

Objective Caml version 3.09.0

```
( * ===== *)
(* Übung 1, Aufgabe 1 a: *)
(* ===== *)

(* ----- *)
(* Für einen konkreten Modul m, etwa m = 5, ist die Addition in  $\mathbb{Z}/m\mathbb{Z}$  eine Funktion mit 2 Para- *)
(* metern, z.B.: *)
(* ----- *)

# let add_mod_5 = fun a -> fun b -> (a + b) mod 5;;

val add_mod_5 : int -> int -> int = <fun>

(* ----- *)
(* oder in "gezuckerter" Schreibweise: *)
(* ----- *)

# let add_mod_5 a b = (a + b) mod 5;;

val add_mod_5 : int -> int -> int = <fun>

(* ----- *)
(* In einem weiteren Abstraktionsschritt führt man den Modul m als zusätzlichen Parameter ein, *)
(* und erhält so die Addition in einem beliebigen Restklassenring  $\mathbb{Z}/m\mathbb{Z}$ : *)
(* ----- *)

# let add_mod = fun m -> fun a -> fun b -> (a + b) mod m;;

val add_mod : int -> int -> int -> int = <fun>

# let add_mod m a b = (a + b) mod m;;

val add_mod : int -> int -> int -> int = <fun>

(* ----- *)
(* Ähnlich definiert man Subtraktion und Multiplikation in  $\mathbb{Z}/m\mathbb{Z}$ . Bei der Subtraktion wird der *)
(* Modul m hinzuaddiert, damit man keine negativen Reste erhält. *)
(* ----- *)

# let sub_mod m a b = (m + a - b) mod m;;

val sub_mod : int -> int -> int -> int = <fun>

# let mul_mod m a b = (a * b) mod m;;

val mul_mod : int -> int -> int -> int = <fun>

(* ----- *)
(* Aus diesen abstrakteren Funktionen erhält man durch partielle Applikation wieder die Opera- *)
(* tionen in einem konkreten Restklassenring, etwa in  $\mathbb{Z}/5\mathbb{Z}$ : *)
(* ----- *)

# let add_mod_5 = add_mod 5;;

val add_mod_5 : int -> int -> int = <fun>

# let sub_mod_5 = sub_mod 5;;

val sub_mod_5 : int -> int -> int = <fun>

# let mul_mod_5 = mul_mod 5;;

val mul_mod_5 : int -> int -> int = <fun>

(* ----- *)
(* Man beachte, dass die Reihenfolge der Parameter in den Funktionen add_mod, sub_mod, mul_mod *)
(* "sinnvoll" gewählt wurde: Der erste Parameter m legt den Restklassenring  $\mathbb{Z}/m\mathbb{Z}$  fest, auf dem *)
(* die Operationen arbeiten, die weiteren Parameter a und b sind die Argumente der jeweiligen *)
(* Operation. *)
(* ----- *)
```

```

(* ----- *)
(* Wählt man die Reihenfolge anders, z.B. m als letzten Parameter, so ist es etwas umständli- *)
(* cher, aus den abstrakten Funktionen die konkreten zu gewinnen, weil man nicht mehr mit par- *)
(* tieller Applikation arbeiten kann. *)
(* ----- *)

# let add_mod' a b m = (a * b) mod m;;

val add_mod' : int -> int -> int -> int = <fun>

# let add_mod_5 = fun a -> fun b -> add_mod' a b 5;;

val add_mod_5 : int -> int -> int = <fun>

# let add_mod_5 a b = add_mod' a b 5;;

val add_mod_5 : int -> int -> int = <fun>

(* ===== *)
(* Übung 1, Aufgabe 1 b: *)
(* ===== *)

(* ----- *)
(* Die Aufgabenstellung ist etwas weit hergeholt, die Lösung sollte etwa so aussehen: *)
(* ----- *)

# let f m a b c =
  let add = add_mod m in
  let mul = mul_mod m in
  add (mul a a) (add (mul b b) (mul c c));;

val f : int -> int -> int -> int -> int = <fun>

(* ===== *)
(* Übung 1, Aufgabe 1 c: *)
(* ===== *)

(* ----- *)
(* Die Potenz  $a^n$  im Restklassenring  $Z/mZ$  erhält man - wie in jedem Ring - durch iterierte *)
(* Multiplikation gemäß der induktiven Definition: *)
(* ----- *)
(*  $a^0 = 1$  *)
(*  $a^n = a * a^{(n-1)}$  falls  $n > 0$  (wobei * hier die Multiplikation modulo m ist) *)
(* ----- *)
(* Die einfachste Implementierung erhält man, indem man diese Definition unmittelbar in ein *)
(* Programm umsetzt: *)
(* ----- *)

# let rec exp_mod m a n =
  let mul = mul_mod m in
  if n = 0
  then 1
  else mul a (exp_mod m a (n - 1));;

val exp_mod : int -> int -> int -> int = <fun>

(* ----- *)
(* Beispiel:  $5^{1000}$  in  $Z/1000$ : *)
(* ----- *)

# exp_mod 1000 5 1000;;

- : int = 625

(* ----- *)
(* Man kann die Abstraktion hier noch weiter treiben: Da das Potenzieren in jedem Ring (sogar *)
(* in jedem Monoid) auf die gleiche Art definiert ist, erhält man eine "generische Potenzfunk- *)
(* tion", indem man die Multiplikation mul und das neutrale Element 1 als zusätzliche Parame- *)
(* ter einführt: *)
(* ----- *)

```

```

# let rec gen_exp mul one a n =
  if n = 0
  then one
  else mul a (gen_exp mul one a (n - 1));;

val gen_exp : ('a -> 'b -> 'b) -> 'b -> 'a -> int -> 'b = <fun>

(* ----- *)
(* Aus dieser allgemeinen Potenzfunktion kann man dann wieder konkrete Potenzfunktionen durch *)
(* partielle Applikation erhalten, z.B. das Potenzieren in Z, Z/mZ, Q, R, C oder im Monoid A* *)
(* für ein Alphabet A. *)
(* ----- *)

# let exp_int = gen_exp ( * ) 1;;

val exp_int : int -> int -> int = <fun>

# exp_int 2 10;;

- : int = 1024

# let exp_mod m = gen_exp (mul_mod m) 1;;

val exp_mod : int -> int -> int -> int = <fun>

# exp_mod 1000 5 1000;;

- : int = 625

# let exp_float = gen_exp ( *. ) 1.0;;

val exp_float : float -> int -> float = <fun>

# exp_float 100. 25;;

- : float = 1e+050

# let exp_string = gen_exp (^) " ";;

val exp_string : string -> int -> string = <fun>

# exp_string "bla " 10;;

- : string = "bla bla bla bla bla bla bla bla bla "

(* ----- *)
(* Übrigens gibt es einen effizienteren Algorithmus zum Potenzieren, bei dem man die folgenden *)
(* Gleichungen ausnutzt: *)
(* a^0 = 1 *)
(* a^n = (a^2)^(n/2) falls n gerade *)
(* a^n = a * (a^2)^(n/2) falls n ungerade *)
(* *)
(* Dabei ist / die ganzzahlige Division. *)
(* ----- *)

# let rec fast_gen_exp mul one a n =
  if n = 0
  then one
  else
    let x = fast_gen_exp mul one (mul a a) (n/2) in
      if n mod 2 = 0 then x else mul a x;;

val fast_gen_exp : ('a -> 'a -> 'a) -> 'a -> 'a -> int -> 'a = <fun>

(* ----- *)
(* Dieser Algorithmus spielt in der Kryptographie eine Rolle, weil man dort - im Zusammenhang *)
(* mit Primzahltests - hohe Potenzen in Restklassenringen bilden muss. *)
(* ----- *)

# let fast_exp_mod m = fast_gen_exp (mul_mod m) 1;;

```

```
val fast_exp_mod : int -> int -> int -> int = <fun>
# fast_exp_mod 1000 5 1000;;
- : int = 625
# fast_exp_mod 1000 5 1000000000;;
- : int = 625
# let fast_exp_float = fast_gen_exp ( *. ) 1.0;;
val fast_exp_float : float -> int -> float = <fun>
# fast_exp_float 0.999999 1000000000;;
- : float = 0.
# fast_exp_float 0.9999999 1000000000;;
- : float = 3.7200575235214152e-044
# fast_exp_float 1.000001 1000000000;;
- : float = infinity
# fast_exp_float 1.0000001 1000000000;;
- : float = 2.688103843211961e+043
#
```