

Objective Caml version 3.06

```

(* ===== *)
(* Aufgabe 1: *)
(* ===== *)

# let rec fact =
  function
    0 -> 1
  | n -> n * fact (n - 1);;

val fact : int -> int = <fun>

# List.map fact [0;1;2;3;4;5;6;7;8;9;10];;

- : int list = [1; 1; 2; 6; 24; 120; 720; 5040; 40320; 362880; 3628800]

(* ===== *)
(* Aufgabe 2: *)
(* ===== *)

# type complex =
  Cartesian of float * float
  | Polar of float * float;;

type complex = Cartesian of float * float | Polar of float * float

(* ----- *)
(* Die Addition wird mit kartesischen Koordinaten durchgeführt, die Multipli- *)
(* kation mit Polarkoordinaten. Die Fallunterscheidung wird in die Funktionen *)
(* x_coord, y_coord, radius und angle gesteckt. So erhält man eine übersicht- *)
(* liche Darstellung für add und mult. *)
(* ----- *)

# let x_coord (c: complex): float =
  match c with
  | Cartesian (x, y) -> x
  | Polar (r, phi) -> r *. cos phi;;

val x_coord : complex -> float = <fun>

# let y_coord (c: complex): float =
  match c with
  | Cartesian (x, y) -> y
  | Polar (r, phi) -> r *. sin phi;;

val y_coord : complex -> float = <fun>

# let radius (c: complex): float =
  match c with
  | Cartesian (x, y) -> sqrt (x *. x +. y *. y)
  | Polar (r, phi) -> r;;

val radius : complex -> float = <fun>

(* ----- *)
(* Zur Berechnung des Winkels gibt es eine passende O'Caml-Funktion atan2. *)
(* Mit atan wäre es mühsam, da man anhand der Vorzeichen von x und y den rich- *)
(* tigen Quadranten bestimmen müsste. *)
(* ----- *)

# let angle (c: complex): float =
  match c with
  | Cartesian (x, y) -> atan2 y x
  | Polar (r, phi) -> phi;;

val angle : complex -> float = <fun>

# let add (c1: complex) (c2: complex): complex =
  Cartesian (x_coord c1 +. x_coord c2, y_coord c1 +. y_coord c2);;

val add : complex -> complex -> complex = <fun>

(* ----- *)
(* Bei der Multiplikation achten wir darauf, dass der Winkel zwischen 0 und *)
(* 2 * pi bleibt. (Darstellungsinvariante) *)
(* ----- *)

```

```

# let two_pi = 8. *. atan 1.;;

val two_pi : float = 6.28318530718

# let mul (c1: complex) (c2: complex): complex =
  Polar (radius c1 *. radius c2, mod_float (angle c1 +. angle c2) two_pi);;

val mul : complex -> complex -> complex = <fun>

(* ----- *)
(* Konstruktoren haben wir schon, nämlich Polar und Cartesian. Die Funktionen *)
(* x_coord, y_coord, radius, angle kann man als Observatoren auffassen. *)
(* ----- *)

(* ===== *)
(* Test-Beispiel: Summe der n-ten Einheitswurzeln *)
(* ===== *)

(* ----- *)
(* Die primitive n-te Einheitswurzel hat die Polarkoordinaten (1, 2 * pi / n). *)
(* Die übrigen n-ten Einheitswurzeln sind die Potenzen dieser Zahl. Die Summe *)
(* der n-ten Einheitswurzeln ist die 0, abgesehen von den unvermeidlichen Run- *)
(* dungsfehlern beim Rechnen mit floats. *)
(* ----- *)

# let root (n: int): complex = Polar (1., two_pi /. float n);;

val root : int -> complex = <fun>

# let one = Cartesian (1., 0.);;

val one : complex = Cartesian (1., 0.)

# let rec exp (c: complex) (i: int): complex =
  if i = 0 then one else mul c (exp c (i - 1));;

val exp : complex -> int -> complex = <fun>

# let zero = Cartesian (0., 0.);;

val zero : complex = Cartesian (0., 0.)

# let rec sum n (f: int -> complex): complex =
  if n < 0 then zero else add (f n) (sum (n - 1) f);;

val sum : int -> (int -> complex) -> complex = <fun>

# let sum_of_roots n = sum (n - 1) (exp (root n));;

val sum_of_roots : int -> complex = <fun>

# sum_of_roots 100;;

- : complex = Cartesian (-3.48610029732e-14, -1.60205182453e-13)

# sum_of_roots 3333;;

- : complex = Cartesian (1.06251007992e-10, 1.08810759882e-11)

(* ===== *)
(* Aufgabe 4: *)
(* ===== *)

# #use "Loesungen_zu_Uebung_4.ml";;

(* ----- *)
(* Im Falle eines Typfehlers soll eine Fehlermeldung ausgegeben werden, die *)
(* folgende Informationen enthält: *)
(* - den Teilausdruck e, an dem die Wohlgetyptheit gescheitert ist, *)
(* - die Typumgebung gamma, in der der Teilausdruck e überprüft wurde. *)
(* *)
(* Zu diesem Zweck definieren wir eine neue exception *)
(* Type_error of type_env * expression *)
(* und ersetzen jeden Aufruf "raise Type_error" in der Definition der Funktion *)
(* exp_type durch "raise (Type_error (gamma, e))", d.h. wir packen die Infor- *)
(* mation einfach auf die geworfene exception. Erst in der Funktion type_check *)
(* wird diese Information wieder aufgefangen und ausgedruckt. *)
(* ----- *)

```

```

(* ----- *)
(* In der Hilfsfunktion lookup wird eine neue exception Unknow_n_identifier be- *)
(* nutzt, die erst in exp_type zu Type_error (gamma, e) "umgewandelt" wird. So *)
(* erhalt man die richtige Typumgebung gamma fur die exception. *)
(* ----- *)

# exception Unknow_n_identifier;;

exception Unknow_n_identifier

# let rec lookup (gamma: type_env) (id: identifier): simple_type =
  match gamma with
  | (id', tau) :: gamma' -> if id = id' then tau else lookup gamma' id
  | _ -> raise Unknow_n_identifier;;

val lookup : type_env -> identifier -> simple_type = <fun>

# exception Type_error of type_env * expression;;

exception Type_error of type_env * expression

# let rec exp_type (gamma: type_env) (e: expression): simple_type =
  match e with
  | Const c -> const_type c
  | Id id ->
    (try lookup gamma id
     with Unknow_n_identifier -> raise (Type_error (gamma, e)))
  | App (e1, e2) ->
    (let tau1 = exp_type gamma e1
     in match tau1 with
      | ARROW (tau, tau') ->
        if tau = exp_type gamma e2
        then tau'
        else raise (Type_error (gamma, e))
      | _ -> raise (Type_error (gamma, e)))
  | If (e0, e1, e2) ->
    if exp_type gamma e0 = BASE BOOL
    then
      let tau1 = exp_type gamma e1
      and tau2 = exp_type gamma e2
      in if tau1 = tau2 then tau1 else raise (Type_error (gamma, e))
    else raise (Type_error (gamma, e))
  | Tuple es -> PRODUCT (List.map (exp_type gamma) es)
  | Lambda (id, tau, e1) ->
    let tau' = exp_type (update gamma tau id) e1
    in ARROW (tau, tau')
  | Let (id, e1, e2) ->
    let tau = exp_type gamma e1
    in exp_type (update gamma tau id) e2
  | Rec (id, tau, e1) ->
    if tau = exp_type (update gamma tau id) e1
    then tau
    else raise (Type_error (gamma, e));;

val exp_type : type_env -> expression -> simple_type = <fun>

# let string_of_entry (id, tau): string =
  string_of_id id ^ ": " ^ string_of_type tau;;

val string_of_entry : identifier * simple_type -> string = <fun>

# let string_of_type_env gamma =
  "[" ^ String.concat ", \n " (List.map string_of_entry gamma) ^ "]";;

val string_of_type_env : (identifier * simple_type) list -> string = <fun>

# let type_check (e: expression): unit =
  print_string ("\nDer Ausdruck\n\n" ^ string_of_exp e ^ "\n\n" ^
    (try "hat den Typ\n\n" ^ string_of_type (exp_type [] e)
     with Type_error (gamma, e') ->
      "ist nicht wohlgetypt: \n\n" ^
      "Die Typuberprufung scheitert am Teilausdruck\n\n" ^
      string_of_exp e' ^ "\n\n" ^
      "in der Typumgebung\n\n" ^
      string_of_type_env gamma) ^ ".\n\n");;

val type_check : expression -> unit = <fun>

```

```
# type_check (Lambda ("x", BASE INT, Id "y"));;
```

Der Ausdruck

```
(lambda x: int. y)
```

ist nicht wohlgetypt:

Die Typüberprüfung scheitert am Teilausdruck

y

in der Typumgebung

```
[x: int].
```

- : unit = ()

```
# type_check (Lambda ("x", BASE INT, App (Id "x", Id "x")));;
```

Der Ausdruck

```
(lambda x: int. (x x))
```

ist nicht wohlgetypt:

Die Typüberprüfung scheitert am Teilausdruck

```
(x x)
```

in der Typumgebung

```
[x: int].
```

- : unit = ()

```
# type_check (Lambda ("x",  
                      BASE INT,  
                      Lambda ("x",  
                               BASE FLOAT,  
                               App (App (Const (Op IAdd), Id "x"), Const (Int 1)))));;
```

Der Ausdruck

```
(lambda x: int. (lambda x: float. ((+ x) 1)))
```

ist nicht wohlgetypt:

Die Typüberprüfung scheitert am Teilausdruck

```
(+ x)
```

in der Typumgebung

```
[x: float,  
 x: int].
```

- : unit = ()

#