5 Higher-order functions

Exercise 5.1 (Warm-up: folding). Redo Exercise 3.2: use the higher-order functions `foldl` and `foldr` to define

1. a function `allTrue :: [Bool] → Bool` that determines whether every element of a list of Booleans is true;
2. a function `allFalse` that similarly determines whether every element of a list of Booleans is false;
3. a function `member :: (Eq a) ⇒ a → [a] → Bool` that determines whether a specified element is contained in a given list;
4. a function `smallest :: [Int] → Int` that calculates the smallest value in a list of integers;
5. a function `largest` that similarly calculates the largest value in a list of integers.

If both recursion schemes are applicable, which one is preferable in terms of running time?

Exercise 5.2 (Programming, `Numeral.lhs`). The decimal and the binary number system are both positional number systems. The meaning of the decimal numeral 4711 is $4 * 10^3 + 7 * 10^2 + 1 * 10^1 + 1 * 10^0$ i.e. the weight of a digit depends on its position. For decimal numerals the most significant digit usually comes first. For binary numerals both conventions, MSD and LSD first, are in use: 1011 denotes either $1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0 = 11$ or $1 * 2^0 + 0 * 2^1 + 1 * 2^2 + 1 * 2^3 = 13$.

1. Use `foldl` and `foldr` to define functions

   ```
   type Base  = Integer
   type Digit = Integer

   msdf, lsdf :: Base → [Digit] → Integer
   ```

   which given a base convert a list of digits into a number (MSD and LSD first). *Hint:* recall Exercise 1.6.3.

2. Try to relate `msdf base` and `lsdf base`. (Can you define one in terms of the other?) Are your findings specific to the application at hand? Try to abstract away from the specifics and derive a general law about `foldl` and `foldr`.

Exercise 5.3 (Programming: parser combinators, `Expression.lhs`). The purpose of this exercise is to explore context-free grammars and parser combinators.

1. The grammar below defines the language of (simple) arithmetic expressions.

```
expr ::= digit { digit }
       | expr '+' expr
       | expr '*' expr
       | '(' expr ')'
```

What's the difference to the grammar shown in the lectures (§5.5, see also below)? Implement the grammar above using parser combinators and test the resulting parser on a few simple examples e.g. `4 * 71 + 1` etc. What do you observe? (You also may want to revisit Exercise 1.6.)

2. Reconsider the expression grammar given in the lectures:

```
expr    ::=  term
          |  term '+' expr
term    ::=  factor
          |  factor '*' term
factor  ::=  digit { digit }
          |  '(' expr ')'
```

   Observe that the alternatives for `expr` and `term` share a common prefix. An important optimization is to *left factor* a grammar to avoid repetitive parses e.g.

```
expr    ::=  term (empty | '+' expr)
term    ::=  factor (empty | '*' term)
```

   Apply the optimization to the expression parser given in the lectures. (The hard part is to adapt the semantic actions.) Does the running time improve? (Within GHCi type `:set +s` to ask GHCi to print timing and memory statistics after each evaluation.)

Exercise 5.4 (Programming: parser combinators, `Lambda.lhs`). Write a parser for Haskell expressions built from variables (`x`, `y` etc), using λ-abstraction (`\ x → e`) and application (`e1 e2`). This tiny language can be seen as the core of Haskell—it is known as the λ-calculus. The *abstract* syntax for λ-expressions is given by the datatype

```haskell
infixl 9 :@

data Lambda var
  =  Var var                  -- variable
  |  Fun var (Lambda var)     -- abstraction/lambda expression
  |  Lambda var :@ Lambda var -- application
  deriving (Show)
```

which abstracts away from the representation of variables. For example,

`Fun 0 (Var 0) :: Lambda Integer` and `Fun "x" (Var "x") :: Lambda String`

represent both the identity but use different types for variables.

1. Define the *concrete* syntax of λ-expressions using a context-free grammar. Haskell's `\ x → x` is traditionally written $\lambda x.x$. Which syntax do you prefer or, perhaps, you want to support both? How do you want to represent variables: by a single letter or using a full-blown identifier? Take you pick. Capture the syntactic conventions that application associates to the left, i.e. $e_1 e_2 e_3$ is shorthand for $(e_1 e_2) e_3$, and that abstraction extends as far as possible to the right, e.g. $\lambda x.xy$ means $\lambda x.(xy)$ rather than $(\lambda x.x)y$.

2. Implement the grammar for λ-expressions using parser combinators and test the resulting parser on a few examples e.g. is $a\lambda x.x$ legal syntax for `Var 'a' :@ Fun 'x' (Var 'x')`?

Exercise 5.5 (Worked example: a tribute a Haskell B. Curry and David Turner, `SKI.lhs`). The purpose of this exercise is to give you an idea how Haskell i.e. the λ-calculus might be implemented. (The technique developed is somewhat outdated, but it was actually used for Haskell's predecessor, David Turner's Miranda.)

The first thing to observe is that a standard stack-based architecture (see §4.5) is not fit for the job. Consider the expression `twice succ`. In a call-by-value regime (eager evaluation) we first push `succ` onto the stack, and then call `twice = \ f → \ x → f (f x)`. But the function `twice` returns immediately, yielding the λ-expression `\ x → f (f x)`, which contains the free variable `f`. Upon return the entry `succ` is removed. But `succ` is required, if the λ-expression is later applied to an argument. (As an aside, this is why the language C features only *top-level* functions.)

So λ-expressions are difficult to implement. Perhaps, we can get rid of them? Surprisingly, this is indeed possible. (The approach is based on results from the mathematical field of *combinatory logic*, which Haskell B. Curry founded.) Let us work through two examples: `twice` and `twice twice`. We first compile the λ-expression into "machine code".

```
⋙ twice = Fun 'f' (Fun 'x' (Var 'f' :@ (Var 'f' :@ Var 'x')))
⋙ compile twice
S (S (K S) (S (K K) I)) (S (S (K S) (S (K K) I)) (K I))
```

Traditional machine code consists of a *sequence* of instructions. Here we have a *binary tree* of instructions instead. To reduce the tree to a normalform we apply it to two "primitives".

```
⋙ reduce it [Free 's', Free 'z']
's' ('s' 'z')
```

(You may want to read `s` as *s*uccessor and `z` as *z*ero.) Thus, `twice s z` is `s (s z)`. The reduction machine has, actually, no notion of primitives: `Free 's'` and `Free 'z'` are really free variables, which are treated purely symbolically. We can also apply `twice` to itself.

```
⋙ compile (twice :@ twice)
S (S (K S) (S (K K) I)) (S (S (K S) (S (K K) I)) (K I)) (S (S (K S)
 (S (K K) I)) (S (S (K S) (S (K K) I)) (K I)))
⋙ reduce it [Free 's', Free 'z']
's' ('s' ('s' ('s' 'z')))
```

Thus, `twice twice s z` is `s (s (s (s z)))`.

The type of machine instructions `SKI var`, defined below, is a stripped-down version of the type of λ-expressions `Lambda var`, defined in Exercise 5.4. The constructor `Free` is the counterpart of `Var`, and `App` (printed as a space in the examples above) is the counterpart of `:@`

```
data SKI var
  = Free var              -- free/unbound variable
  | S
  | K
  | I'
  | App (SKI var) (SKI var)  -- application
```

Compared to `Lambda var`, the type misses a case for λ-expressions—of course, we wanted to get rid of those—and features three additional constants instead: `S`, `K`, and `I`. On the face of it, SKI terms can be seen as binary leaf trees with four kinds of leaves. Now, why these constants? They allow us to *simulate* λ-abstractions. To get an idea how this works consider the λ-expression $\lambda x.f(fx)$:

```
⋙ compile (Fun 'x' (Var 'f' :@ (Var 'f' :@ Var 'x')))
S (K 'f') (S (K 'f') I)
```

The body of the λ-expression consists only of applications, which can be seen as a binary tree. The translation is a binary tree of the same shape: The combinators `S`, `K`, and `I` are machine instructions for performing a substitution: `S` distributes an incoming argument down the two sub-trees, `K` discards an incoming argument, and `I` accepts it. Given

```
i :: env → env
i arg = arg
k :: a → (env → a)
k x _arg = x
s :: (env → a → b) → (env → a) → (env → b)
s x y arg = (x arg) (y arg)
```

we can show that `s (k f) (s (k f) i) =  x → f (f x)`:

```
s (k f) (s (k f) i) ar
   ⇒>   --definition of s}
(k f arg) (s (k f) i arg)
   ⇒>   --definition of k and definition of s
f ((k f arg) (i arg))
   ⇒>   --definition of k and definition of i
f (f arg)
```

Can you see that `S`, `K`, and `I` propagate the actual parameter `arg` to the original occurrence(s) of the formal parameter `x` in the body?

1. Define a function

   ```
   abstr :: (Eq var) ⇒ var → SKI var → SKI var
   ```

   that implements λ-abstraction for SKI terms e.g.

   `abstr 'x' (Free 'x') = I`, `abstr 'x' (Free 'x' ‘App‘ Free 'x') = S I I`, etc.

2. Define a compiler from λ-expressions to SKI machine code:

   ```
   compile :: (Eq var) ⇒ Lambda var → SKI var
   ```

3. Implement a reduction machine that simplifies SKI terms:

   ```
   reduce :: SKI var → [SKI var] → SKI var
   ```

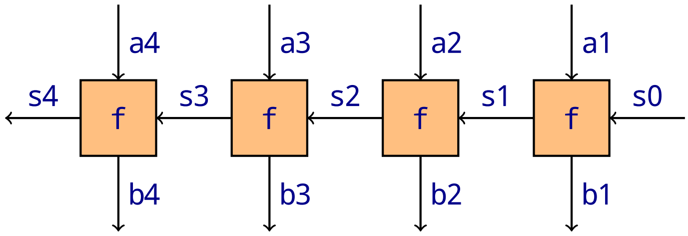
   The second argument of `reduce` serves as a stack. The function traverses the left spine of the binary tree to the leftmost leaf, pushing the visited nodes onto the stack. Then it applies the definitions of `s`, `k`, and `i` as rewrite rules.

4. Test the compiler and the reduction machine on some examples. Do you obtain the same results as in Haskell? In case you are lacking inspiration here are some things to try:

```
⋙ parse expr "(\x.xx)(\x.xx)"
⋙ compile it
⋙ reduce it []
⋙ parse expr "\f.(\x.f(xx))(\x.f(xx))"
⋙ compile it
⋙ reduce it [Free ’s’, Free ’z’]
```

   You may be surprised to learn that, like the λ-calculus, SKI machine code is Turing-complete.

Exercise 5.6 (Programming and hardware design, `Hardware.lhs`). Complex circuits are often assembled from simpler components using some regular "wiring pattern". A simple example is afforded by the ripple carry adder, which implements the school algorithm for addition in hardware. It consists of a series of full adders:



Each full adder takes two summand bits (top input) and a carry (right input), and produces a sum bit (bottom output) and a carry (left output).

1. Capture the wiring scheme as a higher-order function.

```
mapr :: ((a, s) → (b, s)) → (([a], s) → ([b], s))
```

2. Use the higher-order function to implement a ripple carry adder. I propose that you define a tailor-made datatype for binary digits.

```
data Bit = O | I
deriving (Eq, Ord, Show)
```

   Recall that a full adder is defined in terms of two half adders plus some additional circuitry.

Exercise 5.7 (Unfolding, `Unfold.lhs`). Use the higher-order function `unfoldr` to define

1. a function `take :: Int → [a] → [a]` implementing the function `take` from the Prelude;
2. a function `filter :: (a → Bool) → [a] → [a]` implementing the function `filter`;
3. a function `fibs :: [Integer]` that returns the fibonacci sequence $1, 1, 2, 3, 5, 8, \ldots$;
4. a function `primes :: [Integer]` generating a sequence of all prime numbers. The definition can be based on the following alternative implementation

```
primes = sieve [2..] where
        sieve (p:xs) = p : sieve [ n | n ← xs, n ‘mod‘ p /= 0 ]
```

For some functions that produce a list it is hard or even impossible to give a definition in terms of `unfold`. For example, the append function cannot be defined properly,using `unfold`. For this reason we introduce a slightly more general version of `unfold`, which we name `apo` (an abbreviation of the Greek word *apomorphism*)

```
apo :: (t → Either [a] (a, t)) → t → [a]
apo rep seed = produce seed
  where
    produce seed = case rep seed of
       Left l      → l
       Right(a,ns) → a : produce ns
```

Instead of returning a `Maybe`-value, the argument function of `apo` now returns an `Either` value. The recursion of `apo` ends whenever a value `Left l` is produced. The difference with `unfoldr` is that in this case `apo` will return `l` instead of `[]`. The other case is the same as the case for `Just` in `unfoldr`.

5. Give a definition of `unfoldr` in terms of `apo`;

6. use `apo` to define `++`;

7. redefine the function `insert` that inserts a given element in an already sorted list, i.e.;

   ```
   insert :: (Ord a) ⇒ a → [a] → [a]
   insert x [ ] = [x]
   insert x (y : ys)
    | x ≤ y     = x : y : ys
    | otherwise  = y : insert x ys
   ```

   using `apo`

Exercise 5.8 (Notations). Below a number of functions are given. Deduce from each the most general type and explain what the function does.

```
f1 x y       = x y
f2 x y z     = x z (y z)
f3 x y       = x (x y)
f4 x y z     = filter x [y .. z]
f5 x y (z,w) = (x z,y w)
f6           = f5
f7 "-"       = -
f7 "+"       = +
f7 "*"       = *
f7 "/"       = /
```

Exercise 5.9 (With or without curry). Examine the `curry` and `uncurry` functions in `Data.Tuple`.

1. What do these functions do?

2. Deduce the type of the following expressions: `curry fst` and `curry snd`. What are their semantics?

3. Deduce the type of the following expressions: `uncurry (+)`, `uncurry (-)`, `uncurry (*)` and `uncurry (/)`. What are their semantics?

Exercise 5.10 (Function composition). The function composition operator `.` can be used to create a new function from two existing functions. It can be defined as: `f ∘ g = \x → f (g x)`. Explain what the following compositions do:

```
e1 = (* 5)       ∘ (+ 1)
e2 = (+ 1)       ∘ (* 5)
e3 = ( 2)        ∘ (* 2)
e4 = (min 100)   ∘ (max 0)
e5 = (5 <)       ∘ length
e6 = (*)         ∘ ((+) 4)
```

Exercise 5.11 (Flipping arguments). The function `flip` can be used to flip arguments of a function. It can be defined as: `flip f a b = f b a`. Compare the functions below with their "flipped" variant and explain the difference, if any:

1. `(+) 4 2` versus `flip (+) 4 2`.
2. `(-) 4 2` versus `flip (-) 4 2`.
3. `(*) 4 2` versus `flip (*) 4 2`.
4. `(/) 4 2` versus `flip (/) 4 2`.

Exercise 5.12 (Ellipse perimeter). The *perimeter* of an ellipse with radii r_1, r_2 ($r_1 \geq r_2 > 0$) can be approximated by the following series:

$$\begin{aligned}
perimeter &= 2r_1\pi(1 - \sum_{i=1}^{\infty} s_i) \\
s_1 &= \frac{1}{4}e^2 \\
s_i &= s_{i-1} \cdot \frac{(2i-1)(2i-3)}{4i^2} \cdot e^2 \quad \textbf{if } i > 1 \\
e &= \frac{\sqrt{r_1^2 - r_2^2}}{r_1}.
\end{aligned}$$

Write a program that calculates the perimeter of an ellipse with radii r_1 and r_2 ($r_1 \geq r_2 > 0$) up to a desired accuracy. Use `until` from `GHC.Base`.

Exercise 5.13 (Word list). Write a function `words` that receives a list of `Char` and selects all its words. A word is, for the purpose of this exercise, defined as a consecutive sequence of alphanumeric characters. Use `group` from exercise **??** in your definition: call it using a suitable predicate (have a look at `GHC.Char`).

Exercise 5.14 (Origami). Rewrite the following functions using `foldl` or `foldr` and λ-abstractions (look the up on Hoogle if unsure): `sum`, `prod`, `length`, `reverse`, and `takeWhile`. Rename your new functions `sum'`, `prod'`, `length'`, `reverse'`, and `takeWhile'`.

Exercise 5.15 (Any and all). Examine the functions `and`, `or`, `all` and `any` in the standard prelude of Haskell. Explain in your own words what they do. Write the function `and'` which uses `all` and has the same meaning as `and`. Write the function `or'` which uses `any` and has the same meaning as `or`.

`foldl` and `foldr`

The functions `and` and `or` can be expressed using `all` and `any`. Express the function `all` and `any` using both `foldl` and `foldr`. Call these functions `alll`, `allr`, `anyl` and `anyr`.

`foldl` or `foldr`?

Both the `&&` and the `||` operator are *conditional* tests: they inspect first the value of the first argument, and if the second argument is not needed to determine the result, it is not evaluated.

Predict what will happen when `alll`, `allr`, `anyl` and `anyr` are applied to an infinite list of booleans. Do this using the following examples:

```
alll id $ False:repeat True
anyl id $ True:repeat False
allr id $ False:repeat True
anyr id $ True:repeat False
```

Exercise 5.16 (Lift). Deduce the most general type of the following functions and explain what they do:

```
lift0 f          a = f a
lift1 f g1       a = f (g1 a)
lift2 f g1 g2    a = f (g1 a) (g2 a)
lift3 f g1 g2 g3 a = f (g1 a) (g2 a) (g3 a)
```

Exercise 5.17 (Arithmetic sequences). Use only functions from the standard prelude, λ-expressions and list comprehensions in this exercise.

Plus-minus Write a function `plusminus` which, given a list of values $[x_0 \dots x_n]$ $(n \geq 0)$, computes the value $x_0 - x_1 + x_2 - x_3 + \dots$.

Taylor sequence for sine The *Taylor* sequence for the *sine* is defined as:

$$sine\ x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots = \sum_{n=0}^{\infty} \frac{(-1)^n \cdot x^{2n+1}}{(2n+1)!}$$

Implement the function `sine` which approximates this sequence by taking some finite part (use `take`) of the beginning of the infinite list that represents the above computation.

Taylor sequence for cosine The *Taylor* sequence for the *cosine* is defined as:

$$cosine\ x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots = \sum_{n=0}^{\infty} \frac{(-1)^n \cdot x^{2n}}{(2n)!}$$

Implement the function `cosine` which approximates this sequence by taking some finite part (use `take`) of the beginning of the infinite list that represents the above computation.

Gregory-Leibniz sequence for π The *Gregory-Leibniz* sequence to approximate π is defined as:

$$\pi = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \ldots = \sum_{n=0}^{\infty} \frac{4}{(-1)^n \cdot (2n+1)}$$

Implement the function `pi1` which approximates this sequence by taking some finite part (use `take`) of the beginning of the infinite list that represents the above computation.

Nilakantha sequence for π The *Nilakantha* sequence to approximate π is defined as:

$$\pi - 3 = \frac{4}{2 \cdot 3 \cdot 4} - \frac{4}{4 \cdot 5 \cdot 6} + \frac{4}{6 \cdot 7 \cdot 8} - \frac{4}{8 \cdot 9 \cdot 10} + \ldots = \sum_{n=0}^{\infty} \frac{4}{(-1)^n \cdot \prod_{i=2(n+1)}^{2(n+2)} i}$$

Implement the function `pi2` which approximates this sequence by taking some finite part (use `take`) of the beginning of the infinite list that represents the above computation.

Hints to practitioners 5. GHC offers a plethora of extensions to the standard Haskell language. You may have encountered

```
{-# LANGUAGE UnicodeSyntax #-}
```

which enables the use of Unicode characters e.g. `::` for `::` etc. Several of the language extensions can be characterized as *syntactic sugar*: not in any way essential, but nice to have. One of the convenience features is

```
{-# LANGUAGE LambdaCase #-}
```

Here is a use case:

In §2 we have emphasized a type-driven approach to programming. The idea is that the type of a function suggests the code we might want to write. To illustrate the type-driven approach, consider calling the function `fold :: (List Int Bool → Bool) → ([Int] → Bool)` where the non-recursive datatype `List` is defined:

```
data List a b  =  Nil | Cons a b
```

=

The function `fold` is higher-order: it expects a function as an argument. Functions are created using λ-expressions, so we can start coding:

```
fold (\ x → st)
```

How to fill the hole? Well, the argument function consumes an element of a datatype, which suggests using a case analysis. The cases are, of course, dictated by the datatype:

```
fold (\ x → case x of { Nil → st ; Cons age ok  → ste })
```

How to fill the holes? Well, the argument function has to produce a Boolean, an element of a simple enumeration type, which suggests using constructors. So we may replace the first hole by `True` (or by `False`, but these are really the only choices).

```
fold (\ x → case x of { Nil → True ; Cons age ok  → st })
```

The second hole is more interesting because some data is available to play with. The pattern match introduces `age :: Int` and `ok :: Bool`, which we can suitably combine e.g.

```
fold (\ x → case x of { Nil → True ; Cons age ok  → age > 17 && ok })
```

Of course, the specifics depend on the task at hand, but I hope you get the general idea.

Now, functions that consume data are quite frequent. The syntactic nicety mentioned in the beginning allows us to write the combination of `\` and `case` more succinctly,

```
fold (\ case { Nil → True ; Cons age ok  → age > 17 && ok })
```

sparing us the invention of a variable—inventing names is hard.

(Just in case you are curious, here is the definition of `fold`.

```
fold :: (List a ans → ans) → ([a] → ans)
fold alg  =  consume
  where  consume []        =  alg Nil
         consume (x : xs)  =  alg (Cons x (consume xs))
```

It combines the first two arguments of `foldr`, i.e. `##` and `e`, into a single argument, i.e. `alg`: we have `x ## y = alg (Cons x y)` and `e = alg Nil`. The argument `alg :: List a ans → ans` is known as an *algebra*, hence the name. Recall the *slogan*: fold replaces constructors by functions. The cases `Nil -> ...  ; Cons a b -> ...`   `Nil → ... ; Cons a b → ...` define the replacements for `[]` and `:`.)

And if you are curious about the many other language extensions see the GHC Users Guide (https://wiki.haskell.org/GHC).