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.

map :: (a -> b) -> [a] -> [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:

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

Alternatively, for the purposes of proofs, the map function can also be defined using recursion:

\[
\begin{align*}
\text{map } f \ [] & = [] \\
\text{map } f \ (x:xs) & = f \ x : \text{map } f \ xs
\end{align*}
\]
The Filter Function

The higher-order library function `filter` selects every element from a list that satisfies a predicate.

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

For example:

\[
> \text{filter even} \ [1..10] \\
[2,4,6,8,10]
\]
Filter can be defined using a list comprehension:

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

Alternatively, it can be defined using recursion:

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

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

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

\[
\begin{align*}
\text{and } [] & = \text{True} \\
\text{and } (x:xs) & = x \&\& \text{and } xs
\end{align*}
\]
The higher-order library function \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} \ (\ast \ ) \ 1 \\
\text{or} &= \text{foldr} \ (\|\|) \ \text{False} \\
\text{and} &= \text{foldr} \ (\&\&) \ \text{True}
\end{align*}
Foldr itself can be defined using recursion:

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

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

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

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:

\[
\begin{align*}
\text{sum } [1,2,3] &= \\
&= \text{foldr (+) 0 [1,2,3]} \\
&= \text{foldr (+) 0 (1:(2:(3:[])))} \\
&= 1+(2+(3+0)) \\
&= 6
\end{align*}
\]

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:

\[
\text{length} :: [a] \rightarrow \text{Int} \\
\text{length} [] = 0 \\
\text{length} (_:xs) = 1 + \text{length} \; xs
\]
For example:

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

Hence, we have:

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

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

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

For example:

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

Replace each (:) by \( \lambda x \, x\text{s} \rightarrow x\text{s} \, ++ \, [x] \) and [] by [].

Hence, we have:

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

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

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

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:

\[
\text{odd} :: \text{Int} \to \text{Bool}
\]
\[
\text{odd} = \text{not} \cdot \text{even}
\]
The library function \texttt{all} decides if every element of a list satisfies a given predicate.

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

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

For example:

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

\[
\text{True}
\]
Dually, the library function `any` decides if at least one element of a list satisfies a predicate.

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

\[
\text{any \hspace{1em} p \hspace{1em} xs} = \text{or \hspace{1em} [p \hspace{1em} x \mid x \leftarrow xs]}
\]

For example:

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

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

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

For example:

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

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

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 \(foldr\).