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.

\[
twice :: (a \rightarrow a) \rightarrow a \rightarrow a
\]
\[
twice \; f \; x = f \; (f \; x)
\]

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 \texttt{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, for the purposes of proofs, the map function can also be defined using recursion:

$$\text{map } f \, [] = []$$

$$\text{map } f \, (x:\text{xs}) = f \, x : \text{map } f \, \text{xs}$$
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:

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

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

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

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:

\[
\begin{align*}
\text{sum }[] &= 0 \\
\text{sum } (x:xs) &= x + \text{sum } xs
\end{align*}
\]

\[
\begin{align*}
\text{product }[] &= 1 \\
\text{product } (x:xs) &= x * \text{product } xs
\end{align*}
\]

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

For example:

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

```
fldr :: (a → b → b) → b → [a] → b
foldr f v []     = v
foldr f v (x:xs) = f x (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:

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

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

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

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:

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

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

Hence, we have:

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

Replace each \((:)\) by \(\lambda \_ n \to 1+n\) and \([]\) by 0.
Now recall the reverse function:

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

For example:

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

Replace each (:) by \(\lambda x \ xs \rightarrow xs ++ [x]\) 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:

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

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.

\[
(\cdot) :: (b \to c) \to (a \to b) \to (a \to c)
\]
\[
f \cdot g = \lambda x \to f (g x)
\]

For example:

odd :: Int \to Bool
odd = not \cdot even
The library function `all` decides if every element of a list satisfies a given predicate.

```hs
all :: (a → Bool) → [a] → Bool
all p xs = and [p x | x ← xs]
```

For example:

```hs
> 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.

\[
\text{any} :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow \text{Bool}
\]
\[
\text{any } p \text{ } xs = \text{or} \ [p \ x \mid x \leftarrow xs]
\]

For example:

```
> any isSpace "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 isAlpha "abc def"
"abc"
```
Dually, the function `dropWhile` removes elements while a predicate holds of all the elements.

\[
dropWhile :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]
dropWhile p [] = []
dropWhile p (x:xs)
  | p x = dropWhile p xs
  | otherwise = x:xs
\]

For example:

```
> dropWhile isSpace " 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.