Chapter 9 - Higher-Order Functions, Functional Parsers
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 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:

\[
\text{map } f \hspace{1ex} \text{xs} = [f \hspace{1ex} x \mid x \leftarrow \text{xs}]
\]

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

\[
\text{map } f \hspace{1ex} [] = []
\]

\[
\text{map } f \hspace{1ex} (x:xs) = f \hspace{1ex} x : \text{map } f \hspace{1ex} xs
\]
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:

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

Alternatively, it can be defined using recursion:

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

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

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

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:

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

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

\[
\begin{align*}
\text{and} \ [ ] &= \text{True} \\
\text{and} \ (x:xs) &= x \land \text{and} \ xs
\end{align*}
\]
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:

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

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

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

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

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:

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

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.
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:

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

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

For example:

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

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

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

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.

\begin{verbatim}
all :: (a -> Bool) -> [a] -> Bool
all p xs = and [p x | x <- xs]
\end{verbatim}

For example:

\begin{verbatim}
> all even [2,4,6,8,10]
True
\end{verbatim}
Dually, the library function `any` decides if at least one element of a list satisfies a predicate.

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

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

For example:

\[
> \text{any isSpace "abc def"}
\]

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

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

For example:

\[
> \text{takeWhile isAlpha "abc def"}
\]

"abc"
Dually, the function `dropWhile` removes elements while a predicate holds of all the elements.

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

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

For example:

\[
> \text{dropWhile isSpace " abc" } \\
\text{"abc"}
\]
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.
What is a Parser?

A parser is a program that analyses a piece of text to determine its syntactic structure.
Where Are They Used?

Almost every real life program uses some form of parser to **pre-process** its input.

- Hugs
- Unix
- Explorer

parses

- Haskell programs
- Shell scripts
- HTML documents
The Parser Type

In a functional language such as Haskell, parsers can naturally be viewed as functions.

```
type Parser = String -> Tree
```

A parser is a function that takes a string and returns some form of tree.
However, a parser might not require all of its input string, so we also return any unused input:

```
  type Parser = String → (Tree, String)
```

A string might be parsable in many ways, including none, so we generalize to a list of results:

```
  type Parser = String → [(Tree, String)]
```
Finally, a parser might not always produce a tree, so we generalize to a value of any type:

```
type Parser a = String -> [(a, String)]
```

Note:

- For simplicity, we will only consider parsers that either fail and return the empty list of results, or succeed and return a singleton list.
Basic Parsers

The parser `item` fails if the input is empty, and consumes the first character otherwise:

\[
\text{item :: Parser Char}
\]

\[
\text{item} = \lambda \text{inp} \to \text{case inp of}
\]

\[
[\ ] \to [\ ]
\]

\[
(x:xs) \to [(x, xs)]
\]
The parser **failure** always fails:

\[
\text{failure} :: \text{Parser} \ a \\
\text{failure} = \lambda \text{inp} \rightarrow []
\]

The parser **return v** always succeeds, returning the value v without consuming any input:

\[
\text{return} :: a \rightarrow \text{Parser} \ a \\
\text{return} \ v = \lambda \text{inp} \rightarrow [(v, \text{inp})]
\]
The parser $p +++ q$ behaves as the parser $p$ if it succeeds, and as the parser $q$ otherwise:

$$(+++) :: 	ext{Parser } a \rightarrow 	ext{Parser } a \rightarrow 	ext{Parser } a$$

$p +++ q = \lambda \text{inp} \rightarrow \text{case } p \text{ inp of}$

$$[] \rightarrow \text{parse } q \text{ inp}$$

$$[(v,\text{out})] \rightarrow [(v,\text{out})]$$

The function $\text{parse}$ applies a parser to a string:

$\text{parse} :: \text{Parser } a \rightarrow \text{String} \rightarrow [(a,\text{String})]$  
$\text{parse } p \text{ inp } = p \text{ inp}$
Examples

The behavior of the five parsing primitives can be illustrated with some simple examples:

```hugs
% hugs Parsing

> parse item ""
[]

> parse item "abc"
[(['a','bc')]]
```
> parse failure "abc"
[]

> parse (return 1) "abc"
[(1,"abc")]

> parse (item +++ return 'd') "abc"
[('a','bc')]

> parse (failure +++ return 'd') "abc"
[('d','abc')]
Note:

- The library file Parsing is available on the web from the Programming in Haskell home page.

- For technical reasons, the first failure example actually gives an error concerning types, but this does not occur in non-trivial examples.

- The Parser type is a monad, a mathematical structure that has proved useful for modeling many different kinds of computations.
Sequencing

A sequence of parsers can be combined as a single composite parser using the keyword `do`.

For example:

```haskell
p :: Parser (Char,Char)
p  = do x <- item
       item
       y <- item
       return (x,y)
```
Note:

- Each parser must begin in precisely the same column. That is, the layout rule applies.

- The values returned by intermediate parsers are discarded by default, but if required can be named using the ← operator.

- The value returned by the last parser is the value returned by the sequence as a whole.
If any parser in a sequence of parsers fails, then the sequence as a whole fails. For example:

```haskell
> parse p "abcdef"
[[(('a','c'),"def")]]

> parse p "ab"
[]
```

The do notation is not specific to the Parser type, but can be used with any monadic type.
Derived Primitives

Parsing a character that satisfies a predicate:

```haskell
sat :: (Char → Bool) → Parser Char
sat p = do x ← item
  if p x then return x
  else failure
```
Parsing a digit and specific characters:

```haskell
digit :: Parser Char
digit = sat isDigit

char :: Char -> Parser Char
char x = sat (x ==)
```

Applying a parser zero or more times:

```haskell
many :: Parser a -> Parser [a]
many p = many1 p +++ return []
```
Applying a parser **one or more** times:

```
many1 :: Parser a -> Parser [a]
many1 p = do v ← p
            vs ← many p
            return (v:vs)
```

Parsing a specific **string** of characters:

```
string :: String → Parser String
string [] = return []
string (x:xs) = do char x
               string xs
               return (x:xs)
```
We can now define a parser that consumes a list of one or more digits from a string:

```haskell
p :: Parser String
p = do char '['
    d <- digit
    ds <- many (do char ','
                   digit)
    char ']'
    return (d:ds)
```
For example:

```haskell
> parse p "[1,2,3,4]"
[("1234","")]
```

```haskell
> parse p "[1,2,3,4"
[]
```

Note:

- More sophisticated parsing libraries can indicate and/or recover from errors in the input string.
Arithmetic Expressions

Consider a simple form of expressions built up from single digits using the operations of addition + and multiplication *, together with parentheses.

We also assume that:

- * and + associate to the right;
- * has higher priority than +.
Formally, the syntax of such expressions is defined by the following context free grammar:

\[
\begin{align*}
expr & \rightarrow term \ ' + ' \ expr \mid term \\
term & \rightarrow factor \ ' * ' \ term \mid factor \\
factor & \rightarrow digit \mid (expr) \\
digit & \rightarrow '0' \mid '1' \mid \ldots \mid '9'
\end{align*}
\]
However, for reasons of efficiency, it is important to factorise the rules for *expr* and *term*:

\[
expr \rightarrow \text{term} (\text{'}+\text{' expr} \mid \varepsilon) \\
\text{term} \rightarrow \text{factor} (\text{'}\ast\text{' term} \mid \varepsilon)
\]

**Note:**

- The symbol $\varepsilon$ denotes the empty string.
It is now easy to translate the grammar into a parser that evaluates expressions, by simply rewriting the grammar rules using the parsing primitives.

That is, we have:

```plaintext
expr :: Parser Int
expr = do t ← term
    do char '+'
        e ← expr
        return (t + e)
    +++ return t
```
factor :: Parser Int
factor  = do d  digit
  return (digitToInt d)
+++ do char '('
  e  expr
  return e

term :: Parser Int
term  = do f  factor
  do char '*'
    t  term
  return (f * t)
+++ return f
Finally, if we define

\[
\text{eval} :: \text{String} \rightarrow \text{Int} \\
\text{eval } xs = \text{fst} (\text{head} (\text{parse expr} \ xs))
\]

then we try out some examples:

\[
> \text{eval } "2*3+4"
10
\]

\[
> \text{eval } "2*(3+4)"
14
\]
Exercises

(1) Why does factorising the expression grammar make the resulting parser more efficient?

(2) Extend the expression parser to allow the use of subtraction and division, based upon the following extensions to the grammar:

\[
expr \rightarrow term \ (\ '+' \ expr \mid '\-\' \ expr \mid \varepsilon) \\
\]

\[
term \rightarrow factor \ (\ '*' \ term \mid '/' \ term \mid \varepsilon) \\
\]