The function \textit{map}

\begin{align*}
\text{suclist} &:: \text{[Int]} \rightarrow \text{[Int]} \\
\text{suclist} \ [] &\ = \ [] \\
\text{suclist} \ (x:xs) &\ = \ \text{suc} \ x : \ \text{suclist} \ xs \\
\text{sqrtlist} &:: \text{[Float]} \rightarrow \text{[Float]} \\
\text{sqrtlist} \ [] &\ = \ [] \\
\text{sqrtlist} \ (x:xs) &\ = \ \text{sqrt} \ x : \ \text{sqrtlist} \ xs \\
\text{map} &:: (a \rightarrow b) \rightarrow \text{[a]} \rightarrow \text{[b]} \\
\text{map} \ g \ [] &\ = \ [] \\
\text{map} \ g \ (x:xs) &\ = \ g \ x : \ \text{map} \ g \ xs
\end{align*}

\[ \Downarrow \]

\begin{align*}
\text{suclist} &:: \text{[Int]} \rightarrow \text{[Int]} & \text{sqrtlist} &:: \text{[Float]} \rightarrow \text{[Float]} \\
\text{suclist} &\ = \ \text{map} \ \text{suc} & \text{sqrtlist} &\ = \ \text{map} \ \text{sqrt}
\end{align*}
The function filter

\[
\text{dropEven} :: [\text{Int}] \to [\text{Int}]
\]
\[
\text{dropEven} [] = []
\]
\[
\text{dropEven} (x:xs) | \text{odd } x = x : \text{dropEven} \ xs
\]
\[
| \text{otherwise} = \text{dropEven} \ xs
\]

\[
\text{dropUpper} :: [\text{Char}] \to [\text{Char}]
\]
\[
\text{dropUpper} [] = []
\]
\[
\text{dropUpper} (x:xs) | \text{isLower } x = x : \text{dropUpper} \ xs
\]
\[
| \text{otherwise} = \text{dropUpper} \ xs
\]

\[
\text{filter} :: (a \to \text{Bool}) \to [a] \to [a]
\]
\[
\text{filter} g [] = []
\]
\[
\text{filter} g (x:xs) | g x = x : \text{filter} g \ xs
\]
\[
| \text{otherwise} = \text{filter} g \ xs
\]

\[
\text{dropEven} :: [\text{Int}] \to [\text{Int}]
\]
\[
\text{dropEven} = \text{filter} \ \text{odd}
\]
\[
\text{dropUpper} :: [\text{Char}] \to [\text{Char}]
\]
\[
\text{dropUpper} = \text{filter} \ \text{isLower}
\]
The function fold

add :: (List Int) -> Int
add Nil = 0
add (Cons x xs) = plus x (add xs)

prod :: (List Int) -> Int
prod Nil = 1
prod (Cons x xs) = times x (prod xs)

concat :: List (List a) -> List a
concat Nil = Nil
concat (Cons x xs) = append x (concat xs)

fold :: (a -> b -> b) -> b -> (List a) -> b
fold g e Nil = e
fold g e (Cons x xs) = g x (fold g e xs)

add = fold plus 0
prod = fold times 1

concat = fold append Nil
The function foldr

sum :: [Int] -> Int
sum [] = 0
sum (x:xs) = x + sum xs

prod :: [Int] -> Int
prod [] = 1
prod (x:xs) = x * prod xs

concat :: [[a]] -> [a]
concat [] = []
concat (x:xs) = x ++ concat xs

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr g e [] = e
foldr g e (x:xs) = g x (foldr g e xs)

→

sum :: [Int] -> Int
sum = foldr (+) 0
prod :: [Int] -> Int
prod = foldr (*) 1

concat :: [[a]] -> [a]
concat = foldr (++) []