The function \textit{map}

\begin{align*}
suclist &:: [\text{Int}] \rightarrow [\text{Int}] \\
suclist \; [] &\; = \; [] \\
suclist \; (x:xs) &\; = \; \text{suc} \; x \; : \; suclist \; xs \\
\end{align*}

\begin{align*}
\text{sqrtlist} &:: [\text{Float}] \rightarrow [\text{Float}] \\
\text{sqrtlist} \; [] &\; = \; [] \\
\text{sqrtlist} \; (x:xs) &\; = \; \text{sqrt} \; x \; : \; \text{sqrtlist} \; xs \\
\end{align*}

\begin{align*}
\text{map} &:: (a \rightarrow b) \rightarrow [a] \rightarrow [b] \\
\text{map} \; g \; [] &\; = \; [] \\
\text{map} \; g \; (x:xs) &\; = \; g \; x \; : \; \text{map} \; g \; xs \\
\end{align*}

\begin{align*}
suclist \; = \; \text{map} \; \text{suc} \\
\text{sqrtlist} \; = \; \text{map} \; \text{sqrt} \\
\end{align*}
The function \textit{filter}

\begin{verbatim}
dropEven :: [Int] -> [Int]
dropEven [] = []
dropEven (x:xs) | odd x = x : dropEven xs
               | otherwise = dropEven xs

dropUpper :: [Char] -> [Char]
dropUpper [] = []
dropUpper (x:xs) | isLower x = x : dropUpper xs
                 | otherwise = dropUpper xs
\end{verbatim}

\begin{verbatim}
filter :: (a -> Bool) -> [a] -> [a]
filter g [] = []
filter g (x:xs) | g x = x : filter g xs
                | otherwise = filter g xs
\end{verbatim}

\[\downarrow\]

\begin{verbatim}
dropEven :: [Int] -> [Int]
dropEven = filter odd

dropUpper :: [Char] -> [Char]
dropUpper = filter isLower
\end{verbatim}
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 :: (List Int) -> Int
add = fold plus 0

prod :: (List Int) -> Int
prod = fold times 1

concat :: List (List a) -> List a
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 (++) []
```