Higher-order Functions

A function is called **higher-order** if it takes a function as an argument or returns a function as a result.

```
twice :: (a → a) → a → a
twice f x = f (f x)
```

twice is higher-order because it takes a function as its first argument.

Note:

- Higher-order functions are very common in Haskell (and in functional programming).
- Writing higher-order functions is crucial practice for effective programming in Haskell, and for understanding others’ code.
Why Are They Useful?

- **Common programming idioms** can be encoded as functions within the language itself.

- **Domain specific languages** can be defined as collections of higher-order functions. For example, higher-order functions for processing lists.

- **Algebraic properties** of higher-order functions can be used to reason about programs.
The Map Function

The higher-order library function called `map` applies a function to every element of a list.

\[
\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]
\]

For example:

\[
\text{map } (+1) [1,3,5,7] = [2,4,6,8]
\]

The map function can be defined in a particularly simple manner using a list comprehension:

\[
\text{map } f \; \text{xs} = [f \; x \mid x \leftarrow \text{xs}]
\]

Alternatively, it can also be defined using recursion:

\[
\text{map } f \; [] = [] \\
\text{map } f \; (x:xs) = f \; x : \text{map } f \; xs
\]
The Filter Function
The higher-order library function \texttt{filter} selects every element from a list that satisfies a predicate.

For example:

\[
\text{filter} :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]
\]

\[
> \text{filter \ even \ [1..10]}
\]

\[
[2,4,6,8,10]
\]

Filter can be defined using a list comprehension:

\[
\text{filter \ p \ xs} = [x \mid x \leftarrow xs, \ p \ x]
\]

Alternatively, it can also be defined using recursion:

\[
\begin{align*}
\text{filter \ p \ []} &= [] \\
\text{filter \ p \ (x:xs)} &= \begin{cases} \\
\ p \ x &= x : \text{filter \ p \ xs} \\
\text{otherwise} &= \text{filter \ p \ xs}
\end{cases}
\end{align*}
\]
The foldr Function

A number of functions on lists can be defined using the following simple pattern of recursion:

\[
\begin{align*}
  f \ [ ] &= v_0 \\
  f \ (x:xs) &= x \oplus f \ xs
\end{align*}
\]

f maps the empty list to some value v, and any non-empty list to some function \( \oplus \) applied to its head and f of its tail.
For example:

\[ \text{sum } [ ] = 0 \]
\[ \text{sum } (x:xs) = x + \text{sum } xs \]

\[ \text{product } [ ] = 1 \]
\[ \text{product } (x:xs) = x \times \text{product } xs \]

\[ \text{and } [ ] = \text{True} \]
\[ \text{and } (x:xs) = x \text{ \&\& and } xs \]
The higher-order library function \texttt{foldr} (fold right) encapsulates this simple pattern of recursion, with the function $\oplus$ and the value $v$ as arguments.

For example:

\begin{align*}
\text{sum} & = \text{foldr} \ (+) \ 0 \\
\text{product} & = \text{foldr} \ (*) \ 1 \\
\text{or} & = \text{foldr} \ (||) \ \text{False} \\
\text{and} & = \text{foldr} \ (&&) \ \text{True}
\end{align*}
foldr itself can be defined using recursion:

\[
\text{foldr} :: (a \to b \to b) \to b \to [a] \to b
\]

\[
\text{foldr } f \ v \ [] \quad = \quad v
\]

\[
\text{foldr } f \ v \ (x:xs) \quad = \quad f \ x \ (\text{foldr } f \ v \ xs)
\]

However, it is best to think of foldr non-recursively, as simultaneously replacing each (:) in a list by a given function, and [] by a given value.
For example:

\[
\text{sum } [1,2,3] \\
= \quad \text{foldr } (+) 0 [1,2,3] \\
= \quad \text{foldr } (+) 0 (1:(2:(3:[]))) \\
= \quad 1+(2+(3+0)) \\
= \quad 6
\]

Replace each (:) by (+) and [] by 0.
For example:

\[
\text{product } [1,2,3] \\
= \text{foldr } (*) \ 1 \ [1,2,3] \\
= \text{foldr } (*) \ 1 \ (1:(2:(3:[])))) \\
= 1*(2*(3*1)) \\
= 6
\]

Replace each (:) by (*) and [] by 1.
Other foldr Examples

Even though foldr encapsulates a simple pattern of recursion, it can be used to define many more functions than might first be expected.

Recall the length function:

\[
\text{length :: [a] → Int}
\]

\[
\text{length } [] = 0
\]

\[
\text{length } (_:xs) = 1 + \text{length } xs
\]
For example:

\[
\text{length } [1,2,3] \\
= \text{length } (1:(2:(3:[]))) \\
= 1+(1+(1+0)) \\
= 3
\]

Hence, we have:

\[
\text{length } = \text{foldr } (\lambda\_\_ n \rightarrow 1+n) 0
\]
Now the reverse function:

\[
\begin{align*}
\text{reverse } [] & = [] \\
\text{reverse } (x:xs) & = \text{reverse } xs ++ [x]
\end{align*}
\]

For example: \(\text{reverse } [1,2,3]\)

\[
\begin{align*}
= & \text{reverse } (1:(2:(3:[]))) \\
= & (([] ++ [3]) ++ [2]) ++ [1] \\
= & [3,2,1]
\end{align*}
\]

Hence, we have:

\[
\text{reverse } = \text{foldr } (\lambda x xs \rightarrow xs ++ [x]) []
\]

Here, the append function (++) has a particularly compact definition using foldr:

\[
(\text{++) ys) } = \text{foldr } (:) ys
\]
Why Is foldr Useful?

- Some recursive functions on lists, such as sum, are **simpler** to define using foldr.

- Properties of functions defined using foldr can be proved using algebraic properties of foldr.

- Advanced program **optimizations** can be simpler if foldr is used in place of explicit recursion.
foldr and foldl

foldr :: \( (a \to b \to b) \to b \to [a] \to b \)
foldr \( f \) \( v \) \([]\) = \( v \)
foldr \( f \) \( v \) \((x:xs)\) = \( f \) \( x \) (foldr \( f \) \( v \) \( xs \))

foldl :: \( (a \to b \to a) \to a \to [b] \to a \)
foldl \( f \) \( v \) \([]\) = \( v \)
foldl \( f \) \( v \) \((x:xs)\) = foldl \( f \) (f \( v \) \( x \)) \( xs \)

- \( \text{foldr} \quad 1 : 2 : 3 : [] \Rightarrow (1 + (2 + (3 + 0))) \)
- \( \text{foldl} \quad 1 : 2 : 3 : [] \Rightarrow (((0 + 1) + 2) + 3) \)
Other Library Functions

The library function \( . \) returns the \textbf{composition} of two functions as a single function.

\[
( . ) \quad :: \quad (b \to c) \to (a \to b) \to (a \to c)
\]

\[
f \cdot g = \lambda x \to f (g \ x)
\]

For example:

\[
\text{odd} :: \text{Int} \to \text{Bool}
\]

\[
\text{odd} = \text{not} \cdot \text{even}
\]

Exercise: Define \texttt{filterOut} \( p \) \( x s \) that retains elements that do not satisfy \( p \).

\[
\text{filterOut} \ p \ x s = \text{filter} \ (\text{not} \cdot p) \ x s
\]

\[
> \text{filterOut \ odd \ [1..10]}\\
[2,4,6,8,10]
\]
The library function \texttt{all} decides if every element of a list satisfies a given predicate.

\texttt{all} :: (a → \text{Bool}) → [a] → \text{Bool}
\texttt{all \ p \ xs} = \text{and \ [p \ x \mid x \leftarrow xs]}

For example:

\texttt{> all even [2,4,6,8,10]}

\texttt{True}
Conversely the library function \texttt{any} decides if at least one element of a list satisfies a predicate.

\begin{verbatim}
any   :: (a → Bool) → [a] → Bool
any p xs = or [p x | x ← xs]
\end{verbatim}

For example:

\begin{verbatim}
> any isSpace "abc def"
True
\end{verbatim}
The library function `takeWhile` selects elements from a list while a predicate holds of all the elements.

```haskell
takeWhile :: (a → Bool) → [a] → [a]
takeWhile p []     = []
takeWhile p (x:xs)
    | p x           = x : takeWhile p xs
    | otherwise     = []
```

For example:

```haskell
> takeWhile isAlpha "abc def"
"abc"
```
Dually, the function \texttt{dropWhile} removes elements while a predicate holds of all the elements.

\begin{verbatim}
dropWhile :: (a \to \text{Bool}) \to [a] \to [a]
dropWhile p []    = []
dropWhile p (x:xs) |
    | p x           = dropWhile p xs
    | otherwise     = x:xs
\end{verbatim}

For example:

\begin{verbatim}
> dropWhile isSpace "   abc"
"abc"
\end{verbatim}
filter, map and foldr

Typical use is to select certain elements, and then perform a mapping, for example,

```
sumSquaresOfPos ls
    = foldr (+) 0 (map (^2) (filter (>= 0) ls))

> sumSquaresOfPos [-4,1,3,-8,10]
110
```

In pieces:

```
keepPos = filter (>= 0)
mapSquare = map (^2)
sum = foldr (+) 0
sumSquaresOfPos ls = sum (mapSquare (keepPos ls))
```

Alternative definition:

```
sumSquaresOfPos = sum . mapSquare . keepPos
```
Exercises

(1) What are higher-order functions that return functions as results better known as?

(2) Express the comprehension $[f \ x \mid x \leftarrow xs, \ p \ x]$ using the functions map and filter.

(3) Redefine map f and filter p using foldr.