Assignment 2

Solve the following exercises. Any non-code part of the assignment must be handed in as comments in the source code.

You write multiline comments between `(*` and `*)` and single-line comments using `//`.

Some of the exercises (which are marked) are copied more or less verbatim from the course book, for your convenience, possibly with minor reformulations.

Miscelaneous exercises

Exercise 2.1

Write a function `downto1 : int -> int list` that given an integer n returns the n-element list `[n; n-1; ...; 1]` if $n > 0$ and `[]` otherwise. You must use if-then-else expressions to define the function.

Secondly define the function `downto2` having same semantics as `downto1`. This time you must use pattern matching.

Examples

```
> downto1 5;;
- val it : int list = [5; 4; 3; 2; 1]

> downto2 10;;
- val it : int list = [10; 9; 8; 7; 6; 5; 4; 3; 2; 1]

> downto1 0;;
- val it : int list = []

> downto2 -42;;
- val it : int list = []
```

Exercise 2.2

Write a function `removeOddIdx : 'a list -> 'a list` that given a list `xs` returns a list where all odd-indexed elements of `xs` have been removed.

```
// type annotation most likely required here
> removeOddIdx ([] : int list);;
- val it : int list = []

> removeOddIdx [true];;
- val it : bool list = [true]

> removeOddIdx ["Marry"; "had"; "a"; "little"; "lamb"; "its"; "fleece";
                "was"; "white"; "as"; "snow"];;
- val it : string list = ["Marry"; "a"; "lamb"; "fleece"; "white"; "snow"]
```

Exercise 2.3

Write a function `combinePair : 'a list -> ('a * 'a) list` that given a list `xs` returns the list with elements from `xs` combined into pairs. If `xs` contains an odd number of elements, then the last element is thrown away.

Hint: Use pattern matching

Examples:

```
// type annotation most likely required here
> combinePair ([] : int list);;
- val it : (int * int) list = []

> combinePair [true; false];;
- val it : (bool * bool) list = [(true, false)]

> combinePair ["Marry"; "had"; "a"; "little"; "lamb"; "its"; "fleece";
                "was"; "white"; "as"; "snow"];;
- val it : (string * string) list =
   [("Marry", "had"); ("a", "little"); ("lamb", "its");
    ("fleece", "was"); ("white", "as")]
```

Exercise 2.4 (based on exercise 3.3 from HR)

The set of *complex numbers* is the set of pairs of real numbers.

Define a type `complex` that represents complex numbers with floating point components.

Define a function `mkComplex : float -> float -> complex` that given two floating point numbers return the corresponding complex number.

Define a function `complexToPair : complex -> float * float` that given a complex number (a, b) returns the pair `(a, b)`.

When doing the following exercises make sure that you use let-statements to not have to repeat commonly occurring calculations.

Addition and multiplication

addition and multiplication are defined by:

$$\begin{aligned}(a,b)+(c,d) &= (a+c,b+d)\\ (a,b)*(c,d) &= (ac-bd,bc+ad)\end{aligned}$$

Declare infix functions `|+|` and `|*|` of type `complex -> complex -> complex` for addition and multiplication of complex numbers.

Division and subtraction

The additive and multiplicative inverse of a complex number c, that is $-c$ and $1/c$ respectively, are defined by:

$$\begin{aligned}-(a,b) &= (-a,-b)\\ \frac{1}{(a,b)} &= \begin{cases}\left(\frac{a}{a^2+b^2}\ ,\ \frac{-b}{a^2+b^2}\right) & \text{if } a \neq 0 \text{ or } b \neq 0\\ \\ \textbf{undefined} & \text{otherwise}\end{cases}\end{aligned}$$

Declare infix functions `|-|` and `|/|` of type `complex -> complex -> complex` for subtraction and division of complex numbers.

Hint: Use addition and multiplication with their inverses.

Hint: Do not worry about the $\textbf{undefined}$ cases at all. Your program will crash if you try to get the multiplicative inverse of $(0,0)$, since you will divide by zero, which is completely standard. You may, however, add your own exception to handle this case if you feel that makes your code cleaner.

Exercise 2.7

Write a non-recursive function `explode1 : string -> char list` that given a string `s` returns the list of characters in `s`.

Examples:

```
> explode1 "";;
- val it : char list = []

> explode1 "Hello World!";;
- val it : char list =
      ['H'; 'e'; 'l'; 'l'; 'o'; ' '; 'W'; 'o'; 'r'; 'l'; 'd'; '!']
```

Hint: if `s` is a string then `s.ToCharArray()` returns an array of characters. You can then use `List.ofArray` to turn it into a list of characters.

Now write a recursive function `explode2 : string -> char list` that has the same semanics as `explode` except that you now have to use the string function `s.Chars` (or `.[index]`), where `s` is a string. You can also make use of `s.Remove(0,1)`.

Exercise 2.8

Write a function `implode : char list -> string` that given a list of characters `cs` returns a string with all characters of `cs` in the same order.

Hint: Use `List.foldBack`.

Write a function `implodeRev : char list -> string` that given a list of characters `cs` returns a string with all characterl of `cs` in reverse order. You may not use `List.rev` for this exercise.

Hint: Use `List.fold`.

Examples:

```
> implode [];;
- val it : string = ""

> implode ['H'; 'e'; 'l'; 'l'; 'o'; ' '; 'W'; 'o'; 'r'; 'l'; 'd'; '!'];;
- val it : string = "Hello World!"

> implodeRev [];;
- val it : string = ""

> implodeRev ['H'; 'e'; 'l'; 'l'; 'o'; ' '; 'W'; 'o'; 'r'; 'l'; 'd'; '!'];;
- val it : string = "!dlroW olleH"
```

Exercise 2.9

Write a function `toUpper : string -> string` that given a string `s` returns `s` with all characters in upper case.

Hint: Use `System.Char.ToUpper` to convert characters to upper case. You can do this function in one line using `implode`, `List.map` and `explode`.

This is a good exercise to play around with piping `|>` and function composition `>>`. With piping, for instance, you can start your function with

```
let toUpper s = s |> ...
```

If you use function composition you can even get away without naming your parameter at all:

```
let toUpper = ...
```

Feel free to hand in several solutions to this exercise if you want feedback. It is useful to get familiar with these connectives.

Exercise 2.10

The Ackermann function is a recursive function where both value and number of mutually recursive calls grow rapidly.

Write the function `ack : int * int -> int` that given an integer pair `(m, n)` implements the Ackermann function using pattern matching on the cases of $A(m,\ n)$ as given below.

$$A(m,\ n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,\ 1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1,\ A(m,\ n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

Notice that the Ackermann function is defined for non-negative numbers only.

Make sure that your pattern matching is exhaustive.

Exercise 2.11

The function `time : (unit -> 'a) -> 'a * TimeSpan` takes a function `f`, and then measures the time the computation of `f ()` takes to run and returns the result of the funciton and the real time used for the computation.

```
let time f =
  let start = System.DateTime.Now
  let res = f ()
  let finish = System.DateTime.Now
  (res, finish - start)
```

Try compute `time (fun () -> ack (3, 11))`.

Write a new function

```
timeArg1 : ('a -> 'b) -> 'a -> 'b * TimeSpan
```

that given a function `f` and an argument `a`, times the computation of evaluating the function `f` with argument `a`. Try `timeArg1 ack (3, 11)`.

Hint: You can use the function `time` above if you hide `f a` in a lambda (function).

Exercise 2.12 (based on Exercise 5.4 from HR)

Declare a function `downto3 : (int -> 'a -> 'a) -> int -> 'a -> 'a` such that

$$\text{downto3}\ f\ n\ e = \begin{cases} f\ 1\ (f\ 2\ (\cdots (f\ (n-1)\ (f\ n\ e)) \cdots)) & \text{if } n > 0 \\ e & \text{if } n \le 0 \end{cases}$$

Declare the factorial function `fac : int -> int` by use of `downto3`. The factorial funciton is defined as follows:

$$\begin{aligned} !0 &= 1 \\ !n &= n \cdot !(n-1) \end{aligned}$$

Use `downto3` to declare a function `range : (int -> 'a) -> int -> 'a list` that given a function `g` and an integer `n` returns the list of `[g 1, g 2, ..., g n]` if `n` is positive, and the empty list otherwise.

Hint: By far the cleanest solution for this exercise is to use partial application and function composition.
Hint: There is an identity function `id` (which is just `fun x -> x`) that, depending on your implementation, may come in handy.

Scrabble assignments

The exercises we do here build on what we did last week so make sure to brush up on that before you start. This week we will represents our words as lists instead of functions and we will create squares that can reason about points for the entire word (double word score for instance) and not just the points of single letters.

Sequences of tiles

Last week we modelled words as functions of type `int -> char * int` that allowed us to lookup tiles at all positions in a word. This week we will just use lists.

```
type word = (char * int) list
```

This has the advantage that the length of the word is no longer infinite but is just the length of the list. Lookup time is, however, still linear.

Assignment 2.13

Create the value `hello` of type `word` that spells the word HELLO. Just like last week the letter H is worth four points, O is worth two points, and all other letters are worth one point.

Creating squares

The square functions from last week took a word and a position of the letter in the word that was placed on the square and calculated how many points the square generated. This worked fine for single-letter score squares (like Double Letter Score) but it does not work for squares that calculate points based on the entire word (like Double Word Score). The reason for this is that these tiles work by first computing the single-letter scores for all possible tiles and *then* multiplying the result by some number - we need to be able to keep track of how many points other squares have generated.

Consider the following situation

```
       DLS          TLS          DWS
        |            |            |
        v            v            v
... | H 4 | E 1 | L 1 | L 1 | O 2 | ...
```

where we have the word HELLO placed on the board where the H is placed on a Double Letter Score, the first L is placed on a Triple Letter Score, and the O is placed on a Double Word Score. To calculate the number of points you get here we must first calculate and sum up the individual points for each square, which in this case is $4 \cdot 2 + 1 + 3 \cdot 1 + 1 + 2 = 15$ and *then* multiply the result by two for a total of thirty points. This means that our squares must be able to do two things:

1. Apply several of the functions from last week - a square must be able to both sum up points from letters and multiply the score.
2. Have an ordering of when functions are computed - a square must be able to collect the sum of all letters before multiplying the score.

At this point in the game you may feel that this solution is over-engineered. After all, the rules of Scrabble are well known. We are, however, only scratching the top of the ice berg here and we will move on to have squares that calculate points in much more interesting (or convoluted) ways than the ones we present here. For this to work, our solution needs to be general and parametric, which is what we are doing here.

We will now slowly work towards solving both points mentioned above.

To keep track of the number of points calculated so far we will add an accumulator to our function from last week. The type of this new square functions is defined as follows `type squareFun = word -> int -> int -> int`

that given a word `w`, the position in `w` of the tile that is placed on the square `pos`, an accumulator `acc` containing the number of points computed so far, returns the number of points you get for that square.

Assignment 2.14

Create the square functions `singleLetterScore`, `doubleLetterScore` and `tripleLetterScore` that all have the type `squareFun` such that:

- `singleLetterScore` returns the point value of the tile placed on the square plus the accumulated score
- `doubleLetterScore` returns twice the point value of the tile placed on the square plus the accumulated score
- `tripleLetterScore` returns thrice the point value of the tile placed on the square plus the accumulated score

Examples:

```
> singleLetterScore hello 4 0;;
- val it : int = 2

> doubleLetterScore hello 4 0;;
- val it : int = 4

> tripleLetterScore hello 4 0;;
- val it : int = 6

> singleLetterScore hello 4 42;;
- val it : int = 44

> doubleLetterScore hello 4 42;;
- val it : int = 46

> tripleLetterScore hello 4 42;;
- val it : int = 48
```

Assignment 2.15

Create the square functions `doubleWordScore` and `tripleWordScore` that both have the type `squareFun` such that:

- `doubleWordScore` multiplies the accumulator by two

- `tripleWordScore` multiplies the accumulator by three

```
> doubleWordScore hello 4 0;;
- val it : int = 0

> tripleWordScore hello 4 0;;
- val it : int = 0

> doubleWordScore hello 12345 42;;
- val it : int = 84

> tripleWordScore hello 12345 42;;
- val it : int = 126
```

Assignment 2.16 (optional)

So far we have only focussed on the point values of the letters but the squares have access to the entire word.

Create a square function `oddConsonants` of type `squareFun` that negates the accumulator if there are an odd number of consonants in the word placed over the square.

Hint: Use the function `isConsonant` from Assignment 1.18.

Putting the square functions together

The functions from Assignment 2.15 only modify the accumulator, and this is correct behaviour, but when placing a letter on a Double Word Score, for instance, we want to count the number of points on the tile *and* multiply the final score by two. In effect, the square has two do both things. To solve this squares will be defined as follows

`type square = (int * squareFun) list`

where each element `(priority, f)` of the list contains a function `f` as described above and a priority `priority` that dictates in which order the functions should be applied. The intuition is that we collect all functions from all squares that have tiles placed over them, sort them by their priority (lowest to highest) and run the functions in order.

The Triple Letter Score tile can then be defined as

`let TLS = [(0, tripleLetterScore)]`

as the only thing it does calculate the points for one tile, whereas the Double Word Score tile would be defined as `let DWS = [(0, singleLetterScore); (1, doubleWordScore)]`

meaning that the function will first calculate the single letter score for the tile placed on it and once all squares have done the same for their tiles the double word score will be applied.

For your convenience, all squares (of type `square`) are given below.

```
let SLS : square = [(0, singleLetterScore)];;
let DLS : square = [(0, doubleLetterScore)];;
let TLS : square = [(0, tripleLetterScore)];;

let DWS : square = SLS @ [(1, doubleWordScore)];;
let TWS : square = SLS @ [(1, tripleWordScore)];;
```

Assignment 2.17 (optional)

This assignment is optional, but highly recommended. It is a core function in your Scrabble project and we have now covered enough material for you to be able to implement it. It also tests your mastery of higher-order functions such as `map` , `fold` , and `|>` .

For this assignment you are recommended to use library functions. You are not required to do so, but your solution will get needlessly complicated if you do not. In the standard library you will find the following functions

1. `List.mapi : (int -> 'T -> 'U ) -> 'T list -> U list` that given a function `mapper` and a list `[x0; x1; ...; xn]` returns the list `[mapper 0 x0; mapper 1 x1; ...; mapper n xn]` - this allows `mapper` to operate on an index as well as the elements at that index of the list. Otherwise it works like regular maps.
2. `List.sortBy : ('T -> 'U) -> 'T list -> 'T list` that given a projection function `proj` and a list `lst` returns a sorted version of `lst` where the sorting is based on first applying `proj` to each element of the list. Note that the projection is used only for sorting.

Create a function `calculatePoints : square list -> word -> int` that given a list of squares `squares` and a list of tiles `w` calculates the number of points that `w` gives you when placed over `squares` . The lists `squares` and `w` are of equal length and you do not have to take care of the cases when they are not.

This assignment is at first glance daunting, but with the help of higher-order functions it is actually very short and the steps detailed below can be combined either by using the piping operator `(|>)` or function composition `(>>)` .

- Use `mapi` and `map` , on the outer and the inner list respectively, to create a list of type `((int * (int -> int)) list) list` where the priorities in `squares` have been left intact and the functions have been partially applied with `word` and the correct index.

- Use `fold` to append your list of lists into one single list (turn `((int * (int -> int) list) list` into `(int * (int -> int) list`
- Use `sortBy` to sort this list based on priority (lowest to highest)
- Use `map` to discard the priority leaving you with a list of type `(int -> int) list`
- Use `fold` and function composition (`>>`) to compose all functions in the list resulting in one function of type `int -> int`
- Instantiate the function with `0` for the initial accumulator to calculate your points.

```
> calculatePoints [DLS; SLS; TLS; SLS; DWS] hello;;
- val it : int = 30

> calculatePoints [DLS; DWS; TLS; TWS; DWS] hello;;
- vil it : int = 180
```