Chapter 7 - Higher-Order Functions
Introduction

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.
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.

- **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:

> 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:

\[
\text{map} \ f \ \mathbf{x}s = [f \ x \mid x \leftarrow \mathbf{x}s]
\]

Alternatively, for the purposes of proofs, the map function 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 `filter` selects every element from a list that satisfies a predicate.

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

For example:

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

\[
[2,4,6,8,10]
\]
Filter can be defined using a list comprehension:

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

Alternatively, it can be defined using recursion:

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

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

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

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 `foldr` (fold right) encapsulates this simple pattern of recursion, with the function ⊕ and the value v as arguments.

For example:

```
sum = foldr (+) 0
product = foldr (*) 1
or = foldr (||) False
and = foldr (&&) True
```
Foldr itself can be defined using recursion:

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

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

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

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

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

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.
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:

\[
\begin{align*}
\text{length} &: [a] \rightarrow \text{Int} \\
\text{length} [ ] &= 0 \\
\text{length} (_:\text{xs}) &= 1 + \text{length} \text{ xs}
\end{align*}
\]
For example:

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

Hence, we have:

\[\text{length } = \text{foldr } (\lambda_\_ \text{ n } \rightarrow 1+n) 0\]

Replace each (:) by \(\lambda_\_ \text{ n } \rightarrow 1+n\) and [] by 0.
Now recall the reverse function:

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

For example:

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

Replace each (:) by \( \lambda x \hspace{1em} xs \rightarrow xs +\ + \ [x] \) and [] by [].
Hence, we have:

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

Finally, we note that the append function (\(++\)) has a particularly compact definition using foldr:

\[
(\++ \ ys) = \text{foldr} \ (:) \ ys
\]

Replace each (\(:)\) by (\(:)\) and [] by 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, such as fusion and the banana split rule.

- Advanced program optimisations can be simpler if foldr is used in place of explicit recursion.
Other Library Functions

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

\[
(\cdot) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)
\]

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

For example:

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

\[
\text{odd} = \text{not} \ . \ \text{even}
\]
The library function `all` decides if every element of a list satisfies a given predicate.

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

\[
all\ p\ xs = \text{and}\ [p\ x \mid x \leftarrow xs]
\]

For example:

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

True
Dually, the library function `any` decides if at least one element of a list satisfies a predicate.

\[
\text{any} :: (a \to \text{Bool}) \to [a] \to \text{Bool}
\]

\[
\text{any} \ p \ xs = \text{or} \ [p \ x \mid x \leftarrow xs]
\]

For example:

\[
> \text{any} \ (== \ ' ') \ "abc def"
\]

True
The library function `takeWhile` selects elements from a list while a predicate holds of all the elements.

```
takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile p [] = []
takeWhile p (x:xs)
    | p x       = x : takeWhile p xs
    | otherwise = []
```

For example:

```
> takeWhile (/= '') "abc def"
"abc"
```
Dually, the function `dropWhile` removes elements while a predicate holds of all the elements.

\[
\text{dropWhile} :: (a \to \text{Bool}) \to [a] \to [a]
\]

\[
\text{dropWhile} \ p \ [\] = [\]
\]

\[
\text{dropWhile} \ p \ (x:xs)
\]

\[
| \ p \ x \quad = \text{dropWhile} \ p \ xs
\]

\[
| \ \text{otherwise} = x:xs
\]

For example:

\[
> \ \text{dropWhile} \ (== \ ' ' \ ) \ " \ abc"
\]

"abc"
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.