Universiteit Utrecht

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

College 2010-2011
4.Rest IO en Lijsten

Doaitse Swierstra

Utrecht University

September 21, 2010

Herhaling van de IO

- Het type $IO\ a$ heeft als waarde een rij input/output acties

Herhaling van de IO

- Het type $IO\ a$ heeft als waarde een rij input/output acties
- met een **do** constructie kunnen dergelijke acties samengesteld worden; dit leidt tot een sequentiële executie van die acties

Herhaling van de IO

- Het type $IO\ a$ heeft als waarde een rij input/output acties
- met een **do** constructie kunnen dergelijke acties samengesteld worden; dit leidt tot een sequentiële executie van die acties
- Middels de $x \leftarrow expr$ vorm kunnen we waarden die het resultaat zijn van dergelijke acties in de rest van de **do** gebruiken.
- $main$ is van type $IO\ ()$, en stuurt de executie aan

We kunnen eigen controlestructure definiëren: *sequence_*

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

We kunnen eigen controlestructure definiëren: *sequence_*

$$
\begin{aligned}
&\mathit{sequence\_} :: [\mathit{IO}\ a] \to \mathit{IO}\ ()\\
&\mathit{sequence\_}\ \ [\,] \qquad = \mathit{return}\ ()\\
&\mathit{sequence\_}\ \ (s : ss) = \mathbf{do}\ s\\
&\qquad\qquad\qquad\qquad\qquad \mathit{sequence\_}\ ss
\end{aligned}
$$

Nu kunnen we schrijven:

$$\mathit{putStr} = \mathit{sequence\_} \circ \mathit{map}\ \mathit{putChar}$$

Gebruikelijke structuur van Haskell programma's

- De functie *main* is van type $IO\ ()$, en is het startpunt van het programma.

Gebruikelijke structuur van Haskell programma's

- De functie $main$ is van type $IO\ ()$, en is het startpunt van het programma.
- Op topniveau is sprake van sequentiële executie, waarbij functies elkaar de besturing doorgeven.

Gebruikelijke structuur van Haskell programma's

- De functie *main* is van type *IO* (), en is het startpunt van het programma.
- Op topniveau is sprake van sequentiële executie, waarbij functies elkaar de besturing doorgeven.
- Dit "doorgeven" zorg ervoor dat de input/output acties in IO sequentiëel worden afgehandeld.

Gebruikelijke structuur van Haskell programma's

- De functie *main* is van type *IO* (), en is het startpunt van het programma.
- Op topniveau is sprake van sequentiële executie, waarbij functies elkaar de besturing doorgeven.
- Dit "doorgeven" zorg ervoor dat de input/output acties in IO sequentiëel worden afgehandeld.
- Lazy evaluation zorgt er voor dat er al input/output acties kunnen plaatsvinden voordat de hele rij acties is berekend.

Sommetje

Probeer zelf eens een functie te schrijven:

$$vragen :: [String] \rightarrow IO\ [Bool]$$

die als parameter een lijst vragen mee krijgt, en als antwoord een rij Boolean waarden oplevert, die de gegeven antwoorden representeren.

Sommetje

Probeer zelf eens een functie te schrijven:

$vragen :: [String] \rightarrow IO\ [Bool]$

die als parameter een lijst vragen mee krijgt, en als antwoord een rij Boolean waarden oplevert, die de gegeven antwoorden representeren.

```
Test> vragen ["Is het zondag?", "Heet je Piet?"]
Is het zondag?
(j) of (n)?
n
Heet je Piet?
(j) of (n)?
j
FalseTrue
Test>
```

Oplossing

- $vragen :: [String] \rightarrow IO\ [Bool]$

"Let the types do the work"

Oplossing

$$vragen :: [String] \to IO\ [Bool]$$

"Let the types do the work"

$$\begin{aligned}
&stelVragen\ qs = sequence\ (map\ stelVraag\ qs)\\
&demo\ qs = \textbf{do}\ answers \leftarrow stelVragen\ qs\\
&\qquad\qquad\qquad putStrLn\ (show\ answers)
\end{aligned}$$

Oplossing

$vragen :: [String] \rightarrow IO\ [Bool]$

"Let the types do the work"

```
stelVragen qs = sequence (map stelVraag qs)
demo qs = do answers ← stelVragen qs
             putStrLn (show answers)
```

```
stelVraag q = do putStrLn q
                 putStrLn "(j) of (n)?"
                 (v: _) ← getLine
                 if v ≡ 'j' then return True
                            else if v ≡ 'n' then return False
                                            else stelVraag q
```

Nog even IO

We lezen de karakters in, plegen een recursieve aanroep, en bouwen het resultaat als we alle onderdelen hebben.

```
getLine1 = do x ← getChar
              if x ≡ '\n'
                then return []
                else do xs ← getLine1
                        return (x : xs)
```

Nog even IO

Veel mensen proberen het resultaat al te bouwen tijdens het inlezen. Dit kan met een accumulerende parameter. Maar zo is het wel duur.

$$
\begin{array}{l}
getLine2 = getLine2'\ [\,] \\
\quad \textbf{where}\ getLine2'\ r = \textbf{do}\ x \leftarrow getChar \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{if}\ x \equiv \texttt{'\textbackslash n'} \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{then}\ return\ r \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ getLine2'\ (r +\!\!+ [x])
\end{array}
$$

Nog even IO

$$
\begin{aligned}
&getLine3 = getLine3'\ [\,]\\
&\quad \textbf{where}\ getLine3'\ r = \textbf{do}\ x \leftarrow getChar\\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{if}\ x \equiv \texttt{'\textbackslash n'}\\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{then}\ return\ (reverse\ r)\\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ getLine3'\ (x : r)
\end{aligned}
$$

Dit is nauwelijks duurder dan de eerste oplossing.

Een tipje van de sluier

De **do**-notatie is eigenlijk ook weer syntactische suiker. Zo staat

$$\textbf{do}\ v \leftarrow \mathit{expr1}$$
$$\qquad \mathit{expr2}$$

eigenlijk voor:

$$\mathit{expr1} \ggg\!= \lambda v \rightarrow \mathit{expr2}$$

Een tipje van de sluier

De **do**-notatie is eigenlijk ook weer syntactische suiker. Zo staat

$$\textbf{do}\ v \leftarrow \mathit{expr1}$$
$$\quad \mathit{expr2}$$

eigenlijk voor:

$$\mathit{expr1} \ggg\!= \lambda v \rightarrow \mathit{expr2}$$

Vraagje: Wat denk je dat het type is van $\gg\!=$?

Een tipje van de sluier

De **do**-notatie is eigenlijk ook weer syntactische suiker. Zo staat

$$\textbf{do}\ v \leftarrow \mathit{expr1}$$
$$\quad \mathit{expr2}$$

eigenlijk voor:

$$\mathit{expr1} \gg\!= \lambda v \rightarrow \mathit{expr2}$$

Vraagje: Wat denk je dat het type is van $\gg\!=$?
Antwoord:

$$(\gg\!=) :: IO\ a \rightarrow (a \rightarrow IO\ b) \rightarrow IO\ b$$

Helemaal officieel is het nog iets algemener

En als je opvraagt in de GHCi (hier wordt weer overloading gebruikt):

$$
\begin{aligned}
(\ggg\!=) &:: \mathit{Monad}\ m \Rightarrow m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b \\
\mathit{return} &:: \mathit{Monad}\ m \Rightarrow \quad a \qquad\qquad\qquad\quad \rightarrow m\ a
\end{aligned}
$$

Helemaal officieel is het nog iets algemener

En als je opvraagt in de GHCi (hier wordt weer overloading gebruikt):

$$
\begin{aligned}
(\gg\!=) &:: Monad\ m \Rightarrow m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b \\
return &:: Monad\ m \Rightarrow \quad a \qquad\qquad\qquad\quad \rightarrow m\ a
\end{aligned}
$$

Hierin geeft $* \rightarrow *$ aan dat m een type constructor is, d.w.z. een soort functie van types naar types; net zoiets als $[\ldots]$ dus.

Lijsten hebben type $[a]$

Van de lege lijst $[\,]$ wordt het type van de elementen, zonodig, bepaald door de context:

$\mathit{sum}\ [\,]$	$[\,]$ *is een lege lijst getallen*
$\mathit{and}\ [\,]$	$[\,]$ *is een lege lijst Booleans*
$[[\,],[1,2],[3]]$	$[\,]$ *is een lege lijst getallen*
$[[1<2,\mathit{True}],[\,]]$	$[\,]$ *is een lege lijst Booleans*
$[[[1]],[\,]]$	$[\,]$ *is een lege lijst lijsten-van-getallen*
$\mathit{length}\ [\,]$	$[\,]$ *is een lege lijst (doet er niet toe waarvan)*

Constructoren

- Alle data types die we tegen zullen komen hebben constructoren.

Constructoren

- Alle data types die we tegen zullen komen hebben constructoren.
- Voor lijsten zijn dit:

$$
\begin{array}{lll}
(:) & :: & a \to [a] \to [a] \\
[\,] & :: & [a]
\end{array}
$$

Constructoren

- Alle data types die we tegen zullen komen hebben constructoren.
- Voor lijsten zijn dit:

$$
\begin{aligned}
(:) &:: a \to [a] \to [a] \\
[\,] &:: [a]
\end{aligned}
$$

- Het zijn precies deze constructoren die weer in patronen gebruikt kunnen worden:

$$
\begin{aligned}
&head\ (x : xs) = x \\
&tail\ \ (x : xs) = xs \\
&null\ [\,] \qquad\ \ = True
\end{aligned}
$$

Constructoren

- Alle data types die we tegen zullen komen hebben constructoren.
- Voor lijsten zijn dit:

$$
\begin{aligned}
(:) &:: a \to [a] \to [a] \\
[\,] &:: [a]
\end{aligned}
$$

- Het zijn precies deze constructoren die weer in patronen gebruikt kunnen worden:

$$
\begin{aligned}
&\mathit{head}\ (x : xs) = x \\
&\mathit{tail}\ \ (x : xs) = xs \\
&\mathit{null}\ [\,] \qquad\ \ = \mathit{True}
\end{aligned}
$$

- Een patroon kijkt of de desbetreffende constructor is gebruikt bij het bouwen van de waarden en geeft namen aan de constituenten

Notaties voor lijsten (1)

We hebben al de notatie m.b.v. de .. constructor gezien:

```
? [1..5]
[1, 2, 3, 4, 5]
? [2.5 .. 6.0]
[2.5, 3.5, 4.5, 5.5]
```

Notaties voor lijsten (1)

We hebben al de notatie m.b.v. de .. constructor gezien:

```
? [1..5]
[1, 2, 3, 4, 5]
? [2.5 .. 6.0]
[2.5, 3.5, 4.5, 5.5]
```

Dit is weer syntactische suiker, waarbij de compiler ervoor zorgt dat de functie:

$$\begin{aligned} \mathit{enumFromTo}\ x\ y \mid y < x &= [\,] \\ \mid \mathit{otherwise} &= x : \mathit{enumFromTo}\ (x+1)\ y \end{aligned}$$

aangeroepen wordt.

Notaties voor lijsten (1)

We hebben al de notatie m.b.v. de .. constructor gezien:

```
? [1..5]
[1, 2, 3, 4, 5]
? [2.5 .. 6.0]
[2.5, 3.5, 4.5, 5.5]
```

Dit is weer syntactische suiker, waarbij de compiler ervoor zorgt dat de functie:

$$\begin{array}{lll} \mathit{enumFromTo}\ x\ y & \mid y < x & = [\,] \\ & \mid \mathit{otherwise} & = x : \mathit{enumFromTo}\ (x+1)\ y \end{array}$$

aangeroepen wordt.

De meeste notaties maken gebruik van zulk soort standaard functies uit de prelude; je kunt ze ook zelf definiëren.

Gebruik je fantasie!

De .. notatie kan ook als volgt gebruikt worden:

```
? take 5 [2..]
[2,3,4,5,6]
```

Hoe zou dit nu weer gedefinieerd zijn?

Gebruik je fantasie!

De .. notatie kan ook als volgt gebruikt worden:

```
? take 5 [2..]
[2,3,4,5,6]
```

Hoe zou dit nu weer gedefinieerd zijn? Er wordt hier gebruikt gemaakt van de functie:

$$
\begin{aligned}
&\mathit{enumFrom} :: (\mathit{Enum}\ a) \Rightarrow a \to [a] \\
&\mathit{enumFrom}\ x = x : \mathit{enumFrom}\ (x + 1)
\end{aligned}
$$

Lijsten vergelijken

Alhoewel er in Haskell automatisch gelijkheid voor lijsten bestaat, mits er gelijkheid voor de elementen van die lijst bestaat, kunnen we die ook zelf definiëren:

$$
\begin{array}{llll}
eq :: & Eq\ a \Rightarrow [a] \rightarrow [a] \rightarrow Bool & & \\
[\,] & `eq`\ [\,] & = & True \\
[\,] & `eq`\ (y:ys) & = & False \\
(x:xs) & `eq`\ [\,] & = & False \\
(x:xs) & `eq`\ (y:ys) & = & x \equiv y \wedge xs\ `eq`\ ys
\end{array}
$$

Merk op dat deze functie twee argumenten tegelijk afbreekt.

Lijsten vergelijken

Alhoewel er in Haskell automatisch gelijkheid voor lijsten bestaat, mits er gelijkheid voor de elementen van die lijst bestaat, kunnen we die ook zelf definiëren:

$$
\begin{array}{lll}
eq :: & Eq\ a \Rightarrow [a] \rightarrow [a] \rightarrow Bool & \\
[\,] & `eq`\ [\,] & = True \\
[\,] & `eq`\ (y:ys) & = False \\
(x:xs) & `eq`\ [\,] & = False \\
(x:xs) & `eq`\ (y:ys) & = x \equiv y \wedge xs\ `eq`\ ys
\end{array}
$$

Merk op dat deze functie twee argumenten tegelijk afbreekt.

Maakt het iets uit in welke volgorde de alternatieven hier staan?

Lijsten vergelijken

Alhoewel er in Haskell automatisch gelijkheid voor lijsten bestaat, mits er gelijkheid voor de elementen van die lijst bestaat, kunnen we die ook zelf definiëren:

$$
\begin{array}{llll}
eq :: & \multicolumn{3}{l}{Eq\ a \Rightarrow [a] \rightarrow [a] \rightarrow Bool} \\
[\,] & `eq` & [\,] & = True \\
[\,] & `eq` & (y : ys) & = False \\
(x : xs) & `eq` & [\,] & = False \\
(x : xs) & `eq` & (y : ys) & = x \equiv y \wedge xs\ `eq`\ ys
\end{array}
$$

Merk op dat deze functie twee argumenten tegelijk afbreekt.

In principe niet, maar bij een niet zo slimme vertaler wellicht beter in omgekeerde volgorde.

Lexicografische ordening

De gebruikelijke ordening tussen lijsten is die zoals in het woordenboek (lexicon):

$$
\begin{array}{llll}
kg & & :: & Ord\ a \Rightarrow [a] \rightarrow [a] \rightarrow Bool \\
[\,] & `kg`\ ys & = & True \\
(x : xs) & `kg`\ [\,] & = & False \\
(x : xs) & `kg`\ (y : ys) & = & x < y \vee (x \equiv y \wedge xs\ `kg`\ ys)
\end{array}
$$

Lexicografische ordening

De gebruikelijke ordening tussen lijsten is die zoals in het woordenboek (lexicon):

$$
\begin{aligned}
&kg && :: \mathit{Ord}\ a \Rightarrow [a] \rightarrow [a] \rightarrow \mathit{Bool} \\
&[\,]\ \text{`}kg\text{`}\ ys && = \mathit{True} \\
&(x : xs)\ \text{`}kg\text{`}\ [\,] && = \mathit{False} \\
&(x : xs)\ \text{`}kg\text{`}\ (y : ys) && = x < y \lor (x \equiv y \land xs\ \text{`}kg\text{`}\ ys)
\end{aligned}
$$

Nu de functies *eq* en *kg* gedefinieerd zijn, kunnen andere vergelijkings-functies eenvoudig gedefinieerd worden:

$$
\begin{aligned}
xs\ \text{`}ng\text{`}\ ys &= \neg\ (xs\ \text{`}eq\text{`}\ ys) \\
xs\ \text{`}gg\text{`}\ ys &= ys\ \text{`}kg\text{`}\ xs \\
xs\ \text{`}kd\text{`}\ ys &= xs\ \text{`}kg\text{`}\ ys \land xs\ \text{`}ng\text{`}\ ys \\
xs\ \text{`}gd\text{`}\ ys &= ys\ \text{`}kd\text{`}\ xs
\end{aligned}
$$

Lijsten samenstellen

We kunnen twee lijsten aan elkaar hangen m.b.v. de functie $(+\!\!+)$:

$$
\begin{aligned}
&(+\!\!+) && :: [a] \to [a] \to [a] \\
&[\,] +\!\!+ \quad ys && = ys \\
&(x : xs) +\!\!+ ys && = x : (xs +\!\!+ ys)
\end{aligned}
$$

Lijsten samenstellen

We kunnen twee lijsten aan elkaar hangen m.b.v. de functie $(+\!\!+)$:

$$
\begin{aligned}
&(+\!\!+) && :: [a] \to [a] \to [a] \\
&[\,] +\!\!+ \; ys && = ys \\
&(x : xs) +\!\!+ ys && = x : (xs +\!\!+ ys)
\end{aligned}
$$

Kunnen we deze functie ook met een *foldr* schrijven?

Lijsten samenstellen

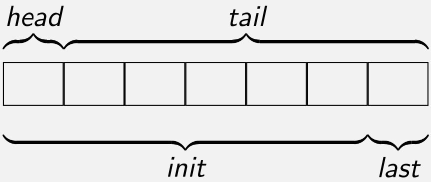
We kunnen twee lijsten aan elkaar hangen m.b.v. de functie $(+\!\!+)$:

$$
\begin{aligned}
&(+\!\!+) && :: [a] \to [a] \to [a] \\
&[\,] +\!\!+ && ys = ys \\
&(x : xs) +\!\!+ && ys = x : (xs +\!\!+ ys)
\end{aligned}
$$

Kunnen we deze functie ook met een *foldr* schrijven?

$$xs +\!\!+ ys = foldr\ (:)\ ys\ xs$$

head, *tail*, *init*, en *last*



De code

$$
\begin{aligned}
&\mathit{head}\ (x:xs) &&= x\\
&\mathit{tail}\ (x:xs) &&= xs\\
&\mathit{last}\ [x] &&= x\\
&\mathit{last}\ (x:y:ys) &&= \mathit{last}\ (y:ys)\\
&\mathit{init}\ (x:[\,]) &&= [\,]\\
&\mathit{init}\ (x:xs) &&= x:\mathit{init}\ xs
\end{aligned}
$$

De code

$$
\begin{aligned}
&\mathit{head}\ (x : xs) &&= x\\
&\mathit{tail}\ (x : xs) &&= xs\\
&\mathit{last}\ [x] &&= x\\
&\mathit{last}\ (x : y : ys) &&= \mathit{last}\ (y : ys)\\
&\mathit{init}\ (x : [\,]) &&= [\,]\\
&\mathit{init}\ (x : xs) &&= x : \mathit{init}\ xs
\end{aligned}
$$

In dictaat:

$$\mathit{last}\ (x : xs) = \mathit{last}\ xs$$

Wat is het verschil?

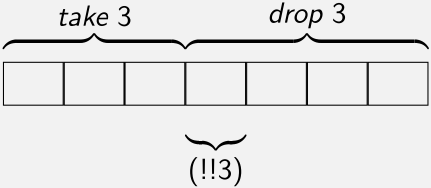
De code

$$
\begin{aligned}
&\mathit{head}\ (x : xs) &&= x\\
&\mathit{tail}\ \ (x : xs) &&= xs\\
&\mathit{last}\ [x] &&= x\\
&\mathit{last}\ (x : y : ys) &&= \mathit{last}\ (y : ys)\\
&\mathit{init}\ (x : [\,]) &&= [\,]\\
&\mathit{init}\ (x : xs) &&= x : \mathit{init}\ xs
\end{aligned}
$$

Nog gemakkelijker is natuurlijk:

$$
\begin{aligned}
\mathit{last} &= \mathit{head} \circ \mathit{reverse}\\
\mathit{init} &= \mathit{reverse} \circ \mathit{tail} \circ \mathit{reverse}
\end{aligned}
$$

take, *drop*, en !!



De code

$$
\begin{aligned}
&\mathit{take}, \mathit{drop} && :: \mathit{Int} \to [a] \to [a] \\
&\mathit{take}\ 0\ \mathit{xs} && = [\,] \\
&\mathit{take}\ n\ [\,] && = [\,] \\
&\mathit{take}\ n\ (x : \mathit{xs}) && = x : \mathit{take}\ (n - 1)\ \mathit{xs}
\end{aligned}
$$

De code

$$
\begin{array}{ll}
\mathit{take}, \mathit{drop} & :: \mathit{Int} \rightarrow [a] \rightarrow [a] \\
\mathit{take}\ 0\ \mathit{xs} & = [\,] \\
\mathit{take}\ n\ [\,] & = [\,] \\
\mathit{take}\ n\ (x : \mathit{xs}) & = x : \mathit{take}\ (n - 1)\ \mathit{xs}
\end{array}
$$

$$
\begin{array}{ll}
\mathit{drop}\ 0\ \mathit{xs} & = \mathit{xs} \\
\mathit{drop}\ n\ [\,] & = [\,] \\
\mathit{drop}\ n\ (x : \mathit{xs}) & = \mathit{drop}\ (n - 1)\ \mathit{xs}
\end{array}
$$

De code

$$
\begin{aligned}
&\mathit{take}, \mathit{drop} && :: \mathit{Int} \rightarrow [a] \rightarrow [a] \\
&\mathit{take}\ 0\ \mathit{xs} && = [\,] \\
&\mathit{take}\ n\ [\,] && = [\,] \\
&\mathit{take}\ n\ (x : \mathit{xs}) && = x : \mathit{take}\ (n-1)\ \mathit{xs}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{drop}\ 0\ \mathit{xs} && = \mathit{xs} \\
&\mathit{drop}\ n\ [\,] && = [\,] \\
&\mathit{drop}\ n\ (x : \mathit{xs}) && = \mathit{drop}\ (n-1)\ \mathit{xs}
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{infixl}\ 9\ !! \\
&(!!) && :: [a] \rightarrow \mathit{Int} \rightarrow a \\
&(x : \mathit{xs})\ !!\ 0 && = x \\
&(x : \mathit{xs})\ !!\ n && = \mathit{xs}\ !!\ (n-1)
\end{aligned}
$$

reverse

We kunnen een lijst omkeren: De definitie kan dus als volgt luiden:

$$
\begin{aligned}
&\mathit{reverse}\ [\,] &&= [\,] \\
&\mathit{reverse}\ (x : xs) &&= \mathit{reverse}\ xs \mathbin{+\!\!+} [x]
\end{aligned}
$$

reverse

We kunnen een lijst omkeren: De definitie kan dus als volgt luiden:

$$
\begin{aligned}
&\mathit{reverse}\ [\,] &&= [\,]\\
&\mathit{reverse}\ (x : xs) &&= \mathit{reverse}\ xs +\!\!+ [x]
\end{aligned}
$$

Dit is heel erg duur bij lange lijsten!!

reverse

We kunnen een lijst omkeren: De definitie kan dus als volgt luiden:

$$\begin{aligned} &\mathit{reverse}\ [\,] &&= [\,] \\ &\mathit{reverse}\ (x : xs) &&= \mathit{reverse}\ xs \mathbin{+\!\!+} [x] \end{aligned}$$

We bouwen het resultaat op in een extra argument, dat we aan het eind opleveren.

$$\begin{aligned} &\mathit{reverse}\ x = \mathit{reverseacc}\ x\ [\,] \\ &\quad \textbf{where}\ \mathit{reverseacc}\ (x : xs)\ \mathit{result} = \mathit{reverseacc}\ xs\ (x : \mathit{result}) \\ &\quad \phantom{\textbf{where}}\ \mathit{reverseacc}\ [\,]\ \ \mathit{result} = \mathit{result} \end{aligned}$$

We noemen een parameter waarin een resultaat wordt opgebouwd een accumulerende parameter.

Precies dezelfde *reverse*

Een iets ander formulering van precies hetzelfde algoritme is:

$$
\begin{aligned}
&\mathit{reverse}\ x = (\mathit{reverseacc}\ x)\ [\,] \\
&\quad \textbf{where}\ \mathit{reverseacc}\ (x : xs) = \mathit{reverseacc}\ xs \circ (x{:}) \\
&\qquad\qquad\ \ \mathit{reverseacc}\ [\,] \qquad\ = \mathit{id}
\end{aligned}
$$

- Ga na dat deze code een directe transformatie is van de die op de vorige slide.
- Kun je *reverseacc* weer met een *foldr* schrijven?

Opgave van Haskell-caf mailing list

Schrijf een functie die de laatste n elementen van een lijst oplevert:

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

Opgave van Haskell-caf mailing list

Schrijf een functie die de laatste n elementen van een lijst oplevert:

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

Oplossing:

$$last\ n\ l = reverse\ (take\ n\ (reverse\ l))$$

Opgave van Haskell-caf mailing list

Schrijf een functie die de laatste n elementen van een lijst oplevert:

$$lastn :: Int \to [a] \to [a]$$

Oplossing:

$$last\ n\ l = reverse\ (take\ n\ (reverse\ l))$$

Of mooier:

$$last\ n = reverse \circ take\ n \circ reverse$$

Opgave van Haskell-caf mailing list

Schrijf een functie die de laatste n elementen van een lijst oplevert:

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

Oplossing:

$$last\ n\ l = reverse\ (take\ n\ (reverse\ l))$$

Of mooier:

$$last\ n = reverse \circ take\ n \circ reverse$$

Hoeveel ruimte neemt deze functie?

Opgave van Haskell-caf mailing list

Schrijf een functie die de laatste n elementen van een lijst oplevert:

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

Oplossing:

$$last\ n\ l = reverse\ (take\ n\ (reverse\ l))$$

Of mooier:

$$last\ n = reverse \circ take\ n \circ reverse$$

Hoeveel ruimte neemt deze functie? Het ruimtebeslag is hier gelijk aan de lengte van de lijst, want $take\ n$ begint met te matchen op de laatste : constructor!

Opgave van Haskell-caf mailing list

Schrijf een functie die de laatste n elementen van een lijst oplevert:

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

Oplossing:

$$last\ n\ l = reverse\ (take\ n\ (reverse\ l))$$

Of mooier:

$$last\ n = reverse \circ take\ n \circ reverse$$

Hoeveel ruimte neemt deze functie? Het ruimtebeslag is hier gelijk aan de lengte van de lijst, want $take\ n$ begint met te matchen op de laatste : constructor! Kan je het ook doen met een hoeveelheid ruimte die $O(n)$ is?

takeWhile en *dropWhile*

$$
\begin{array}{lll}
\mathit{takeWhile} & & :: (a \rightarrow \mathit{Bool}) \rightarrow [a] \rightarrow [a] \\
\mathit{takeWhile}\ p\ [\,] & & = [\,] \\
\mathit{takeWhile}\ p\ (x : xs) & \mid p\ x & = x : \mathit{takeWhile}\ p\ xs \\
 & \mid \mathit{otherwise} & = [\,]
\end{array}
$$

Vergelijk deze definitie met die van *filter*.

takeWhile en *dropWhile*

$$\begin{aligned}
&\mathit{takeWhile} && :: (a \to Bool) \to [a] \to [a] \\
&\mathit{takeWhile}\ p\ [\,] && = [\,] \\
&\mathit{takeWhile}\ p\ (x:xs)\ \mid p\ x && = x : \mathit{takeWhile}\ p\ xs \\
& \qquad\qquad\qquad\quad\ \mid \mathit{otherwise} && = [\,]
\end{aligned}$$

Vergelijk deze definitie met die van *filter*.

$$\begin{aligned}
&\mathit{dropWhile} && :: (a \to Bool) \to [a] \to [a] \\
&\mathit{dropWhile}\ p\ [\,] && = [\,] \\
&\mathit{dropWhile}\ p\ (x:xs)\ \mid p\ x && = \mathit{dropWhile}\ p\ xs \\
& \qquad\qquad\qquad\quad\ \mid \mathit{otherwise} && = x : xs
\end{aligned}$$

foldl

$$
\begin{array}{rcccccccccc}
xs = & [\ 1 & , & 2 & , & 3 & , & 4 & , & 5] \\
 & \downarrow & & \downarrow & & \downarrow & & \downarrow & & \downarrow \\
\mathit{foldl}\ (+)\ 0\ xs = & (((((0 + 1) & + & 2) & + & 3) & + & 4) & + & 5)
\end{array}
$$

Twee voorbeelden

We bekijken een stukje van de gewenste berekening:

$$\mathit{foldl}\ (\oplus)\quad a\ \ [\,x\quad,\,y\quad,\,z\,]$$

Twee voorbeelden

We bekijken een stukje van de gewenste berekening:

$$\mathit{foldl}\ (\oplus)\ a\ [x, y, z] = ((a \oplus x) \oplus y) \oplus z$$

Twee voorbeelden

We bekijken een stukje van de gewenste berekening:

$$
\begin{array}{l}
\mathit{foldl}\ (\oplus)\ \ a\ \ [\,x\ \ ,y\ \ ,z\,] \\
\quad = \\
\quad ((a\ \oplus x)\ \oplus y) \oplus z \\
\quad = \\
\mathit{foldl}\ (\oplus)\ (a\ \oplus x)\ \ [\,y\ \ ,z\,]
\end{array}
$$

Twee voorbeelden

We bekijken een stukje van de gewenste berekening:

$$
\begin{array}{l}
\mathit{foldl}\ (\oplus)\ a\ [x, y, z] \\
= \\
((a \oplus x) \oplus y) \oplus z \\
= \\
\mathit{foldl}\ (\oplus)\ (a \oplus x)\ [y, z]
\end{array}
$$

$$
\begin{array}{l}
\mathit{foldl}\ (\oplus) a\ [\,] \\
= \\
a
\end{array}
$$

Twee voorbeelden

We bekijken een stukje van de gewenste berekening:

$$\begin{array}{l} \mathit{foldl}\ (\oplus)\ \ a\ \ [x\ ,\ y\ ,\ z] \\ \qquad = \\ \qquad ((a \oplus x) \oplus y) \oplus z \\ \qquad = \\ \mathit{foldl}\ (\oplus)\ (a \oplus x)\ [y\ ,\ z] \end{array}$$

$$\begin{array}{l} \mathit{foldl}\ (\oplus) a\ [\,] \\ \qquad = \\ \qquad a \end{array}$$

Hieruit blijkt dat aanroep van foldl op de lijst $x : xs$ (met $xs = [y, z]$ in het voorbeeld) gelijk is aan $\mathit{foldl} \ldots xs$, mits in de recursieve aanroep als startwaarde de waarde $a \oplus x$ genomen wordt.

De code

Met deze observatie kan de definitie geschreven worden:

$$
\begin{aligned}
&\mathit{foldl}\ \mathit{op}\ e\ [\,] &&= e\\
&\mathit{foldl}\ \mathit{op}\ e\ (x:xs) &&= \mathit{foldl}\ \mathit{op}\ (e\ \text{`}\mathit{op}\text{`}\ x)\ xs
\end{aligned}
$$

De code

Met deze observatie kan de definitie geschreven worden:

$$\begin{aligned} &\mathit{foldl}\ \mathit{op}\ e\ [\,] &&= e \\ &\mathit{foldl}\ \mathit{op}\ e\ (x : xs) &&= \mathit{foldl}\ \mathit{op}\ (e\ `\mathit{op}`\ x)\ xs \end{aligned}$$

We maken hier dus weer gebruik van een accumulerende parameter. Verder noemen we de functie *foldl* tail-recursief, omdat zijn definitie weer bestaat uit een aanroep van de functie zelf.

De code

Met deze observatie kan de definitie geschreven worden:

$$\begin{aligned} &\mathit{foldl}\ \mathit{op}\ e\ [\,] &&= e \\ &\mathit{foldl}\ \mathit{op}\ e\ (x : \mathit{xs}) &&= \mathit{foldl}\ \mathit{op}\ (e\ \text{`}\mathit{op}\text{`}\ x)\ \mathit{xs} \end{aligned}$$

We maken hier dus weer gebruik van een accumulerende parameter. Verder noemen we de functie *foldl* tail-recursief, omdat zijn definitie weer bestaat uit een aanroep van de functie zelf. Hoeveel ruimte neemt deze functie?

De code

Met deze observatie kan de definitie geschreven worden:

$$
\begin{aligned}
&\mathit{foldl}\ \mathit{op}\ e\ [\,] && = e\\
&\mathit{foldl}\ \mathit{op}\ e\ (x:xs) && = \mathit{foldl}\ \mathit{op}\ (e\,`\mathit{op}`\,x)\ xs
\end{aligned}
$$

We maken hier dus weer gebruik van een accumulerende parameter. Verder noemen we de functie *foldl* tail-recursief, omdat zijn definitie weer bestaat uit een aanroep van de functie zelf. Hoeveel ruimte neemt deze functie? $O\ (\mathit{length}\ l)$

De code

Met deze observatie kan de definitie geschreven worden:

$$
\begin{aligned}
&\mathit{foldl}\ \mathit{op}\ e\ [\,] &&= e\\
&\mathit{foldl}\ \mathit{op}\ e\ (x : \mathit{xs}) &&= \mathit{foldl}\ \mathit{op}\ (e\ \text{`}\mathit{op}\text{`}\ x)\ \mathit{xs}
\end{aligned}
$$

We maken hier dus weer gebruik van een accumulerende parameter. Verder noemen we de functie *foldl* tail-recursief, omdat zijn definitie weer bestaat uit een aanroep van de functie zelf. Hoeveel ruimte neemt deze functie? $O\ (\mathit{length}\ l)$ Gebruik waar mogelijk dus foldl':

$$
\begin{aligned}
&\mathit{foldl}'\ \mathit{op}\ e\ [\,] &&= e\\
&\mathit{foldl}'\ \mathit{op}\ e\ (x : \mathit{xs}) &&= \textbf{let}\ \mathit{next} = e\ \text{`}\mathit{op}\text{`}\ x\\
& &&\quad\ \textbf{in}\ \mathit{next}\ \text{`}\mathit{seq}\text{`}\ \mathit{foldl}'\ \mathit{op}\ \mathit{next}\ \mathit{xs}
\end{aligned}
$$

De ‘*seq*‘ dwingt af dat eerst de linkerkant wordt uitgerekend voordat de rechterkant wordt geëvalueerd.

Sorteren door samenvoegen

We kunnen twee gesorteerde lijsten samenvoegen tot een lijst die weer gesorteerd is:

$$
\begin{array}{llll}
\mathit{merge} :: & \multicolumn{3}{l}{\mathit{Ord}\ a \Rightarrow [a] \rightarrow [a] \rightarrow [a]} \\
\mathit{merge}\ [\,]\ \mathit{ys} & = \mathit{ys} & & \\
\mathit{merge}\ \mathit{xs}\ [\,] & = \mathit{xs} & & \\
\mathit{merge}\ (x : \mathit{xs})\ (y : \mathit{ys}) & \mid x \leqslant y & = x : \mathit{merge}\ \mathit{xs}\ (y : \mathit{ys}) & \\
 & \mid \mathit{otherwise} & = y : \mathit{merge}\ (x : \mathit{xs})\ \mathit{ys} &
\end{array}
$$

Sorteren door samenvoegen

We kunnen twee gesorteerde lijsten samenvoegen tot een lijst die weer gesorteerd is:

$$
\begin{array}{lllll}
\mathit{merge} :: & \mathit{Ord}\ a \Rightarrow [a] \rightarrow [a] \rightarrow [a] & & & \\
\mathit{merge}\ [\,]\ \mathit{ys} & = \mathit{ys} & & & \\
\mathit{merge}\ \mathit{xs}\ [\,] & = \mathit{xs} & & & \\
\mathit{merge}\ (x : \mathit{xs})\ (y : \mathit{ys}) & \mid x \leqslant y & = x : \mathit{merge} & \mathit{xs} & (y : \mathit{ys}) \\
 & \mid \mathit{otherwise} & = y : \mathit{merge} & (x : \mathit{xs}) & \mathit{ys}
\end{array}
$$

Hier kunnen we gebruik van maken om een lijst te sorteren.

$$
\begin{array}{llll}
\mathit{msort}\ \mathit{xs} & \mid \mathit{lengte} \leqslant 1 & = \mathit{xs} \\
 & \mid \mathit{True} & = (\mathit{msort}\ \mathit{ys})\ \text{`}\mathit{merge}\text{`}\ (\mathit{msort}\ \mathit{zs}) \\
 & \textbf{where}\ \mathit{half} & = \mathit{length}\ \mathit{xs}\ \text{`}\mathit{div}\text{`}\ 2 \\
 & \qquad\quad \mathit{ys} & = \mathit{take}\ \mathit{half}\ \mathit{xs} \\
 & \qquad\quad \mathit{zs} & = \mathit{drop}\ \mathit{half}\ \mathit{xs}
\end{array}
$$

Splitsen

Het gebruik van de functies *take* en *drop* is natuurlijk een beetje duur. We doorlopen uiteindelijk de eerste helft tweemaal. Kan het ook een beetje handiger?

Splitsen

Het gebruik van de functies *take* en *drop* is natuurlijk een beetje duur. We doorlopen uiteindelijk de eerste helft tweemaal. Kan het ook een beetje handiger?

$$
\begin{array}{lcl}
\mathit{splits} :: [a] \rightarrow ([a],[a]) & & \\
\mathit{splits}\ (x : xs) & = & \textbf{let}\ (vs, ws) = \mathit{splits}\ xs\ \textbf{in}\ (x : ws, vs) \\
\mathit{splits}\ [\,] & = & ([\,],[\,])
\end{array}
$$

Splitsen

Het gebruik van de functies *take* en *drop* is natuurlijk een beetje duur. We doorlopen uiteindelijk de eerste helft tweemaal. Kan het ook een beetje handiger?

$$
\begin{array}{lcl}
\mathit{splits} :: [a] \rightarrow ([a],[a]) & & \\
\mathit{splits}\ (x : xs) & = & \mathbf{let}\ (vs, ws) = \mathit{splits}\ xs\ \mathbf{in}\ (x : ws, vs) \\
\mathit{splits}\ [\,] & = & ([\,],[\,])
\end{array}
$$

Vraagje: had je dit ook met een *foldr* kunnen schijven?

Splitsen

Het gebruik van de functies *take* en *drop* is natuurlijk een beetje duur. We doorlopen uiteindelijk de eerste helft tweemaal. Kan het ook een beetje handiger?

$$
\begin{aligned}
&splits :: [a] \to ([a],[a]) \\
&splits\ (x : xs) = \mathbf{let}\ (vs, ws) = splits\ xs\ \mathbf{in}\ (x : ws, vs) \\
&splits\ [\,] \qquad\quad = ([\,],[\,])
\end{aligned}
$$

Vraagje: had je dit ook met een *foldr* kunnen schijven?

$$splits = foldr\ (\lambda x\ (vs, ws) \to (x : ws, vs))\ ([\,],[\,])$$

Anders kan natuurlijk ook ...

De functie *group* *merge*-t telkens twee buren uit een lijst van lijsten:

$$
\begin{array}{lcl}
\mathit{group} :: \mathit{Ord}\ a \Rightarrow [[a]] \rightarrow [[a]] & & \\
\mathit{group}\ (x : y : zs) & = & ((x\ `\mathit{merge}`\ y) : \mathit{group}\ zs) \\
\mathit{group}\ [x] & = & [x] \\
\mathit{group}\ [\,] & = & [\,]
\end{array}
$$

Anders kan natuurlijk ook ...

Nu roepen we op het resultaat weer *group* aan:

$$
\begin{aligned}
&group :: Ord\ a \Rightarrow [[a]] \rightarrow [[a]] \\
&group\ (x : y : zs) = group\ ((x \text{ `}merge\text{` } y) : group\ zs) \\
&group\ [x] \qquad\quad\ \ = [x] \\
&group\ [\,] \qquad\quad\ \ = [\,]
\end{aligned}
$$

Universiteit Utrecht

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

Anders kan natuurlijk ook ...

Nu roepen we op het resultaat weer *group* aan:

$$
\begin{array}{lcl}
\mathit{group} :: \mathit{Ord}\ a \Rightarrow [[a]] \rightarrow [[a]] & & \\
\mathit{group}\ (x : y : zs) & = & \mathit{group}\ ((x\ \text{`}\mathit{merge}\text{`}\ y) : \mathit{group}\ zs) \\
\mathit{group}\ [x] & = & [x] \\
\mathit{group}\ [\,] & = & [\,]
\end{array}
$$

Sorteren is nu een makkie: maak van alle elementen een lijst ($\mathit{map}\ (\lambda x \rightarrow [x])$), groepeer paarsgewijs totdat je er nog maar één over hebt (*group*), en pak dan de eerste (*head*):

Universiteit Utrecht

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

Anders kan natuurlijk ook ...

Nu roepen we op het resultaat weer *group* aan:

$$
\begin{aligned}
&group :: Ord\ a \Rightarrow [[a]] \rightarrow [[a]] \\
&group\ (x : y : zs) = group\ ((x \text{ `}merge\text{` } y) : group\ zs) \\
&group\ [x] \qquad\quad\ = [x] \\
&group\ [\,] \qquad\quad\ \ = [\,]
\end{aligned}
$$

Sorteren is nu een makkie: maak van alle elementen een lijst ($map\ (\lambda x \rightarrow [x])$), groepeer paarsgewijs totdat je er nog maar één over hebt (*group*), en pak dan de eerste (*head*):

$$sort = head \circ group \circ map\ (\lambda x \rightarrow [x])$$

Universiteit Utrecht

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

Oneindige lijsten

We hebben al wat voorbeelden gezien van oneindige lijsten.

Oneindige lijsten

We hebben al wat voorbeelden gezien van oneindige lijsten.

Lazy evaluation (normal order reduction)

In Haskell worden expressies niet verder uitgerekend dan nodig is!

Oneindige lijsten

We hebben al wat voorbeelden gezien van oneindige lijsten.

Lazy evaluation (normal order reduction)

In Haskell worden expressies niet verder uitgerekend dan nodig is!

```
? takeWhile (<1000) (map (3^) [0..])
[1, 3, 9, 27, 81, 243, 729]
```

Hammings problem

Hammings problem

Genereer een oplopende lijst van waarden, die geschreven kunnen worden als $\{2^i 3^j 5^k | i >= 0, j >= 0, k >= 0\}$.

Hammings problem

Hammings problem

Genereer een oplopende lijst van waarden, die geschreven kunnen worden als $\{2^i 3^j 5^k | i >= 0, j >= 0, k >= 0\}$.

Een typische manier om dit wat algoritmischer te formuleren is:

1. 1 is een Hamming getal.

Hammings problem

Hammings problem

Genereer een oplopende lijst van waarden, die geschreven kunnen worden als $\{2^i 3^j 5^k | i >= 0, j >= 0, k >= 0\}$.

Een typische manier om dit wat algoritmischer te formuleren is:

1. 1 is een Hamming getal.
2. Als n een Hamming getal is dat zijn $2 * n$, $3 * n$ en $5 * n$ ook Hamming getallen.

Hammings problem

Hammings problem

Genereer een oplopende lijst van waarden, die geschreven kunnen worden als $\{2^i 3^j 5^k | i >= 0, j >= 0, k >= 0\}$.

Een typische manier om dit wat algoritmischer te formuleren is:

1. 1 is een Hamming getal.
2. Als n een Hamming getal is dat zijn $2 * n$, $3 * n$ en $5 * n$ ook Hamming getallen.
3. Puriteinen voegen hier aan toe "En er zijn geen andere Hamming getallen", maar voor Informatici is dit vanzelfsprekend.

Hamming's probleem (code)

We redeneren nu als volgt:

1. Stel dat *ham* de gevraagde lijst is, dan zijn de lijsten *map* $(*2)$ *ham*, *map* $(*3)$ *ham*, en *map* $(*5)$ *ham* ook lijsten met Hamming getallen.

Hamming's probleem (code)

We redeneren nu als volgt:

1. Stel dat *ham* de gevraagde lijst is, dan zijn de lijsten *map* $(*2)$ *ham*, *map* $(*3)$ *ham*, en *map* $(*5)$ *ham* ook lijsten met Hamming getallen.
2. Als *ham* monotoon stijgt dan is dat voor deze nieuwe lijsten ook het geval.

Hamming's probleem (code)

We redeneren nu als volgt:

1. Stel dat ham de gevraagde lijst is, dan zijn de lijsten $map\ (*2)\ ham$, $map\ (*3)\ ham$, en $map\ (*5)\ ham$ ook lijsten met Hamming getallen.
2. Als ham monotoon stijgt dan is dat voor deze nieuwe lijsten ook het geval.
3. Veel getallen zullen in meerdere lijsten voor komen.

$$ham = 1 : \ldots$$

Hamming's probleem (code)

We redeneren nu als volgt:

1. Stel dat ham de gevraagde lijst is, dan zijn de lijsten $map\ (*2)\ ham$, $map\ (*3)\ ham$, en $map\ (*5)\ ham$ ook lijsten met Hamming getallen.
2. Als ham monotoon stijgt dan is dat voor deze nieuwe lijsten ook het geval.
3. Veel getallen zullen in meerdere lijsten voor komen.

$$
\begin{aligned}
ham = 1 : \ldots\ & (map\ (*2)\ ham) \\
& \ldots \\
& (map\ (*3)\ ham) \\
& \ldots \\
& (map\ (*5)\ ham)
\end{aligned}
$$

Hamming's probleem (code)

We redeneren nu als volgt:

1. Stel dat *ham* de gevraagde lijst is, dan zijn de lijsten $map\ (*2)\ ham$, $map\ (*3)\ ham$, en $map\ (*5)\ ham$ ook lijsten met Hamming getallen.
2. Als *ham* monotoon stijgt dan is dat voor deze nieuwe lijsten ook het geval.
3. Veel getallen zullen in meerdere lijsten voor komen.

$$
\begin{aligned}
ham = 1 : \ldots\ & (map\ (*2)\ ham) \\
& \quad \text{`}merge\text{`} \\
& (map\ (*3)\ ham) \\
& \quad \text{`}merge\text{`} \\
& (map\ (*5)\ ham)
\end{aligned}
$$

Hamming's probleem (code)

We redeneren nu als volgt:

1. Stel dat ham de gevraagde lijst is, dan zijn de lijsten $map\ (*2)\ ham$, $map\ (*3)\ ham$, en $map\ (*5)\ ham$ ook lijsten met Hamming getallen.
2. Als ham monotoon stijgt dan is dat voor deze nieuwe lijsten ook het geval.
3. Veel getallen zullen in meerdere lijsten voor komen.

$$
\begin{aligned}
ham = 1 : remdup\ (&(map\ (*2)\ ham) \\
&\quad \text{`}merge\text{`} \\
&(map\ (*3)\ ham) \\
&\quad \text{`}merge\text{`} \\
&(map\ (*5)\ ham) \\
&)
\end{aligned}
$$

$$remdup\ (x : ys) = x : remdup\ (dropWhile\ (\equiv x)\ ys)$$

Vraag

Waarom werkt dit niet met:

$$\begin{aligned} &\mathit{remdup}\ (x : y : zs) \mid x \equiv y &&= \mathit{remdup}\ (y : zs) \\ &\phantom{\mathit{remdup}\ (x : y : zs)} \mid \mathit{otherwise} &&= x : \mathit{remdup}\ (y : zs) \end{aligned}$$

Vraag

Waarom werkt dit niet met:

$$\begin{aligned} \mathit{remdup}\ (x:y:zs) \mid x \equiv y &= \mathit{remdup}\ (y:zs) \\ \mid \mathit{otherwise} &= x:\mathit{remdup}\ (y:zs) \end{aligned}$$

We evalueren een stukje:

Vraag

Waarom werkt dit niet met:

$$
\begin{aligned}
remdup\ (x : y : zs) \mid x \equiv y \quad &= \quad remdup\ (y : zs) \\
\mid otherwise &= x : remdup\ (y : zs)
\end{aligned}
$$

$$
\begin{aligned}
ham = 1 : remdup\ (&(map\ (*2)\ ham) \\
&\text{`}merge\text{`} \\
&(map\ (*3)\ ham) \\
&\text{`}merge\text{`} \\
&(map\ (*5)\ ham) \\
)&
\end{aligned}
$$

Vraag

Waarom werkt dit niet met:

$$
\begin{aligned}
\mathit{remdup}\ (x:y:zs) \mid x \equiv y &= \phantom{x:{}} \mathit{remdup}\ (y:zs) \\
\mid \mathit{otherwise} &= x : \mathit{remdup}\ (y:zs)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{ham} = 1 : \mathit{remdup}\ (&(2 : \mathit{map}\ (*2)\ (\mathit{tail}\ \mathit{ham})) \\
&\text{`}\mathit{merge}\text{`} \\
&(3 : \mathit{map}\ (*3)\ (\mathit{tail}\ \mathit{ham})) \\
&\text{`}\mathit{merge}\text{`} \\
&(5 : \mathit{map}\ (*5)\ (\mathit{tail}\ \mathit{ham})) \\
&)
\end{aligned}
$$

Vraag

Waarom werkt dit niet met:

$$
\begin{aligned}
&\mathit{remdup}\ (x : y : zs) \mid x \equiv y &&= \mathit{remdup}\ (y : zs)\\
&\phantom{\mathit{remdup}\ (x : y : zs)} \mid \mathit{otherwise} &&= x : \mathit{remdup}\ (y : zs)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{ham} = 1 : \mathit{remdup}\ (&(2 : (\mathit{map}\ ({*}2)\ (\mathit{tail}\ \mathit{ham}))\\
&\quad\text{`}\mathit{merge}\text{`}\\
&\quad (3 : \mathit{map}\ ({*}3)\ (\mathit{tail}\ \mathit{ham}))\\
&\text{`}\mathit{merge}\text{`}\\
&(5 : \mathit{map}\ ({*}5)\ (\mathit{tail}\ \mathit{ham}))\\
)&
\end{aligned}
$$

Vraag

Waarom werkt dit niet met:

$$
\begin{aligned}
\mathit{remdup}\ (x : y : zs) \mid x \equiv y &= \mathit{remdup}\ (y : zs) \\
\mid \mathit{otherwise} &= x : \mathit{remdup}\ (y : zs)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{ham} = 1 : \mathit{remdup}\ (2 : (\quad & (\mathit{map}\ (*2)\ (\mathit{tail}\ \mathit{ham})) \\
& \text{`}\mathit{merge}\text{`} \\
& (3 : \mathit{map}\ (*3)\ (\mathit{tail}\ \mathit{ham})) \\
& \text{`}\mathit{merge}\text{`} \\
& (5 : \mathit{map}\ (*5)\ (\mathit{tail}\ \mathit{ham})) \\
&)
\end{aligned}
$$

Vraag

Waarom werkt dit niet met:

$$\begin{aligned} remdup\ (x : y : zs)\ \mid x \equiv y &= remdup\ (y : zs) \\ \mid otherwise &= x : remdup\ (y : zs) \end{aligned}$$

$$\begin{aligned} ham = 1 : remdup\ (2 : (((2 * (head\ (tail\ ham) : map\ (*2)\ (tail\ (tail \\ &`merge` \\ &(3 : map\ (*3)\ (tail\ ham)) \\ &`merge` \\ &(5 : map\ (*5)\ (tail\ ham)) \\ &) \end{aligned}$$

Voor de *head* (*tail ham*) hebben we het resultaat van *remdup* nodig!

Productiviteit

Vergelijk de twee definities van *remdup*

$$
\begin{array}{llcl}
\mathit{remdup}\ (x:y:zs) & \mid x \equiv y & = & \mathit{remdup}\ (y:zs) \\
 & \mid \mathit{otherwise} & = & x:\mathit{remdup}\ (y:zs) \\
\mathit{remdup}'\ (x:ys) & & = & x:\mathit{remdup}'\ (\mathit{dropWhile}\ (\equiv x)\ ys)
\end{array}
$$

Productiviteit

Vergelijk de twee definities van *remdup*

$$
\begin{array}{lll}
remdup\ (x:y:zs) & \mid x \equiv y & = \quad remdup\ (y:zs) \\
 & \mid otherwise & = x : remdup\ (y:zs) \\
remdup'\ (x:ys) & & = x : remdup'\ (dropWhile\ (\equiv x)\ ys)
\end{array}
$$

Wanner we dit loslaten op een rij $[1, <expr1>, <expr2>]$ dan evalueert de eerste definitie de $<expr1>$, alvorens de 1 op te leveren. De tweede definitie levert die direct op.

Productiviteit

Vergelijk de twee definities van *remdup*

$$
\begin{array}{llcl}
\mathit{remdup}\ (x:y:zs) & \mid x \equiv y & = & \mathit{remdup}\ (y:zs) \\
 & \mid \mathit{otherwise} & = & x : \mathit{remdup}\ (y:zs) \\
\mathit{remdup}'\ (x:ys) & & = & x : \mathit{remdup}'\ (\mathit{dropWhile}\ (\equiv x)\ ys)
\end{array}
$$

Wanner we dit loslaten op een rij $[1, <\mathit{expr1}>, <\mathit{expr2}>]$ dan evalueert de eerste definitie de $<\mathit{expr1}>$, alvorens de 1 op te leveren. De tweede definitie levert die direct op.

Strictness

We zeggen dat de tweede definitie minder strict is dan de eerste: als er iets wordt opgeleverd door beide definities is dat hetzelfde, maar de tweede definitie zal vaker (eerder) iets opleveren dan de eerste.

Lijst-comprehensies

Haskell bevat veel syntactische suiker om lijsten gemakkelijk te noteren:

```
? [ (x, y) | x <- [1..5], even x, y <- [1..x]]
[(2,1),(2,2),(4,1),(4,2),(4,3),(4,4)]
```

Lijst-comprehensies

Haskell bevat veel syntactische suiker om lijsten gemakkelijk te noteren:

```
? [ (x, y) | x <- [1..5], even x, y <- [1..x]]
[(2,1),(2,2),(4,1),(4,2),(4,3),(4,4)]
```

Anders hadden we moeten schrijven:

$$\begin{array}{l} \mathit{concat}\ (\mathit{map}\ f\ (\mathit{filter}\ \mathit{even}\ [1\,..\,5])) \\ \textbf{where}\ f\ x = \mathit{map}\ g\ [1\,..\,x] \\ \qquad\qquad\quad \textbf{where}\ g\ y = (x, y) \end{array}$$

Ook dit is weer syntactische suiker

We noemen de elementen achter de verticale streep qualifiers.
Ze hebben een van de volgende drie vormen:

Ook dit is weer syntactische suiker

We noemen de elementen achter de verticale streep qualifiers.
Ze hebben een van de volgende drie vormen:

1. Generator: $pat \leftarrow exp$

Ook dit is weer syntactische suiker

We noemen de elementen achter de verticale streep qualifiers.
Ze hebben een van de volgende drie vormen:

1. Generator: $pat \leftarrow exp$
2. Locale declaratie: **let** *decls*

Ook dit is weer syntactische suiker

We noemen de elementen achter de verticale streep qualifiers. Ze hebben een van de volgende drie vormen:

1. Generator: $pat \leftarrow exp$
2. Locale declaratie: **let** *decls*
3. Guard: *exp*

De vertaling naar "Puur Haskell"

De vertaling van de lijst-comprehensies is in het Haskell rapport gedefiniëerd, met inductie over de lijst van qualifiers:

$$[e \mid \mathit{True}] \Rightarrow [e]$$

De vertaling naar "Puur Haskell"

De vertaling van de lijst-comprehensies is in het Haskell rapport gedefiniëerd, met inductie over de lijst van qualifiers:

$$
\begin{array}{ll}
[e \mid \mathit{True}] & \Rightarrow [e] \\
[e \mid q] & \Rightarrow [e \mid q, \mathit{True}]
\end{array}
$$

De vertaling naar "Puur Haskell"

De vertaling van de lijst-comprehensies is in het Haskell rapport gedefiniëerd, met inductie over de lijst van qualifiers:

$$
\begin{array}{lll}
[e \mid \mathit{True}] & \Rightarrow & [e] \\
[e \mid q] & \Rightarrow & [e \mid q, \mathit{True}] \\
[e \mid b, Q] & \Rightarrow & \textbf{if } b \textbf{ then } [e \mid Q] \textbf{ else } [\,]
\end{array}
$$

De vertaling naar "Puur Haskell"

De vertaling van de lijst-comprehensies is in het Haskell rapport gedefiniëerd, met inductie over de lijst van qualifiers:

$$
\begin{array}{lll}
[e \mid \mathit{True}] & \Rightarrow & [e] \\
[e \mid q] & \Rightarrow & [e \mid q, \mathit{True}] \\
[e \mid b, Q] & \Rightarrow & \textbf{if } b \textbf{ then } [e \mid Q] \textbf{ else } [\,] \\
[e \mid p \leftarrow l, Q] & \Rightarrow & \textbf{let } ok\ p = [\,e \mid Q] \\
 & & \qquad ok\ \_ = [\,] \\
 & & \textbf{in}\ \ concat\ (map\ ok\ l)
\end{array}
$$

De vertaling naar "Puur Haskell"

De vertaling van de lijst-comprehensies is in het Haskell rapport gedefiniëerd, met inductie over de lijst van qualifiers:

$$
\begin{array}{lll}
[e \mid \mathit{True}] & \Rightarrow & [e] \\
[e \mid q] & \Rightarrow & [e \mid q, \mathit{True}] \\
[e \mid b, Q] & \Rightarrow & \textbf{if}\ b\ \textbf{then}\ [e \mid Q]\ \textbf{else}\ [\,] \\
[e \mid p \leftarrow l, Q] & \Rightarrow & \textbf{let}\ \mathit{ok}\ p = [\,e \mid Q\,] \\
 & & \phantom{\textbf{let}}\ \mathit{ok}\ \_ = [\,] \\
 & & \textbf{in}\ \ \mathit{concat}\ (\mathit{map}\ \mathit{ok}\ l) \\
[e \mid \textbf{let}\ \mathit{decls}, Q] & \Rightarrow & \textbf{let}\ \mathit{decls}\ \textbf{in}\ [e \mid Q]
\end{array}
$$

De vertaling naar "Puur Haskell"

De vertaling van de lijst-comprehensies is in het Haskell rapport gedefiniëerd, met inductie over de lijst van qualifiers:

$$
\begin{array}{lll}
[e \mid \mathit{True}] & \Rightarrow & [e] \\
[e \mid q] & \Rightarrow & [e \mid q, \mathit{True}] \\
[e \mid b, Q] & \Rightarrow & \mathbf{if}\ b\ \mathbf{then}\ [e \mid Q]\ \mathbf{else}\ [\,] \\
[e \mid p \leftarrow l, Q] & \Rightarrow & \mathbf{let}\ \mathit{ok}\ p = [\,e \mid Q\,] \\
 & & \qquad \mathit{ok}\ \_ = [\,] \\
 & & \mathbf{in}\ \ \mathit{concat}\ (\mathit{map}\ \mathit{ok}\ l) \\
[e \mid \mathbf{let}\ \mathit{decls}, Q] & \Rightarrow & \mathbf{let}\ \mathit{decls}\ \mathbf{in}\ [e \mid Q]
\end{array}
$$

Door het patroon links om te schrijven naar het patroon rechts ontstaat stap voor stap een Haskell expressie zonder lijstcomprehensies.

Voorbeeld

We laten in een aantal stappen zien hoe de volgende expressie uitgedrukt kan worden in “simple haskell”.

- $[(x, y) \mid x \leftarrow [1 \mathinner{..} 5], \mathit{even}\ x, y \leftarrow [1 \mathinner{..} x]]$

Voorbeeld

$$[(x,y) \mid x \leftarrow [1\mathinner{\ldotp\ldotp}5], even\ x, y \leftarrow [1\mathinner{\ldotp\ldotp}x]]$$

De eerste qualifier is een generator:

$$\begin{array}{ll} \textbf{let} & ok\ x = [(x,y) \mid even\ x, y \leftarrow [1\mathinner{\ldotp\ldotp}x]] \\ & ok\ \_ = [\,] \\ \textbf{in} & concat\ (map\ ok\ [1\mathinner{\ldotp\ldotp}5]) \end{array}$$

Voorbeeld

$$
\begin{array}{l}
\textbf{let}\ ok\ x = [(x,y) \mid even\ x, y \leftarrow [1\,..\,x]] \\
\quad\ \ ok\ \_ = [\,] \\
\textbf{in}\ \ concat\ (map\ ok\ [1\,..\,5])
\end{array}
$$

De volgende qualifier is een guard:

$$
\begin{array}{l}
\textbf{let}\ ok\ x = \textbf{if}\quad\ even\ x \\
\qquad\qquad\quad\ \ \textbf{then}\ [(x,y) \mid y \leftarrow [1\,..\,x]] \\
\qquad\qquad\quad\ \ \textbf{else}\ \ [\,] \\
\quad\ \ ok\ \_ = [\,] \\
\textbf{in}\ \ concat\ (map\ ok\ [1\,..\,5])
\end{array}
$$

Voorbeeld

$$
\begin{array}{l}
\textbf{let}\ ok\ x = \textbf{if}\quad even\ x \\
\qquad\qquad\quad \textbf{then}\ [(x,y) \mid y \leftarrow [1\mathinner{\ldotp\ldotp}x]] \\
\qquad\qquad\quad \textbf{else}\ \ [\,] \\
\quad\ \ ok\ \_ = [\,] \\
\textbf{in}\ \ concat\ (map\ ok\ [1\mathinner{\ldotp\ldotp}5])
\end{array}
$$

We hebben nu nog maar een enkele qualifier:

$$
\begin{array}{l}
\textbf{let}\ ok\ x = \textbf{if}\quad even\ x \\
\qquad\qquad\quad \textbf{then}\ [(x,y) \mid y \leftarrow [1\mathinner{\ldotp\ldotp}x], True] \\
\qquad\qquad\quad \textbf{else}\ \ [\,] \\
\quad\ \ ok\ \_ = [\,] \\
\textbf{in}\ \ concat\ (map\ ok\ [1\mathinner{\ldotp\ldotp}5])
\end{array}
$$

Voorbeeld

$$
\begin{array}{l}
\textbf{let}\ ok\ x = \textbf{if}\quad even\ x \\
\qquad\qquad\quad \textbf{then}\ [(x,y) \mid y \leftarrow [1\,..\,x], True] \\
\qquad\qquad\quad \textbf{else}\ \ [\,] \\
\quad\ \ ok\ \_ = [\,] \\
\textbf{in}\ \ concat\ (map\ ok\ [1\,..\,5])
\end{array}
$$

We werken de volgende generator weg:

$$
\begin{array}{l}
\textbf{let}\ ok\ x = \textbf{if}\quad even\ x \\
\qquad\qquad\quad \textbf{then}\ \textbf{let}\ ok\ y = [(x,y) \mid True] \\
\qquad\qquad\qquad\qquad\quad\ \ ok\ \_ = [\,] \\
\qquad\qquad\qquad\quad\ \ \textbf{in}\ \ concat\ (map\ ok\ [1\,..\,x]) \\
\qquad\qquad\quad \textbf{else}\ \ [\,] \\
\quad\ \ ok\ \_ = [\,] \\
\textbf{in}\ \ concat\ (map\ ok\ [1\,..\,5])
\end{array}
$$

Voorbeeld

```
let ok x = if   even x
           then let ok y = [(x,y) | True]
                    ok _ = []
                in  concat (map ok [1..x])
           else []
    ok _ = []
in  concat (map ok [1..5])
```

Vervang de *True*:

```
let ok x = if   even x
           then let ok y = [(x,y)]
                    ok _ = []
                in  concat (map ok [1..x])
           else []
    ok _ = []
in  concat (map ok [1..5])
```

Voorbeeld

$$
\begin{array}{l}
\textbf{let}\ ok\ x = \textbf{if}\quad even\ x \\
\qquad\qquad\quad \textbf{then let}\ ok\ y = [(x,y)] \\
\qquad\qquad\qquad\qquad\quad ok\ \_ = [\,] \\
\qquad\qquad\qquad\quad\ \ \textbf{in}\ \ concat\ (map\ ok\ [1\mathinner{..}x]) \\
\qquad\qquad\quad \textbf{else}\ \ [\,] \\
\quad\ \ ok\ \_ = [\,] \\
\textbf{in}\ \ concat\ (map\ ok\ [1\mathinner{..}5])
\end{array}
$$

In ons voorbeeld slagen patronen altijd:

$$
\begin{array}{l}
\textbf{let}\ ok\ x = \textbf{if}\quad even\ x \\
\qquad\qquad\quad \textbf{then let}\ ok\ y = [(x,y)] \\
\qquad\qquad\qquad\quad\ \ \textbf{in}\ \ concat\ (map\ ok\ [1\mathinner{..}x]) \\
\qquad\qquad\quad \textbf{else}\ \ [\,] \\
\textbf{in}\ \ concat\ (map\ ok\ [1\mathinner{..}5])
\end{array}
$$

Rijtjes getallen

Afkortingen voor rijtjes getallen (*Num*'s) worden als volgt geïnterpreteerd:

$$
\begin{array}{ll}
[e1\,..] & \Rightarrow \mathit{enumFrom}\ e1 \\
[e1, e2\,..] & \Rightarrow \mathit{enumFromThen}\ e1\ e2 \\
[e1\,..\,e3] & \Rightarrow \mathit{enumFromTo}\ e1\ e3 \\
[e1, e2\,..\,e3] & \Rightarrow \mathit{enumFromThenTo}\ e1\ e2\ e3
\end{array}
$$

waarbij *enumFrom*, *enumFromThen*, *enumFromTo*, and *enumFromThenTo* gedefinieerd zijn in de prelude.

Rijtjes getallen

Afkortingen voor rijtjes getallen (*Num*'s) worden als volgt geïnterpreteerd:

$$\begin{array}{ll} [e1\,..] & \Rightarrow \mathit{enumFrom}\ e1 \\ [e1,e2\,..] & \Rightarrow \mathit{enumFromThen}\ e1\ e2 \\ [e1\,..\,e3] & \Rightarrow \mathit{enumFromTo}\ e1\ e3 \\ [e1,e2\,..\,e3] & \Rightarrow \mathit{enumFromThenTo}\ e1\ e2\ e3 \end{array}$$

waarbij *enumFrom*, *enumFromThen*, *enumFromTo*, and *enumFromThenTo* gedefinieerd zijn in de prelude. Voorbeeld:

```
Programs> [1,4..12]
[1,4,7,10]
Programs> [12,9 ..1]
[12,9,6,3]
```

enumFromThenTo

We moeten bij het ingewikkeldste geval er rekening mee houden dat we zowel "omhoog" als "naar beneden" kunnen aftellen:

$$
\begin{aligned}
&\mathit{enumFromByTo} :: \mathit{Int} \to \mathit{Int} \to \mathit{Int} \to [\mathit{Int}] \\
&\mathit{enumFromByTo}\ x\ y\ z = \mathit{takeWhile}\ \mathit{stop}\ (\mathit{iterate}\ \mathit{next}\ x) \\
&\textbf{where}\ \mathit{stop} \mid x \leqslant y = (\leqslant z) \\
&\qquad\qquad\quad \mid \mathit{otherwise} = (\geqslant z) \\
&\qquad\quad\ \mathit{next} = (+)\ (y - x)
\end{aligned}
$$

Tuples

We hebben ze, zonder er heel moeilijk over te doen, al wel ens gebruikt, maar voor de volledigheid vermelden we:

Haskell kent tuples

Voorbeelden van tuples zijn:

`(1,'a')`	een tupel met als elementen de integer 1 en het character `'a'`;
`("aap",` True`,2)`	een tupel met drie elementen: de string `"aap"`, de boolean True en het getal 2;
$([1,2], \mathit{sqrt})$	een tupel met twee elementen: de lijst integers $[1,2]$, en de float-naar-float functie sqrt;
$(1,(2,3))$	een tupel met twee elementen: het getal 1, en het tupel van de getallen 2 en 3.

De bijbehorende types zijn

$$
\begin{array}{ll}
(1, \texttt{'a'}) & :: (\mathit{Int}, \mathit{Char}) \\
(\texttt{"aap"}, \mathit{True}, 2) & :: ([\mathit{Char}], \mathit{Bool}, \mathit{Int}) \\
(1, (2, 3)) & :: (\mathit{Int}, (\mathit{Int}, \mathit{Int}))
\end{array}
$$

Functies met meerdere resultaten ..

Worden gesimuleerd met functies die een tuple opleveren:

```
Programs> splitAt 4 "haskell"
("hask","ell")
```

Functies met meerdere resultaten ..

Worden gesimuleerd met functies die een tuple opleveren:

```
Programs> splitAt 4 "haskell"
("hask","ell")
```

De bijbehorende code is:

$$
\begin{aligned}
&\mathit{splitAt} :: \quad \mathit{Int} \rightarrow [a] \rightarrow ([a],[a]) \\
&\mathit{splitAt}\ n\ \mathit{xs} = (\mathit{take}\ n\ \mathit{xs}, \mathit{drop}\ n\ \mathit{xs})
\end{aligned}
$$

Functies met meerdere resultaten ..

Worden gesimuleerd met functies die een tuple opleveren:

```
Programs> splitAt 4 "haskell"
("hask","ell")
```

De bijbehorende code is:

$$
\begin{aligned}
&\mathit{splitAt} :: \quad Int \to [a] \to ([a],[a]) \\
&\mathit{splitAt}\ n\ xs = (\mathit{take}\ n\ xs, \mathit{drop}\ n\ xs)
\end{aligned}
$$

of beter:

$$
\begin{aligned}
&\mathit{splitAt}\ 0\ xs = \quad ([\,], xs) \\
&\mathit{splitAt}\ n\ [\,] = \quad ([\,],[\,]) \\
&\mathit{splitAt}\ n\ (x : xs) = (x : ys, zs) \\
&\qquad\qquad \textbf{where}\ (ys, zs) = \mathit{splitAt}\ (n-1)\ xs
\end{aligned}
$$

Type Definities

We geven ingewikkelde types een naam. Dus stel we hebben een stuk code:

$$
\begin{aligned}
&afstand :: (Float, Float) \rightarrow Float \\
&verschil :: (Float, Float) \rightarrow (Float, Float) \rightarrow Float
\end{aligned}
$$

Dan wordt het vooral bij hogere-orde functies wat lastiger te overzien:

$$
\begin{aligned}
opp\_veelhoek \;&::\; [(Float, Float)] \rightarrow Float \\
transf\_veelhoek \;&::\; ((Float, Float) \rightarrow (Float, Float)) \\
&\quad \rightarrow [(Float, Float)] \rightarrow [(Float, Float)]
\end{aligned}
$$

In zo'n geval komt een type-definitie van pas.

Type Definities

We geven ingewikkelde types een naam. Met type-definities:

$$\begin{aligned}
&\textbf{type}\ \mathit{Punt} &&= (\mathit{Float}, \mathit{Float}) \\
&\textbf{type}\ \mathit{Veelhoek} &&= [\mathit{Punt}] \\
&\textbf{type}\ \mathit{Trafo} &&= \mathit{Punt} \to \mathit{Punt}
\end{aligned}$$

krijgen we nu:

$$\begin{aligned}
&\mathit{afstand} &&:: \mathit{Punt} \to \mathit{Float} \\
&\mathit{verschil} &&:: \mathit{Punt} \to \mathit{Punt} \to \mathit{Float} \\
&\mathit{opp_veelhoek} &&:: \mathit{Veelhoek} \to \mathit{Float} \\
&\mathit{transf_veelhoek} &&:: \mathit{Trafo} \to \mathit{Veelhoek} \to \mathit{Veelhoek}
\end{aligned}$$