

## Aufgabe 1 [\*] Minimum

(Seite 1 / 2)

Implementiere eine Funktion

```
min3 : int -> int -> int -> int
```

die das Minimum dreier Zahlen zurückgibt.

## Aufgabe 2 [\*\*] Minimum von Listen

Implementiere eine Funktion

```
minList : 'a list -> 'a
```

die das Minimum einer beliebigen Liste zurückgibt.

Hinweis: Verwende für Vergleiche `<` : `'a -> 'a -> bool`.

## Aufgabe 3 [\*] Exklusives Oder

Implementiere eine Funktion

```
xor : bool -> bool -> bool
```

die die beiden Argumente mit exklusivem Oder verknüpft.

## Aufgabe 4 [\*\*] Mitternachtsformel

Implementiere eine Funktion

```
mitternacht : float -> float -> float -> float * float
```

die die Mitternachtsformel zur Lösung quadratischer Gleichungen anwendet, also Gleichungen der Form  $ax^2 + bx + c = 0$  löst und das Ergebnis als Zweier-Tupel (Paar) zurückgibt. Was gibt die Prodezur bei nicht-reellen Lösungen zurück?

## Aufgabe 5 [\*] Liste ausgeben

Gib eine Funktion mit folgendem Typ an, welche eine Liste von Strings ausgibt:

```
print_string_list : string list -> unit
```

## Aufgabe 6 [\*\*] Liste schön ausgeben

Gib eine Funktion

```
print_string_list_newline : string list -> unit
```

an, welche eine Liste von Strings ausgibt und dabei zwischen zwei Strings eine neue Zeile ausgibt. Für die leere Liste und nach dem letzten Elemente soll keine neue Zeile ausgegeben werden.

## Aufgabe 7 [\*] Fakultät

Programmiere die Fakultätsfunktion

```
fac : int -> int
```

## Aufgabe 8 [\*] Potenz

Programmiere die Potenzfunktion

```
power : int -> int -> int
```

Hinweis: verwende Rekursion.

---

## Aufgabe 9 [\*] Paare

(Seite 2 / 2)

Programmiere die OCaml-Funktionen

```
fst : 'a * 'b -> 'a  
snd : 'a * 'b -> 'b
```

welche das erste/zweite Element eines Paares zurückliefern.

## Aufgabe 10 [\*] Tripel

Programmiere je eine Funktion um die drei Elemente eines Tripels zu erhalten.

## Aufgabe 11 [\*] Tupel

Warum ist es nicht möglich, eine Funktion zu schreiben, die das erste Element eines beliebigen Tupels zurückliefert?

## Aufgabe 12 [\*\*] Listenfunktionen

Programmiere folgende Funktionen für Listen, ohne dabei auf Funktionen aus dem Modul `List` zurückzugreifen!

(a) [\*\*] Die Funktion `duplicate` verdoppelt alle Einträge einer Liste. Aus der Liste  $[a_1; a_2 \dots a_n]$  berechnet `duplicate` die Liste  $[a_1; a_1; a_2; a_2 \dots a_n, a_n]$ . `duplicate` hat die folgende Signatur:

```
duplicate : 'a list -> 'a list
```

(b) [\*\*] Die Funktion `adjoin` fügt einer Liste `lst` ein Element `el` hinzu, sofern dies nicht bereits in `lst` enthalten ist. `adjoin` hat die folgende Signatur:

```
adjoin : 'a -> 'a list -> 'a list
```

Hinweis: Verwende `=`, um die Elemente zu Vergleichen.

(c) [\*\*] Die Funktion `splitAt` nimmt eine Liste `lst` und eine Integer-Zahl `n` als Argumente. `splitAt` zerlegt `lst` in eine Liste mit den ersten `n` Elementen aus `lst` und eine Liste mit den restlichen Elementen. Beide Teillisten gibt `splitAt` als Tupel zurück. `splitAt` hat die folgende Signatur:

```
splitAt : 'a list -> int -> 'a list * 'a list
```

## Aufgabe 13 [\*\*\*] Mergesort

Programmiere eine Sortierfunktion

```
mergeSort : 'a list -> 'a list
```

welche den Mergesort-Algorithmus verwendet, um eine Liste zu sortieren.

Implementiere dazu eine Hilfsfunktion

```
merge : 'a list -> 'a list -> 'a list
```

welche zwei sortierte Listen zu einer sortierten Liste zusammenfasst.

Hinweis:

- Verwende `<`, um die Elemente zu Vergleichen.
  - Verwende die Funktion `splitAt` aus Aufgabe 12c.
-