Universiteit Utrecht

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

College 2010-2011
8. Arrays, Huffman Trees, Profiling

Doaitse Swierstra

Utrecht University

October 11, 2010

Unresolved overloading

Vorige keer probeerde ik:

```
let sr = show.read
```

en kreeg niet de foutmelding die ik verwachtte.

Unresolved overloading

Vorige keer probeerde ik:

```
let sr = show.read
```

en kreeg niet de foutmelding die ik verwachtte. Dit gedrag is GHCi specifiek. Voor het onbekende “tussentype” wordt () gekozen (zie GHC manual 2.4.5):

```
Prelude> let sr = show . read
Prelude> :t sr
sr :: String -> String
Prelude> sr "5"
"*** Exception: Prelude.read: no parse
Prelude> show.read $ "()"
"()"
Prelude>
```

Unresolved overloading

Echter:

> **module** *Test* **where**
>
> $sr = show \circ read$

levert:

```
Ambiguous type variable ‘b’ in the constraints:
 ‘Read b’
   arising from a use of ‘read’
     at Test.hs:3:12-15
 ‘Show b’
   arising from a use of ‘show’
     at Test.hs:3:5-8
    Probable fix: add a type signature that fixes
      these type variable(s)
Failed, modules loaded: none.
```

combs: alle sublijsten van gegeven lengte

Deze functie *combs* heeft, behalve de lijst, ook een getal als parameter:

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

We geven een inefficiënte specificatie:

$$\begin{aligned} &combs\ n\ xs = filter\ goed\ (subs\ xs) \\ &\quad \textbf{where} \\ &\qquad goed\ xs = length\ xs \equiv n \end{aligned}$$

Hoe beginnen we?

We gaan op jacht naar een definite van de volgende vorm (afbreken van beide argumenten):

$$
\begin{aligned}
combs\ 0\ xs &= \ldots \\
combs\ n\ [\,] &= \ldots \\
combs\ n\ (x : xs) &= \ldots
\end{aligned}
$$

...

We bekijken eerst de eenvoudige gevallen:

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

In het laatste geval kunnen we twee dingen doen:

...

We bekijken eerst de eenvoudige gevallen:

$$\begin{aligned} combs\ 0\ xs &= [[\,]] \\ combs\ n\ [\,] &= [\,] \\ combs\ n\ (x : xs) &= \ldots \end{aligned}$$

In het laatste geval kunnen we twee dingen doen:

1. het element x wel meenemen, en dan op jacht gaan naar alle combinaties met $n - 1$ elementen

...

We bekijken eerst de eenvoudige gevallen:

$$\begin{aligned} combs\ 0\ xs &= [[\,]] \\ combs\ n\ [\,] &= [\,] \\ combs\ n\ (x : xs) &= \ldots \end{aligned}$$

In het laatste geval kunnen we twee dingen doen:

1. het element x wel meenemen, en dan op jacht gaan naar alle combinaties met $n - 1$ elementen
2. de x niet gebruiken, maar dan moeten de n elementen uit de xs gekozen worden:

...

We bekijken eerst de eenvoudige gevallen:

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

In het laatste geval kunnen we twee dingen doen:

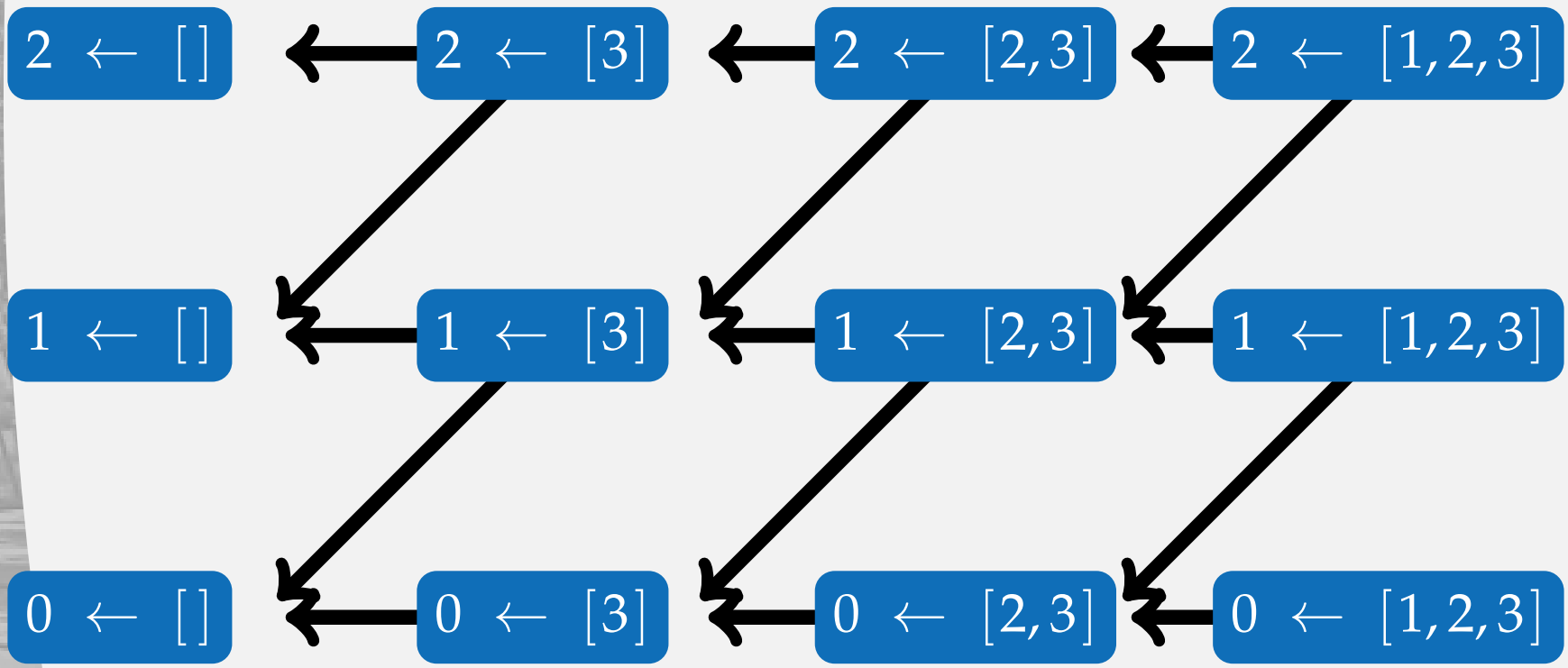
1. het element x wel meenemen, en dan op jacht gaan naar alle combinaties met $n-1$ elementen
2. de x niet gebruiken, maar dan moeten de n elementen uit de xs gekozen worden:

$$
\begin{aligned}
&\mathit{combs}\ 0\ \mathit{xs} &&= [[\,]]\\
&\mathit{combs}\ n\ [\,] &&= [\,]\\
&\mathit{combs}\ n\ (x : \mathit{xs}) &&= \mathit{map}\ (x:)\ (\mathit{combs}\ (n-1)\ \mathit{xs})\\
&&&\quad +\!\!+\quad \mathit{combs}\ n\quad \mathit{xs}
\end{aligned}
$$

Hier ligt weer inëfficientie op de loer

$$
\begin{aligned}
& combs\ (n+2)\ (x:y:ys) \\
& = \\
& map\ (x:)\ (combs\ (n+1)\ (y:ys)) +\!\!+ combs\ (n+2)\ (y:ys) \\
& = \\
& map\ (x:)\ ((map\ (y:)\ (combs\ n\ ys) \quad +\!\!+ combs\ (n+1)\ ys)) \\
& \quad +\!\!+ \\
& combs\ (n+2)\ (y:ys) \\
& = \\
& \quad map\ (x:)\ (map\ (y:)\ (combs\ n\ ys)) \\
& +\!\!+ map\ (x:)\ (combs\ (n+1)\ ys) \\
& +\!\!+ map\ (y:)\ (combs\ (n+1)\ ys) \\
& +\!\!+ \quad (combs\ (n+2)\ ys)
\end{aligned}
$$

De data flow



Observaties

- de waarden op de onderste regel zijn allemaal $[[]]$
- de rest van de waarden in de eerste kolom zijn allemaal $[]$
- we kunnen elke volgende kolom uit de vorige berekenen

Observaties

- de waarden op de onderste regel zijn allemaal $[[]]$
- de rest van de waarden in de eerste kolom zijn allemaal $[]$
- we kunnen elke volgende kolom uit de vorige berekenen

De functie *allcombs* berekent een hele kolom:

$$
\begin{aligned}
&allcombs :: [a] \rightarrow [[[a]]] \\
&allcombs\ [\,] = [[\,]] : repeat\ [\,] \\
&allcombs\ (x : xs) = \textbf{let}\ previous = allcombs\ xs \\
&\qquad\qquad\qquad\qquad \textbf{in}\ [[\,]] : zipWith\ (\lambda l\ r \rightarrow map\ (x:)\ l +\!\!+ r) \\
&\qquad\qquad\qquad\qquad\qquad\qquad (tail\ previous) \\
&\qquad\qquad\qquad\qquad\qquad\qquad previous
\end{aligned}
$$

Observaties

- de waarden op de onderste regel zijn allemaal $[[]]$
- de rest van de waarden in de eerste kolom zijn allemaal $[]$
- we kunnen elke volgende kolom uit de vorige berekenen

De functie *allcombs* berekent een hele kolom:

$$
\begin{aligned}
&allcombs :: [a] \rightarrow [[[a]]] \\
&allcombs\ [\,] = [[\,]] : repeat\ [\,] \\
&allcombs\ (x : xs) = \textbf{let}\ previous = allcombs\ xs \\
&\qquad\qquad\qquad\qquad \textbf{in}\ [[\,]] : zipWith\ (\lambda l\ r \rightarrow map\ (x:)\ l +\!\!+ r) \\
&\qquad\qquad\qquad\qquad\qquad\qquad (tail\ previous) \\
&\qquad\qquad\qquad\qquad\qquad\qquad previous
\end{aligned}
$$

We hebben zo een kwadratische berekening met een exponentiëel resultaat!!

Voorbeeldgebruik

```
Programs> allcombs [1,2,3,4] !! 0
[[]]
Programs> allcombs [1,2,3,4] !! 1
[[4],[3],[2],[1]]
Programs> allcombs [1,2,3,4] !! 2
[[3,4],[2,4],[2,3],[1,4],[1,3],[1,2]]
Programs> allcombs [1,2,3,4] !! 3
[[2,3,4],[1,3,4],[1,2,4],[1,2,3]]
Programs> allcombs [1,2,3,4] !! 4
[[1,2,3,4]]
Programs> allcombs [1,2,3,4] !! 5
[]
```

Oude resultaten kunnen we ook bewaren:

```
allcombs' :: [a]→[[[[a]]]]
allcombs' []       = [[[]] : repeat []]
allcombs' (x : xs) = let previous = allcombs' xs
                         l        = last previous
                     in  previous ++
                         [[[]] :  zipWith (λl r→map (x:) l ++ r)
                                          (tail l)
                                          l
                         ]
```

Universiteit Utrecht

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

Arrays

Haskell kent ook arrays. Je moet daarvoor de module *Array* importeren. We geven een paar voorbeelden:

$$
\begin{aligned}
&array :: (Ix\ a) \Rightarrow (a,a) \rightarrow [(a,b)] \rightarrow Array\ a\ b \\
&demo1 = array\ (1,10) \\
&\qquad [(i, i*i) \mid i \leftarrow [1..10]] \\
&demo2 = array\ (1,10)\ ((1,1) : \\
&\qquad [(i, i * demo2\,!\,(i-1)) \mid i \leftarrow [2..10]]) \\
&fibs \quad = array\ (1,10)\ ((1,1) : (2,1) : \\
&\qquad [(i, fibs\,!\,(i-1) + fibs\,!\,(i-2)) \mid i \leftarrow [3..10]])
\end{aligned}
$$

```
*Programs> fibs!5
5
```

Arrays ...

Merk op:

1. het eerste argument van *array* is een paar van indices

Arrays ...

Merk op:

1. het eerste argument van *array* is een paar van indices
2. het tweede argument is een lijst van paren: (index, waarde)

Arrays ...

Merk op:

1. het eerste argument van *array* is een paar van indices
2. het tweede argument is een lijst van paren: (index, waarde)
3. elementen mogen gedefinieerd worden in termen van andere argumenten.

Arrays ...

Merk op:

1. het eerste argument van *array* is een paar van indices
2. het tweede argument is een lijst van paren: (index, waarde)
3. elementen mogen gedefinieerd worden in termen van andere argumenten.

Arrays ...

Merk op:

1. het eerste argument van *array* is een paar van indices
2. het tweede argument is een lijst van paren: (index, waarde)
3. elementen mogen gedefinieerd worden in termen van andere argumenten.

Ook kan:

$$demo3 = array\ ((1,1),(10,10))\ [((i,j),i+j)\mid i \leftarrow [1..10], j \leftarrow [1..10]]$$

Arrays ...

Merk op:

1. het eerste argument van *array* is een paar van indices
2. het tweede argument is een lijst van paren: (index, waarde)
3. elementen mogen gedefinieerd worden in termen van andere argumenten.

Ook kan:

$$demo3 = array\ ((1,1),(10,10))\ [((i,j),i+j)\mid i \leftarrow [1..10], j \leftarrow [1..10]]$$

```
*Programs> demo3 ! (3,5)
8
```

subs revisited

We kunnen nu een alternatieve formulering van *subs* geven.

We definieren een array, waarbij op positie (j, i) staat op welke manieren je i elementen uit de eerste j elementen kunt kiezen:

subs revisited

We kunnen nu een alternatieve formulering van *subs* geven.

We definieren een array, waarbij op positie (j, i) staat op welke manieren je i elementen uit de eerste j elementen kunt kiezen:

$$subs\ l = \textbf{let}\ ll = length\ l$$

subs revisited

We kunnen nu een alternatieve formulering van *subs* geven.

We definieren een array, waarbij op positie (j,i) staat op welke manieren je i elementen uit de eerste j elementen kunt kiezen:

$$
\begin{aligned}
&\mathit{subs}\ l = \textbf{let}\ ll = \mathit{length}\ l \\
&\mathit{result} = \mathit{array}\ ((0,0),(ll,ll)) \\
&\qquad\qquad (\ [((j,0),\ \ [[]]) \mid j \leftarrow [0\,..\,ll]]
\end{aligned}
$$

subs revisited

We kunnen nu een alternatieve formulering van *subs* geven.

We definieren een array, waarbij op positie (j, i) staat op welke manieren je i elementen uit de eerste j elementen kunt kiezen:

$$
\begin{aligned}
&\mathit{subs}\ l = \textbf{let}\ ll = \mathit{length}\ l \\
&\mathit{result} = \mathit{array}\ ((0,0),(ll,ll)) \\
&\qquad\qquad (\ [((j,0),\ \ [[\,]]) \mid j \leftarrow [0\,..\,ll]] \\
&\qquad\qquad +\!\!+\ [((0,i),\ \ [\ \ ]) \mid i \leftarrow [1\,..\,ll]]
\end{aligned}
$$

Universiteit Utrecht

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

subs revisited

We kunnen nu een alternatieve formulering van *subs* geven.

We definieren een array, waarbij op positie (j, i) staat op welke manieren je i elementen uit de eerste j elementen kunt kiezen:

$$
\begin{aligned}
&subs\ l = \textbf{let}\ ll = length\ l \\
&result = array\ ((0,0),(ll,ll)) \\
&\qquad (\ [((j,0),\ \ [[\,]]) \mid j \leftarrow [0\,..\,ll]] \\
&\qquad +\!\!+ [((0,i),\ \ [\ \ ]) \mid i \leftarrow [1\,..\,ll]] \\
&\qquad +\!\!+ [((j,i),\ \ map\ ((l\ !!\ (j-1)):)\ (result\ !\ (j-1,i-1)) \\
&\qquad\qquad\qquad +\!\!+ \\
&\qquad\qquad\qquad\qquad\qquad\qquad result\ !\ (j-1,i\ \ \ \ )) \\
&\qquad\quad \mid j \leftarrow [1\,..\,ll], i \leftarrow [1\,..\,ll] \\
&\qquad\quad] \\
&\qquad)
\end{aligned}
$$

subs revisited

We kunnen nu een alternatieve formulering van *subs* geven.

We definieren een array, waarbij op positie (j, i) staat op welke manieren je i elementen uit de eerste j elementen kunt kiezen:

$$
\begin{array}{l}
subs\ l = \textbf{let}\ ll = length\ l \\
result = array\ ((0,0),(ll,ll)) \\
\qquad (\ [((j,0),\ \ [[]])\mid j \leftarrow [0\,..\,ll]] \\
\qquad +\!\!+\ [((0,i),\ \ [\ \ ])\mid i \leftarrow [1\,..\,ll]] \\
\qquad +\!\!+\ [((j,i),\ \ map\ ((l\ !!\ (j-1)):)\ (result\ !\ (j-1,i-1)) \\
\qquad\qquad\qquad +\!\!+ \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad result\ !\ (j-1,i\ \ \ \ \ \ ))\\
\qquad\qquad\qquad \mid j \leftarrow [1\,..\,ll], i \leftarrow [1\,..\,ll] \\
\qquad\qquad\qquad] \\
\qquad\qquad) \\
\textbf{in}\ result
\end{array}
$$

Universiteit Utrecht

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

subs revisited

We kunnen nu een alternatieve formulering van *subs* geven.

We definieren een array, waarbij op positie (j,i) staat op welke manieren je i elementen uit de eerste j elementen kunt kiezen:

$$
\begin{array}{l}
subs\ l = \textbf{let}\ ll = length\ l \\
result = array\ ((0,0),(ll,ll)) \\
\qquad (\ [((j,0),\ [[]]) \mid j \leftarrow [0\,..\,ll]] \\
\qquad +\!\!+ [((0,i),\ [\ ]) \mid i \leftarrow [1\,..\,ll]] \\
\qquad +\!\!+ [((j,i),\ map\ ((l\ !!\ (j-1)) :)\ (result\ !\ (j-1,i-1)) \\
\qquad\qquad \mid j \leftarrow [1\,..\,ll], i \leftarrow [1\,..\,ll] \\
\qquad\qquad] \\
\qquad) \\
\textbf{in}\ result
\end{array}
$$

```
*Programs> subs [1,2,3,4] ! (4,2)
[[4,3],[4,2],[4,1],[3,2],[3,1],[2,1]]
```

Universiteit Utrecht

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

Huffman Encoding

When transmitting lots of data (e.g. by a fax machine) we want to encode dat as efficiently as possible.

- frequently occurring symbols are assigned short sequences of 0's and 1's

Huffman Encoding

When transmitting lots of data (e.g. by a fax machine) we want to encode dat as efficiently as possible.

- frequently occurring symbols are assigned short sequences of 0's and 1's
- less frequent symbols are assigned longer sequences

Huffman Encoding

When transmitting lots of data (e.g. by a fax machine) we want to encode dat as efficiently as possible.

- frequently occurring symbols are assigned short sequences of 0's and 1's
- less frequent symbols are assigned longer sequences
- we can see where a subsequence that encodes for a single symbols ends in the complete sequence, so we can split the sequence in individual symbols

Huffman Encoding

When transmitting lots of data (e.g. by a fax machine) we want to encode dat as efficiently as possible.

- frequently occurring symbols are assigned short sequences of 0's and 1's
- less frequent symbols are assigned longer sequences
- we can see where a subsequence that encodes for a single symbols ends in the complete sequence, so we can split the sequence in individual symbols
- so, no prefix of an encoding encodes for another symbol.

Example encodings

If we take:

$$t \rightarrow 0$$
$$e \rightarrow 10$$
$$x \rightarrow 11$$

we can encode the word "text" as 010110.

The encoding:

$$t \rightarrow 0$$
$$e \rightarrow 10$$
$$x \rightarrow 1$$

does not fulfill the last property.

Optimality

Given the relative frequencies p_j of symbols j, and choosing an encoding of length l_j we want to minimize:

$$\Sigma_{j=1}^{n} p_j l_j$$

Optimality

Given the relative frequencies p_j of symbols j, and choosing an encoding of length l_j we want to minimize:

$$\Sigma_{j=1}^{n} p_j l_j$$

For the decoding we use a binary tree with symbols in the leaves. A left branch stands for a 1 in the encoding and a right branch for a 0.

```
data Huff a = Fork (Huff a) (Huff a)
            | Leaf a
data Bit    = Zero
            | One
```

Decoding

We can now decode a sequence of *Bit*'s:

```
decode :: Huff a → [Bit] → [a]
decode t = decode'
           where decode' (Leaf v)  xs          = v : decode' t xs
                 decode' (Fork l r) (Zero : xs) =     decode' l xs
                 decode' (Fork l r) (One : xs)  =     decode' r xs
                 decode' _          []          = []
```

Building the tree

The basic idea is:

- the two symbols with the lowest weight share a common prefix

Building the tree

The basic idea is:

- the two symbols with the lowest weight share a common prefix
- and only differ in their last bit

Building the tree

The basic idea is:

- the two symbols with the lowest weight share a common prefix
- and only differ in their last bit
- the weight of a tree is the sum of the weights of its contained symbols

Building the tree

The basic idea is:

- the two symbols with the lowest weight share a common prefix
- and only differ in their last bit
- the weight of a tree is the sum of the weights of its contained symbols
- elements from the two trees with the lowest weight share a common prefix

Building the tree

The basic idea is:

- the two symbols with the lowest weight share a common prefix
- and only differ in their last bit
- the weight of a tree is the sum of the weights of its contained symbols
- elements from the two trees with the lowest weight share a common prefix

Building the tree

The basic idea is:

- the two symbols with the lowest weight share a common prefix
- and only differ in their last bit
- the weight of a tree is the sum of the weights of its contained symbols
- elements from the two trees with the lowest weight share a common prefix

The input is a list of symbols paired with their (relative) number of occurrences:

$$[(c_0, w_0), (c_1, w_1), (c_2, w_2), (c_3, w_3), (c_4, w_4), ...]$$

with the property $w_0 <= w_1 <= w_2 <= w_3 <= w_4 <= ...$

Building the tree

$$
\begin{aligned}
&mkLeaf :: (a, Weight) \rightarrow (Huff\ a, Weight)\\
&mkLeaf\ (s, w) = (Leaf\ s, w)\\
&singleton\ [x] = True\\
&singleton\ \_\ \ = False
\end{aligned}
$$

Building the tree

```
mkLeaf :: (a, Weight) → (Huff a, Weight)
mkLeaf (s, w) = (Leaf s, w)
singleton [x] = True
singleton _   = False
   -- assume input is sorted by weight
combine :: [(Huff a, Weight)] → [(Huff a, Weight)]
combine ((xt, xw) : (yt, yw) : xts)
    =    insert ((Fork xt yt, xw + yw)) xts
```

Building the tree

```
mkLeaf :: (a, Weight) → (Huff a, Weight)
mkLeaf (s, w) = (Leaf s, w)
singleton [x] = True
singleton _   = False
   -- assume input is sorted by weight
combine :: [(Huff a, Weight)] → [(Huff a, Weight)]
combine ((xt, xw) : (yt, yw) : xts)
    =   insert ((Fork xt yt, xw + yw)) xts
insert   x@(tx, wx) ys@(y@(ty, wy) : rest)
           | wx < wy = x : ys
           | _       = y : insert x rest
insert x         []  = [x]
```

Building the tree

```
mkLeaf :: (a, Weight) → (Huff a, Weight)
mkLeaf (s, w) = (Leaf s, w)
singleton [x] = True
singleton _   = False
   -- assume input is sorted by weight
combine :: [(Huff a, Weight)] → [(Huff a, Weight)]
combine ((xt, xw) : (yt, yw) : xts)
    =   insert ((Fork xt yt, xw + yw)) xts
insert  x@(tx, wx) ys@(y@(ty, wy) : rest)
          | wx < wy = x : ys
          | _       = y : insert x rest
insert x          []  = [x]
mkHuff :: [(a, Weight)] → Tree a
mkHuff = fst ∘ head ∘ until singleton combine ∘ map mkLeaf
```

Analysing a sample document

Analyzing a document consists of three steps:

1. sort all the symbols
2. group symbols while computing their weight
3. sort by frequency

Analysing a sample document

Analyzing a document consists of three steps:

1. sort all the symbols
2. group symbols while computing their weight
3. sort by frequency

$$sortBy :: Ord\ b \Rightarrow (a \rightarrow b) \rightarrow [a] \rightarrow [a]$$

Analysing a sample document

Analyzing a document consists of three steps:

1. sort all the symbols
2. group symbols while computing their weight
3. sort by frequency

$$
\begin{aligned}
&sample :: [a] \rightarrow [(a, Weight)] \\
&sample = sortBy\ snd \circ collate \circ map\ occursOnce \circ sortBy\ id \\
&occursOnce\ s = (s, 1) \\
&sortBy :: Ord\ b \Rightarrow (a \rightarrow b) \rightarrow [a] \rightarrow [a] \\
&collate :: Eq\ a \Rightarrow [(a, Weight)] \rightarrow [(a, Weight)] \\
&collate\ [\,] = [\,] \\
&collate\ [x] = [x] \\
&collate\ ((x, wx) : ys@(y : wy) : z) \\
&\quad \mid x \equiv y \quad \rightarrow \quad collate\ ((x, wx + wy) : z) \\
&\quad \mid otherwise \rightarrow (x, wx) : collate\ (ys : z)
\end{aligned}
$$

Encoding a single symbol

When encoding we look for the symbol in the *Huff* tree, while building the path to the *Leaf* that contains the symbol s to be encoded:

```
lookup (Leaf x) s  | s ≡ x      = Just []
                   | otherwise = Nothing
lookup (Tree l r) s = case lookup l s of
                        Just p    → Just (One : p)
                        Nothing   → case lookup r a of
                                      Just p   → Just (Zero : p)
                                      Nothing  → Nothing
```

Control.Applicative

The function $(<|>)$ is actually defined in the module *Control.Applicative*:

```
class (Applicative f) ⇒ Alternative f where
  ...
  (<|>) :: f a → f a → f a
        -- Defined in Control.Applicative
infixl 3 <|>
```

Control.Applicative

The function $(<|>)$ is actually defined in the module *Control.Applicative*:

```
class (Applicative f) ⇒ Alternative f where
  ...
  (<|>) :: f a → f a → f a
        -- Defined in Control.Applicative
infixl 3 <|>
```

and the type *Maybe* is an instance of *Alternative*:

```
instance Alternative Maybe where
Just v <|> _ = Just v
_      <|> r = r
```

Control.Applicative

The function $(<|>)$ is actually defined in the module *Control.Applicative*:

```
class (Applicative f) ⇒ Alternative f where
  ...
  (<|>) :: f a → f a → f a
        -- Defined in Control.Applicative
infixl 3 <|>
```

and the type *Maybe* is an instance of *Alternative*:

```
instance Alternative Maybe where
Just v <|> _ = Just v
_      <|> r = r
```

```
lookup (Tree l r) s =      (One:) <$> lookup s l
                      <|> (Zero:) <$> lookup s r
```

Encoding a document

- analyze the document and build the *Huff a* tree
- use the tree to encode all symbols
- concatenate the encodings

Encoding a document

- analyze the document and build the $Huff\ a$ tree
- use the tree to encode all symbols
- concatenate the encodings

$$
\begin{aligned}
&encode :: [a] \rightarrow [Bit] \\
&encode\ ss = concat \circ map\ (lookup\ huffTree) \quad \$\ ss \\
&\qquad \textbf{where}\ huffTree = mkHuff \circ sample\ \$\ ss
\end{aligned}
$$

lets over (in)efficiëncy

We hebben inmiddels gezien dat Haskell:

- korte formuleringen toe laat
- die echter ook wel eens inefficiënt kunnen zijn.

Vraag

Hoe kom er nou achter waar de tijd gaat zitten?

lets over (in)efficiëncy

We hebben inmiddels gezien dat Haskell:

- korte formuleringen toe laat
- die echter ook wel eens inefficiënt kunnen zijn.

Vraag

Hoe kom er nou achter waar de tijd gaat zitten?

Antwoord:

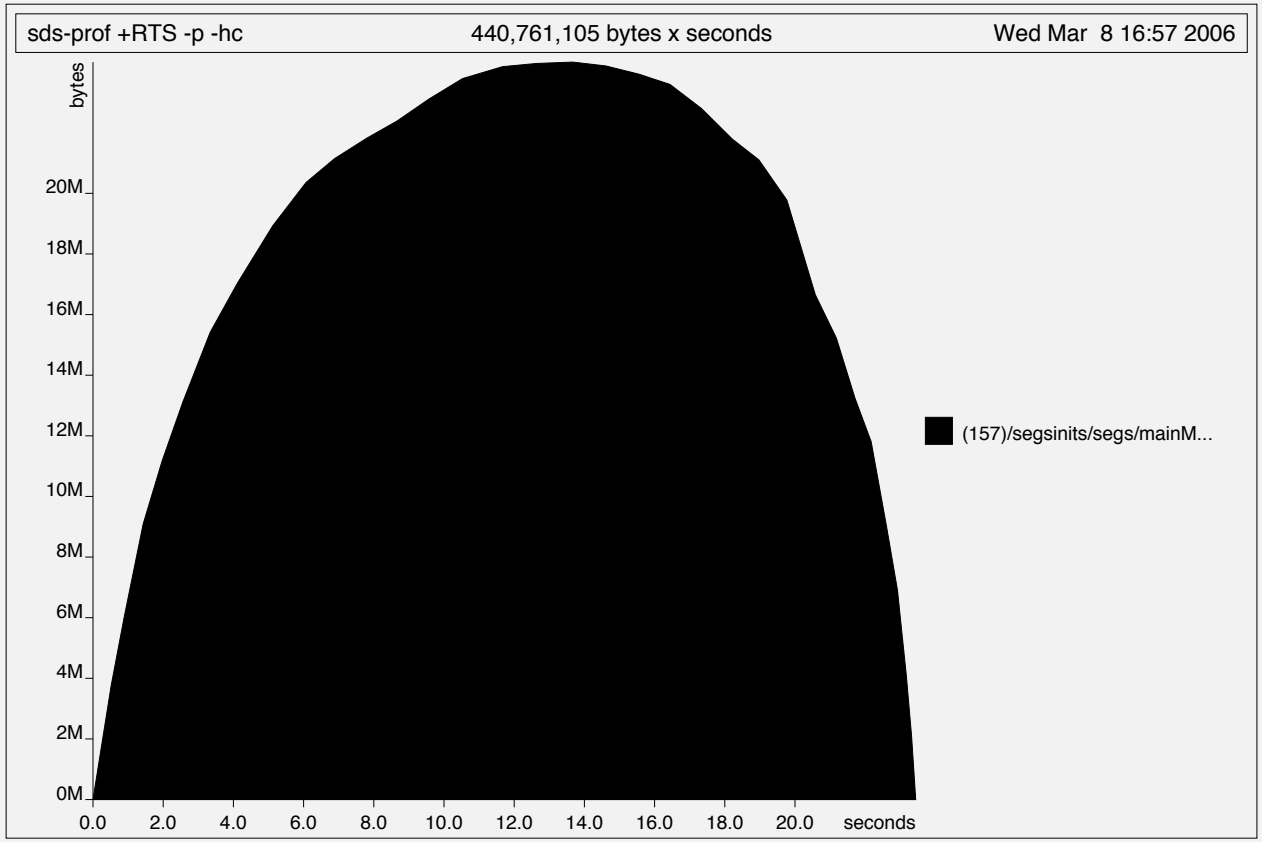
Gebruik profiling

Voorbeeld

```
module Main where

import Data.List (inits, tails)

segsinits []       = ([[]], [[]])
segsinits (x : xs) = let (segsxs, initsxs) = segsinits xs
                         newinits          = map (x:) initsxs
                     in  (segsxs ++ newinits
                         , [] : newinits
                         )

segs = fst ∘ segsinits

pointfree = let  p = ¬ ∘ null
                 next = filter p ∘ map tail ∘ filter p
            in   concat ∘ takeWhile p ∘ iterate next ∘ inits

listcomp xs = [] : [t | i ← inits xs, t ← tails i, ¬ (null t)]
main = print (length (concat (listcomp [1 :: Int .. 300])))
```

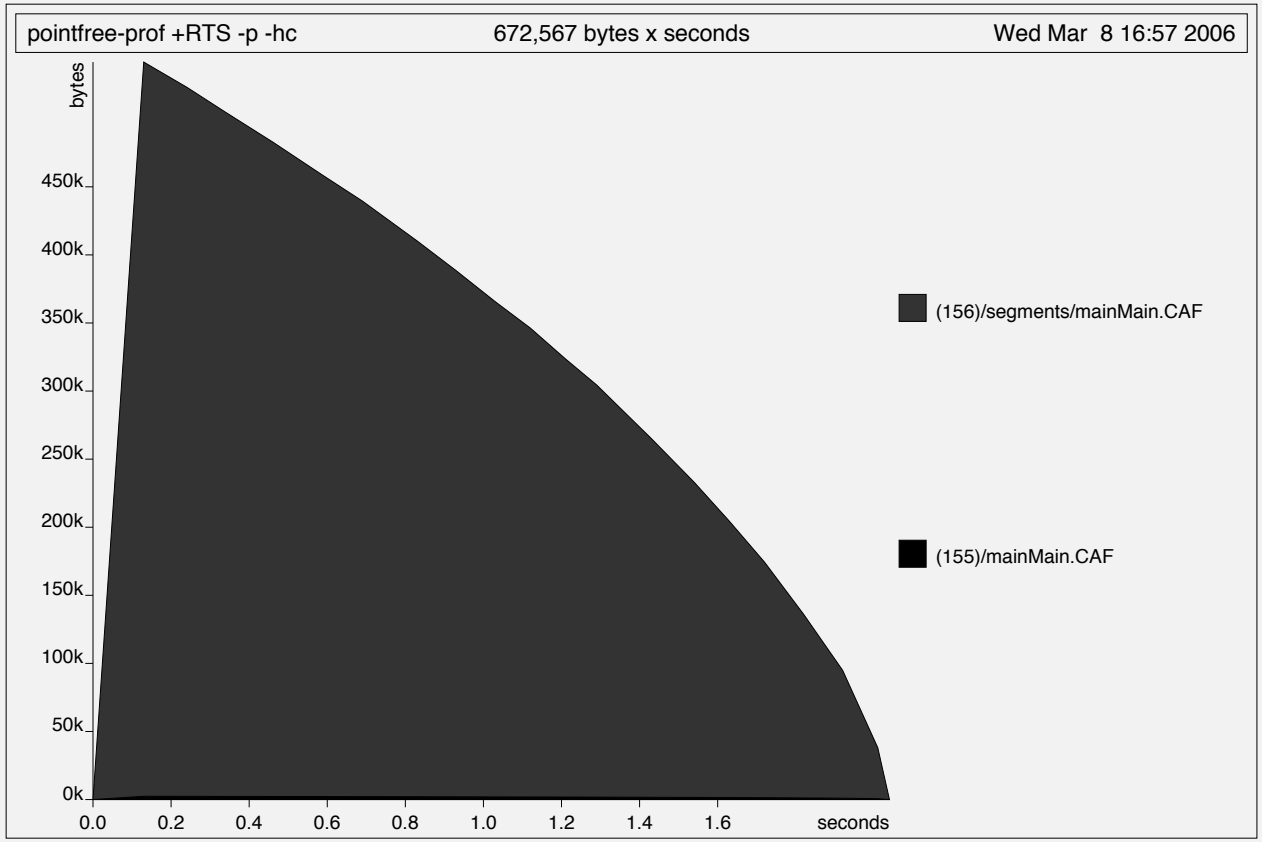
Heap profile

Met gebruik van *segsinits*:



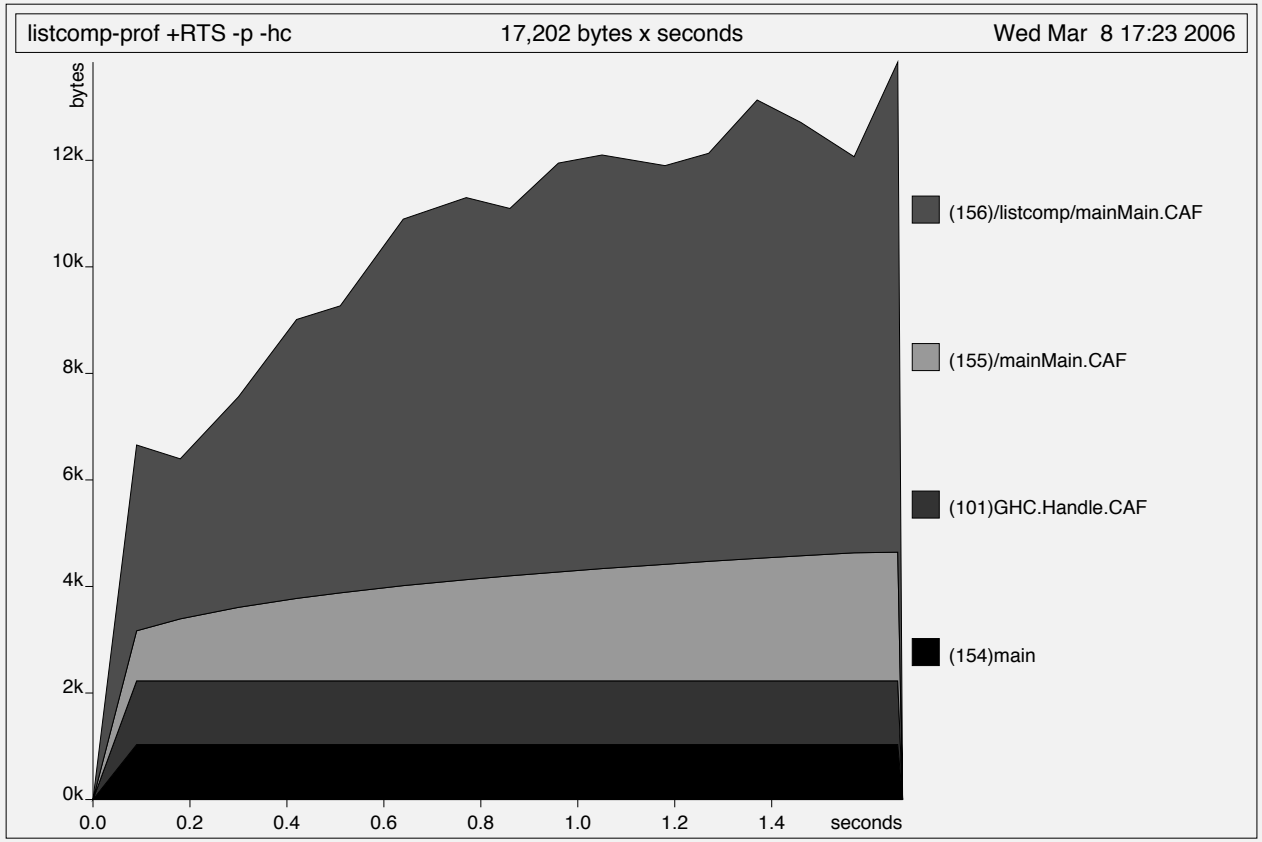
Heap profile

Met gebruik van *pointfree*:



Heap profile

Met gebruik van *listcomp*:



Hoe doe je dat?

```
prompt> ghc -prof -auto-all -o listcomp-prof
            -O2 Segments.hs
prompt> ./listcomp-prof +RTS -hc -p
4545100
prompt> hp2ps listcomp-prof.hp
```

Case study: Calendar

- part of the original unix distribution
- used by Bird & Wadler in their book
- example program in the Hugs distribution
- example program in the Helium distribution

Description of Cal

The Helium session:

```
Calendar> :l /Helium/demo/Calendar.hs
Calendar> main
See calendar for which year? 2006
```

geeft:

```
+----------------------+----------------------+----------------------+
|     January 2006     |    February 2006     |      March 2006      |
| su mo tu we th fr sa | su mo tu we th fr sa | su mo tu we th fr sa |
|  1  2  3  4  5  6  7 |           1  2  3  4 |           1  2  3  4 |
|  8  9 10 11 12 13 14 |  5  6  7  8  9 10 11 |  5  6  7  8  9 10 11 |
| 15 16 17 18 19 20 21 | 12 13 14 15 16 17 18 | 12 13 14 15 16 17 18 |
| 22 23 24 25 26 27 28 | 19 20 21 22 23 24 25 | 19 20 21 22 23 24 25 |
| 29 30 31             | 26 27 28             | 26 27 28 29 30 31    |
|                      |                      |                      |
+----------------------+----------------------+----------------------+
|      April 2006      |       May 2006       |      June 2006       |
| su mo tu we th fr sa | su mo tu we th fr sa | su mo tu we th fr sa |
|                    1 |     1  2  3  4  5  6 |           1  2  3    |
|  2  3  4  5  6  7  8 |  7  8  9 10 11 12 13 |  4  5  6  7  8  9 10 |
|  9 10 11 12 13 14 15 | 14 15 16 17 18 19 20 | 11 12 13 14 15 16 17 |
| 16 17 18 19 20 21 22 | 21 22 23 24 25 26 27 | 18 19 20 21 22 23 24 |
| 23 24 25 26 27 28 29 | 28 29 30 31          | 25 26 27 28 29 30    |
| 30                   |                      |                      |
+----------------------+----------------------+----------------------+
|      July 2006       |     August 2006      |    September 2006    |
| su mo tu we th fr sa | su mo tu we th fr sa | su mo tu we th fr sa |
|                    1 |        1  2  3  4  5 |                1  2   |
|  2  3  4  5  6  7  8 |  6  7  8  9 10 11 12 |  3  4  5  6  7  8  9 |
|  9 10 11 12 13 14 15 | 13 14 15 16 17 18 19 | 10 11 12 13 14 15 16 |
| 16 17 18 19 20 21 22 | 20 21 22 23 24 25 26 | 17 18 19 20 21 22 23 |
| 23 24 25 26 27 28 29 | 27 28 29 30 31       | 24 25 26 27 28 29 30 |
| 30 31                |                      |                      |
+----------------------+----------------------+----------------------+
|     October 2006     |    November 2006     |    December 2006     |
| su mo tu we th fr sa | su mo tu we th fr sa | su mo tu we th fr sa |
|  1  2  3  4  5  6  7 |           1  2  3  4 |                1  2   |
|  8  9 10 11 12 13 14 |  5  6  7  8  9 10 11 |  3  4  5  6  7  8  9 |
| 15 16 17 18 19 20 21 | 12 13 14 15 16 17 18 | 10 11 12 13 14 15 16 |
| 22 23 24 25 26 27 28 | 19 20 21 22 23 24 25 | 17 18 19 20 21 22 23 |
| 29 30 31             | 26 27 28 29 30       | 24 25 26 27 28 29 30 |
|                      |                      | 31                   |
+----------------------+----------------------+----------------------+
```

```
+----------------------+----------------------+----------------------+
|     January 2006     |    February 2006     |      March 2006      |
| su mo tu we th fr sa | su mo tu we th fr sa | su mo tu we th fr sa |
|  1  2  3  4  5  6  7 |           1  2  3  4 |           1  2  3  4 |
|  8  9 10 11 12 13 14 |  5  6  7  8  9 10 11 |  5  6  7  8  9 10 11 |
| 15 16 17 18 19 20 21 | 12 13 14 15 16 17 18 | 12 13 14 15 16 17 18 |
| 22 23 24 25 26 27 28 | 19 20 21 22 23 24 25 | 19 20 21 22 23 24 25 |
| 29 30 31             | 26 27 28             | 26 27 28 29 30 31    |
|                      |                      |                      |
+----------------------+----------------------+----------------------+
|      April 2006      |       May 2006       |      June 2006       |
| su mo tu we th fr sa | su mo tu we th fr sa | su mo tu we th fr sa |
|                    1 |     1  2  3  4  5  6 |           1  2  3    |
|  2  3  4  5  6  7  8 |  7  8  9 10 11 12 13 |  4  5  6  7  8  9 10 |
|  9 10 11 12 13 14 15 | 14 15 16 17 18 19 20 | 11 12 13 14 15 16 17 |
| 16 17 18 19 20 21 22 | 21 22 23 24 25 26 27 | 18 19 20 21 22 23 24 |
| 23 24 25 26 27 28 29 | 28 29 30 31          | 25 26 27 28 29 30    |
| 30                   |                      |                      |
+----------------------+----------------------+----------------------+
|      July 2006       |     August 2006      |    September 2006    |
| su mo tu we th fr sa | su mo tu we th fr sa | su mo tu we th fr sa |
|                    1 |        1  2  3  4  5 |                1  2   |
|  2  3  4  5  6  7  8 |  6  7  8  9 10 11 12 |  3  4  5  6  7  8  9 |
|  9 10 11 12 13 14 15 | 13 14 15 16 17 18 19 | 10 11 12 13 14 15 16 |
| 16 17 18 19 20 21 22 | 20 21 22 23 24 25 26 | 17 18 19 20 21 22 23 |
| 23 24 25 26 27 28 29 | 27 28 29 30 31       | 24 25 26 27 28 29 30 |
| 30 31                |                      |                      |
+----------------------+----------------------+----------------------+
|     October 2006     |                      |                      |
| su mo tu we th fr sa  |                      |                      |
|  1  2  3  4  5  6  7 |                      |                      |
|  8  9 10 11 12 13 14 |                      |                      |
| 15 16 17 18 19 20 21 |                      |                      |
| 22 23 24 25 26 27 28 |                      |                      |
| 29 30 31             |                      |                      |
|                      |                      | 31                   |
+----------------------+----------------------+----------------------+
```

- each month contains 7 lines
- months vertically separated by 22 * -
- 4*3 months
- joins : +

The main program

We start with the main program, which:

The main program

We start with the main program, which:

- prompts for the year of the calendar

$$main = \textbf{do}\ putStr\ \texttt{"See calendar for which year? "}$$

The main program

We start with the main program, which:

- prompts for the year of the calendar
- reads the number representing the year

```
main = do putStr "See calendar for which year? "
          input ← getLine
          let year :: Int
              year = readUnsigned input
```

The main program

We start with the main program, which:

- prompts for the year of the calendar
- reads the number representing the year
- prints the result

```
module Calendar where
main :: IO ()
main = do putStr "See calendar for which year? "
          input ← getLine
          let year :: Int
              year = readUnsigned input
          if year > 1752
            then putStrLn (showCalendarForYear year)
            else putStrLn "year should be >1752"
```

Building the complete calendar

showCalendarForYear :: *Year*→*String*

$ [0 . . 11]

Building the complete calendar

$$showCalendarForYear :: Year \rightarrow String$$

$$\circ\ map\ (calendarMonth\ year)$$
$$\$\ [0\,..\,11]$$

Building the complete calendar

$$showCalendarForYear :: Year \rightarrow String$$

$$\begin{aligned} &\circ\ makeGroupsOf\ 3 \\ &\circ\ map\ (calendarMonth\ year) \\ &\$\ [0\,..\,11] \end{aligned}$$

Universiteit Utrecht

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

Building the complete calendar

$$showCalendarForYear :: Year \rightarrow String$$

$$\begin{aligned}
&\circ\ map\ besides \\
&\circ\ makeGroupsOf\ 3 \\
&\circ\ map\ (calendarMonth\ year) \\
\$\ &[0\,..\,11]
\end{aligned}$$

Universiteit Utrecht

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

Building the complete calendar

showCalendarForYear :: *Year*→*String*

- *separateBy horizontal*
- *map besides*
- *makeGroupsOf* 3
- *map* (*calendarMonth year*)

$ [0 . . 11]

Building the complete calendar

showCalendarForYear :: *Year*→*String*

- *concat*
- *separateBy horizontal*
- *map besides*
- *makeGroupsOf* 3
- *map* (*calendarMonth year*)

$ [0 . . 11]

Building the complete calendar

```
showCalendarForYear :: Year→String
showCalendarForYear year = unlines
                         ∘ concat
                         ∘ separateBy horizontal
                         ∘ map besides
                         ∘ makeGroupsOf 3
                         ∘ map (calendarMonth year)
                         $ [0..11]
```

Building the complete calendar

```
showCalendarForYear :: Year→String
showCalendarForYear year = unlines
                         ∘ concat
                         ∘ separateBy horizontal
                         ∘ map besides
                         ∘ makeGroupsOf 3
                         ∘ map (calendarMonth year)
                         $ [0..11]
unlines ls = unlines' ls ""
           where unlines' []     res = res
                 unlines' (l:ls) res = l ++
                                       ('\n' : unlines' ls res)
f $ x = f x
```

Types, Strings, Numbers

```
type Year      = Int
type Month     = Int
type Day       = Int
monthNames :: [String]
monthNames =
              ["January","February","March"
              ,"April"  ,"May"     ,"June"
              ,"July"   ,"August"  ,"September"
              ,"October","November","December"
              ]
```

When do we have a leap year?

$$
\begin{aligned}
&isLeapYear && :: Year \rightarrow Bool \\
&isLeapYear\ year && = year\ `mod`\ 4 \equiv 0 \\
&&& \quad \wedge \neg\ (year\ `mod`\ 100 \equiv 0 \\
&&& \qquad \wedge \\
&&& \quad year\ `mod`\ 400 \not\equiv 0 \\
&&& \quad)
\end{aligned}
$$

How many days does a specific month have?

```
daysInMonth :: Year→Month→Int
daysInMonth month year = list !! month
  where
    list :: [Int]
    list = [31,february,31,30,31,30,31,31,30,31,30,31]
    february :: Int
    february = if isLeapYear year then 29 else 28
```

Formatting within a field of a given width

Justify a string s to the right in a field of width i:

$$
\begin{aligned}
&rjustify\ ::\ Int \rightarrow String \rightarrow String \\
&rjustify\ i\ s = replicate\ (i - length\ s)\ '\ '\ +\!\!+\ s
\end{aligned}
$$

Formatting within a field of a given width

Justify a string s to the right in a field of width i:

$$
\begin{aligned}
&rjustify :: Int \to String \to String \\
&rjustify\ i\ s = replicate\ (i - length\ s)\ '\ '\ +\!\!+\ s
\end{aligned}
$$

Center a string s in a field of width i:

$$
\begin{aligned}
&cjustify :: Int \to String \to String \\
&cjustify\ i\ s = \textbf{let}\ sp :: String \\
&\qquad\qquad\qquad\qquad sp = replicate\ ((i - length\ s)\ `div`\ 2)\ '\ ' \\
&\qquad\qquad\qquad\ \textbf{in}\ take\ i\ (sp +\!\!+ s +\!\!+ repeat\ '\ ')
\end{aligned}
$$

Separating and grouping

$$
\begin{aligned}
&separateBy :: a \to [a] \to [a] \\
&separateBy\ sep\ xs = foldr\ (\lambda x\ r \to sep : x : r)\ [sep]\ xs
\end{aligned}
$$

$$
\begin{aligned}
&makeGroupsOf\ \ :: Int \to [a] \to [[a]]) \\
&makeGroupsOf\ \_\ [\,] = [\,] \\
&makeGroupsOf\ i\ \ xs = take\ i\ xs : makeGroupsOf\ i\ (drop\ i\ xs)
\end{aligned}
$$

Create the entry for a specific month and year

```
calendarMonth :: Year→Month→[String]
calendarMonth month year = title : body
  where
    title ::  String
    title =  cjustify 22 (monthNames !! month ++ "  "
                          ++ showInt year)

    body :: [String]
    body = ...
           ...
           ...
           $ (["su","mo","tu","we","th","fr","sa"]
           ...
           ...
           ...
           ...
             )
```

Universiteit Utrecht

Day headers

Create the entry for a specific month and year

```
calendarMonth :: Year→Month→[String]
calendarMonth month year = title : body
  where
    title ::  String
    title =  cjustify 22 (monthNames !! month ++ "  "
                          ++ showInt year)

    body :: [String]
    body = ...
           ...
           ...
           $  ( ["su","mo","tu","we","th","fr","sa"]
              ++ replicate firstDayOfMonth "    "
           ...
           ...
           ...
             )
```

Universiteit Utrecht

Leading spaces

Create the entry for a specific month and year

```
calendarMonth :: Year→Month→[String]
calendarMonth month year = title : body
  where
    title ::  String
    title =  cjustify 22 (monthNames !! month ++ "  "
                          ++ showInt year)

    body :: [String]
    body = ...
           ...
           ...
           $  ( ["su","mo","tu","we","th","fr","sa"]
              ++ replicate firstDayOfMonth "    "
              ++ map (rjustify 2 ∘ showInt)
                     [1..daysInMonth year month]
              ...
               )
```

Universiteit Utrecht

The days themselves

Create the entry for a specific month and year

```
calendarMonth :: Year→Month→[String]
calendarMonth month year = title : body
  where
    title ::  String
    title =  cjustify 22 (monthNames !! month ++ "  "
                          ++ showInt year)

    body :: [String]
    body = ...
           ...
           ...
           $  ( ["su","mo","tu","we","th","fr","sa"]
              ++ replicate firstDayOfMonth "   "
              ++ map (rjustify 2 ∘ showInt)
                     [1..daysInMonth year month]
              ++ repeat "   "
               )
```

Universiteit Utrecht

At the end of the month

Create the entry for a specific month and year

```
calendarMonth :: Year→Month→[String]
calendarMonth month year = title : body
  where
    title ::  String
    title =  cjustify 22 (monthNames !! month ++ "  "
                          ++ showInt year)

    body :: [String]
    body = ...
           ...
           ∘ makeGroupsOf 7
           $ (   ["su","mo","tu","we","th","fr","sa"]
             ++ replicate firstDayOfMonth "   "
             ++ map (rjustify 2 ∘ showInt)
                    [1..daysInMonth year month]
             ++ repeat "   "
             )
```

Universiteit Utrecht

Build weeks

Create the entry for a specific month and year

```
calendarMonth :: Year→Month→[String]
calendarMonth month year = title : body
  where
    title ::  String
    title =  cjustify 22 (monthNames !! month ++ "  "
                          ++ showInt year)

    body :: [String]
    body = ...
           ∘ map (concat ∘ separateBy "  ")
           ∘ makeGroupsOf 7
           $ (   ["su","mo","tu","we","th","fr","sa"]
             ++ replicate firstDayOfMonth "   "
             ++ map (rjustify 2 ∘ showInt)
                    [1..daysInMonth year month]
             ++ repeat "   "
             )
```

Universiteit Utrecht

Spaces between days

Create the entry for a specific month and year

```
calendarMonth :: Year→Month→[String]
calendarMonth month year = title : body
  where
    title ::  String
    title =  cjustify 22 (monthNames !! month ++ "  "
                          ++ showInt year)

    body :: [String]
    body =    take 7
            ∘ map (concat ∘ separateBy "  ")
            ∘ makeGroupsOf 7
            $ (   ["su","mo","tu","we","th","fr","sa"]
              ++ replicate firstDayOfMonth "   "
              ++ map (rjustify 2 ∘ showInt)
                     [1..daysInMonth year month]
              ++ repeat "   "
              )
```

Universiteit Utrecht

Each month takes 7 lines

... continuation

```
firstDayOfMonth :: Int
firstDayOfMonth
                = (   year    -- 365 `mod` 7 == 1
                  + nrOfLeapYears
                  + sum [daysInMonth year m | m ← [0..n
                  ) `mod` 7

nrOfLeapYears :: Int
nrOfLeapYears = (year − 1) `div` 4
              − (year − 1) `div` 100
              + (year − 1) `div` 400
```

Positioning of blocks

Positioning of blocks

$$
\begin{aligned}
&\mathit{besides} \quad :: [[\mathit{String}]] \to [\mathit{String}] \\
&\mathit{besides}\ \mathit{xxs} = \mathit{foldr1}\ (\mathit{zipWith}\ (+\!\!+))\ \$\ \mathit{separateBy}\ \mathit{vertical}\ \mathit{xxs}
\end{aligned}
$$

Positioning of blocks

```
vertical     :: [String]
vertical     = repeat "|"
besides      :: [[String]]→[String]
besides xxs = foldr1 (zipWith (++)) $ separateBy vertical xxs
horizontal  :: [String]
horizontal  = [concat (separateBy "+" (replicate 3
                                       (replicate 22 '-')))
              ]
```