..

```
tweeLetters'' :: IO ()
tweeLetters'' = do putChar 'H'
                   tekstje  <- return ", FP studenten"
                   putChar 'i'
                   putStr  tekstje
```

Vergelijk dit eens met..

```
tweeLetters'' :: IO ()
tweeLetters'' = do putChar 'H'
                   tekstje  <- return ", FP studenten"
                   putChar 'i'
                   putStr  tekstje
```

```
Test> tweeLetters''
Hi, FP studenten
Test>
```

De onderliggende notatie

```
tweeLetters'' :: IO ()
tweeLetters'' = do _      ← putChar 'H'
                   tekstje ← return ", FP studenten"
                   _      ← putChar 'i'
                   _      ← putStr  tekstje
```

De resultaten van alle acties (meestal $()$) krijgen geen naam, en wordt dus weggegooid.

Waarden van type $IO\ a$ zijn ook gewone waarden!

```
printLater =
  do _          ← putChar 'H'
     pr_tekstje ← return   (putStr ", FP studenten")
     _          ← putChar 'i'
     _          ← pr_tekstje
```

Waarden van type $IO\ a$ zijn ook gewone waarden!

```
printLater =
  do _          ← putChar 'H'
     pr_tekstje ← return   (putStr ", FP studenten")
     _          ← putChar 'i'
     _          ← pr_tekstje
```

De types zij nu:

- $return :: a \rightarrow IO\ a$

Waarden van type $IO\ a$ zijn ook gewone waarden!

```
printLater =
  do _          ← putChar 'H'
     pr_tekstje ← return   (putStr ", FP studenten")
     _          ← putChar 'i'
     _          ← pr_tekstje
```

De types zij nu:

- $return :: a \rightarrow IO\ a$
- $putStr$ `", FP studenten"` $:: IO\ ()$

Waarden van type $IO\ a$ zijn ook gewone waarden!

```
printLater =
  do _          ← putChar 'H'
     pr_tekstje ← return   (putStr ", FP studenten")
     _          ← putChar 'i'
     _          ← pr_tekstje
```

De types zij nu:

- $return :: a \rightarrow IO\ a$
- $putStr$ `", FP studenten"` $:: IO\ ()$
- $return\ (putStr$ `", FP studenten"`$) :: IO\ (IO\ ())$

Waarden van type $IO\ a$ zijn ook gewone waarden!

```
printLater =
  do _          ← putChar 'H'
     pr_tekstje ← return   (putStr ", FP studenten")
     _          ← putChar 'i'
     _          ← pr_tekstje
```

De types zij nu:

- $return :: a \rightarrow IO\ a$
- $putStr$ `", FP studenten"` $:: IO\ ()$
- $return\ (putStr$ `", FP studenten"`$) :: IO\ (IO\ ())$
- $pr\_tekstje :: IO\ ()$

getLine

De funcie $getLine :: IO\ String$ leest een hele regel in:

```
groet :: IO ()
groet = do putStr "Wat is je naam? "
           naam ← getLine
           putStrLn ("Hallo, " ++ naam)
```

getLine

De funcie $getLine :: IO\ String$ leest een hele regel in:

```
groet :: IO ()
groet = do putStr "Wat is je naam? "
           naam ← getLine
           putStrLn ("Hallo, " ++ naam)
```

```
Test> groet
Wat is je naam? Doaitse
Hallo, Doaitse
Test>
```

De functie *putStrLn* functioneert net als *putStr*, maar geeft nog een extra `'\n'` aan het eind.

Eenvoudige IO uit de prelude (1)

```
print :: Show a⇒a→IO ()
print e = putStrLn (show e)
   -- alternatief
print   = putStrLn ∘ show

getLine :: IO String
getLine = do x ← getChar
             if c ≡ '\n' then    return ""
                         else do xs ← getLine
                                 return (x : xs)
```

Eenvoudige IO uit de prelude (1)

```
print :: Show a⇒a→IO ()
print e = putStrLn (show e)
   -- alternatief
print   = putStrLn ∘ show

getLine :: IO String
getLine = do x ← getChar
             if c ≡ '\n' then    return ""
                         else do xs ← getLine
                                 return (x : xs)
```

Let op: beide takken van de **if** ... **then** ... **else** ... leveren weer iets van type *IO* ... op.

Eenvoudige IO uit de prelude (2)

De basis IO acties

$$
\begin{array}{lll}
return & :: & a \rightarrow IO\ a \\
putChar & :: & Char \rightarrow IO\ () \\
putStr & :: & String \rightarrow IO\ () \\
putStrLn & :: & String \rightarrow IO\ () \\
print & :: & Show\ a \Rightarrow a \rightarrow IO\ () \\
getLine & :: & IO\ String \\
writeFile & :: & String \rightarrow String \rightarrow IO\ () \\
readFile & :: & String \rightarrow IO\ String
\end{array}
$$

Ook hier “eigen controlestructuren”

We kunnen natuurlijk weer allemaal functies definiëren die acties samenstellen:

sequence

$$sequence :: [IO\ a] \rightarrow IO\ [a]$$

Ook hier “eigen controlestructuren”

We kunnen natuurlijk weer allemaal functies definiëren die acties samenstellen:

sequence

```
sequence :: [IO a] -> IO [a]
sequence []       = return []
sequence (s : ss) = do x <- s
                       xs <- sequence ss
                       return (x : xs)
```

Ook hier “eigen controlestructuren”

We kunnen natuurlijk weer allemaal functies definiëren die acties samenstellen:

sequence

```
sequence :: [IO a] → IO [a]
sequence []       = return []
sequence (s : ss) = do x ← s
                       xs ← sequence ss
                       return (x : xs)
```

De functie *sequence* neemt als argument een rij acties die allemaal een resultaat van type a opleveren, en bouwt een enkele, samengestelde actie, die de lijst van deelresultaten oplevert.

*sequence_*

*sequence_*

```
sequence_ :: [IO a] → IO ()
sequence_ []       = return ()
sequence_ (s : ss) = do s
                        sequence_ ss
```

*sequence_*

*sequence_*

```
sequence_ :: [IO a] → IO ()
sequence_ []       = return ()
sequence_ (s : ss) = do s
                        sequence_ ss
```

Nu kunnen we schrijven:

```
putStr str = sequence_ (map putChar str)
```

*sequence_*

*sequence_*

```
sequence_ :: [IO a] → IO ()
sequence_ []       = return ()
sequence_ (s : ss) = do s
                        sequence_ ss
```

Nu kunnen we schrijven:

```
putStr str = sequence_ (map putChar str)
```

Of nog korter:

```
putStr = sequence_ ∘ map putChar
```

Een voorbeeld van een “echt programma”

We gaan een "raad het getal" programma schrijven:

```
Test> main
Neem een getal <= 100  in gedachten.
Is het 50? (g = groter, k = kleiner, j = ja)
g
Is het 75? (g = groter, k = kleiner, j = ja)
k
Is het 62? (g = groter, k = kleiner, j = ja)
g
Is het 68? (g = groter, k = kleiner, j = ja)
k
Is het 65? (g = groter, k = kleiner, j = ja)
j
Geraden!
Test>
```

Het hoofdprogramma *main*

```
main = do
  putStrLn "Neem een getal <= 100 in gedachten."
  raad 1 100
```

De functie *raad*

```
raad :: Int→Int→IO ()
raad onder boven =
  do putStrLn ("Is het " ++ show midden
               ++     "? (g = groter, k = kleiner, j = ja)"
              )
     antwoord ← getLine
     verwerkAntwoord antwoord
  where ...
```

Universiteit Utrecht

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

De functie *raad*

where ...

```
verwerkAntwoord      :: String→IO ()
verwerkAntwoord "g" = raad (midden + 1) boven
verwerkAntwoord "k" = raad onder (midden - 1)
verwerkAntwoord "j" = putStrLn "Geraden!"
verwerkAntwoord _   = raad onder boven

midden               :: Int
midden               = (onder + boven) `div` 2
```

De functie *raad*

where ...

$$
\begin{aligned}
&\mathit{verwerkAntwoord} && :: \mathit{String} \to \mathit{IO}\ () \\
&\mathit{verwerkAntwoord}\ \texttt{"g"} && = \mathit{raad}\ (\mathit{midden} + 1)\ \mathit{boven} \\
&\mathit{verwerkAntwoord}\ \texttt{"k"} && = \mathit{raad}\ \mathit{onder}\ (\mathit{midden} - 1) \\
&\mathit{verwerkAntwoord}\ \texttt{"j"} && = \mathit{putStrLn}\ \texttt{"Geraden!"} \\
&\mathit{verwerkAntwoord}\ \_ && = \mathit{raad}\ \mathit{onder}\ \mathit{boven} \\
&\mathit{midden} && :: \mathit{Int} \\
&\mathit{midden} && = (\mathit{onder} + \mathit{boven})\ `\mathit{div}`\ 2
\end{aligned}
$$

Merk op hoe *main* de besturing over geeft aan *raad*, en *raad* aan *verwerkAntwoord*, en *verwerkAntwoord* zonodig weer aan een nieuwe instantie van *raad*.