

Objective Caml version 3.06

```

(* ===== *)
(* Aufgabe 1: *)
(* *)
(* Um eine halbwegs effiziente Implementierung zu erhalten, stellen wir Mengen *)
(* als aufsteigend geordnete Listen dar, also in der Form [x_1, ..., x_n] mit *)
(* n >= 0 und x_1 < ... < x_n. Wir nehmen dabei an, dass die Operation < auf *)
(* dem Elementtyp 'a eine TOTALE Ordnung ist (d.h. dass für zwei Elemente x,y *)
(* vom Typ 'a stets x < y, x = y oder x > y gilt). *)
(* *)
(* Bei der Implementierung der gewünschten Funktionen müssen wir wieder darauf *)
(* achten, dass die Darstellungsinvariante stets erfüllt ist, d.h. dass tat- *)
(* sächlich nur aufsteigend geordnete Listen entstehen können. *)
(* ===== *)

# type 'a set = 'a list;;

type 'a set = 'a list

(* ----- *)
(* Zuerst implementieren wir insert und delete, da wir insert zur Implementie- *)
(* rung des Mengenkonstruktors set benötigen. *)
(* ----- *)

# let rec insert (x: 'a) (s: 'a set): 'a set =
  match s with
  | [] -> [x]
  | y :: s' ->
    if x < y
    then x :: s
    else if x = y then s else y :: insert x s';;

val insert : 'a -> 'a set -> 'a set = <fun>

# let rec delete (x: 'a) (s: 'a set): 'a set =
  match s with
  | [] -> []
  | y :: s' ->
    if x < y
    then s
    else if x = y then s' else y :: delete x s';;

val delete : 'a -> 'a set -> 'a set = <fun>

# let rec set (l: 'a list): 'a set =
  match l with
  | [] -> []
  | x :: l' -> insert x (set l');;

val set : 'a list -> 'a set = <fun>

# let s1 = set [3;6;1;3;1;0;6;7;1];;

val s1 : int set = [0; 1; 3; 6; 7]

(* ----- *)
(* union könnte man mit Hilfe von insert definieren, effizienter ist aber das *)
(* Zusammenfügen der beiden geordneten Listen nach dem 'merge'-Verfahren. *)
(* ----- *)

# let rec union (s1: 'a set) (s2: 'a set): 'a set =
  match (s1, s2) with
  | ([], _) -> s2
  | (_, []) -> s1
  | (x1 :: s1', x2 :: s2') ->
    if x1 < x2
    then x1 :: union s1' s2
    else if x1 = x2 then x1 :: union s1' s2' else x2 :: union s1 s2';;

val union : 'a set -> 'a set -> 'a set = <fun>

# let s2 = set [0;4;6;5;4;7;6;5];;

```

```

val s2 : int set = [0; 4; 5; 6; 47]
# let u12 = union s1 s2;;

val u12 : int set = [0; 1; 3; 4; 5; 6; 7; 47]

(* ----- *)
(* Analog zu union implementiert man intersection und difference. *)
(* ----- *)

# let rec intersection (s1: 'a set) (s2: 'a set): 'a set =
  match (s1, s2) with
  | ([], _) -> []
  | (_, []) -> []
  | (x1 :: s1', x2 :: s2') ->
    if x1 < x2
    then intersection s1' s2
    else if x1 = x2 then x1 :: intersection s1' s2' else intersection s1 s2';;

val intersection : 'a set -> 'a set -> 'a set = <fun>

# let i12 = intersection s1 s2;;

val i12 : int set = [0; 6]

# let rec difference (s1: 'a set) (s2: 'a set): 'a set =
  match (s1, s2) with
  | ([], _) -> []
  | (_, []) -> s1
  | (x1 :: s1', x2 :: s2') ->
    if x1 < x2
    then x1 :: difference s1' s2
    else if x1 = x2 then difference s1' s2' else difference s1 s2';;

val difference : 'a set -> 'a set -> 'a set = <fun>

# let d12 = difference s1 s2;;

val d12 : int set = [1; 3; 7]

(* ----- *)
(* member könnte man einfach durch die entsprechende Listenoperation List.mem *)
(* implementieren, effizienter wird es aber, wenn man die Ordnung ausnutzt. *)
(* ----- *)

# let rec member (x: 'a) (s: 'a set): bool =
  match s with
  | [] -> false
  | y :: s' -> x = y || (x > y && member x s');;

val member : 'a -> 'a set -> bool = <fun>

# List.map (fun x -> member x s1) [0;1;2;3;4;5;6;7;8;9];;

- : bool list =
[true; true; false; true; false; false; true; true; false; false]

# let is_empty (s: 'a set): bool = s = [];;

val is_empty : 'a set -> bool = <fun>

(* ----- *)
(* disjoint und subset könnte man auf member zurückführen, aber man erhält *)
(* eine effizientere Implementierung, wenn man die Ordnung ausnutzt. *)
(* ----- *)

# let rec disjoint (s1: 'a set) (s2: 'a set): bool =
  match (s1, s2) with
  | ([], _) -> true
  | (_, []) -> true
  | (x1 :: s1', x2 :: s2') ->
    (x1 < x2 && disjoint s1' s2) || (x1 > x2 && disjoint s1 s2');;

val disjoint : 'a set -> 'a set -> bool = <fun>

# disjoint s1 s2;;

- : bool = false

```

```

# let rec subset (s1: 'a set) (s2: 'a set): bool =
  match (s1, s2) with
  | ([], _) -> true
  | (_, []) -> false
  | (x1 :: s1', x2 :: s2') ->
    (x1 = x2 && subset s1' s2') || (x1 > x2 && subset s1 s2');

val subset : 'a set -> 'a set -> bool = <fun>

# subset s1 s2;;

- : bool = false

(* ----- *)
(* Da die Darstellung einer Menge als aufsteigend geordnete Liste eindeutig *)
(* ist, kann man equal als die Gleichheit von Listen implementieren. *)
(* ----- *)

# let equal: 'a set -> 'a set -> bool = (=);;

val equal : 'a set -> 'a set -> bool = <fun>

# equal s1 s2;;

- : bool = false

(* ----- *)
(* Die höheren Funktionen map, exists, forall und filter kann man auf die ent- *)
(* sprechenden Listenfunktionen zurückführen, weil sich die Ordnung hier nicht *)
(* Effizienz-Steigerung ausnutzen lässt. Man beachte dabei, dass List.filter *)
(* die Darstellungsinvariante erhält, aber List.map im allgemeinen nicht, also *)
(* muss man sie mit Hilfe des Konstruktors set wiederherstellen. *)
(* ----- *)

# let map (f: 'a -> 'b) (s: 'a set): 'b set = set (List.map f s);;

val map : ('a -> 'b) -> 'a set -> 'b set = <fun>

# map (fun x -> x mod 4) s1;;

- : int set = [0; 1; 2; 3]

# let exists: ('a -> bool) -> 'a set -> bool = List.exists;;

val exists : ('a -> bool) -> 'a set -> bool = <fun>

# exists (fun x -> x mod 4 = 0) s1;;

- : bool = true

# let forall: ('a -> bool) -> 'a set -> bool = List.for_all;;

val forall : ('a -> bool) -> 'a set -> bool = <fun>

# forall (fun x -> x mod 2 = 0) s1;;

- : bool = false

# let filter: ('a -> bool) -> 'a set -> 'a set = List.filter;;

val filter : ('a -> bool) -> 'a set -> 'a set = <fun>

# filter (fun x -> x mod 4 = 0) s1;;

- : int set = [0]

(* ----- *)
(* Als Observer wählen wir eine Funktion, die Mengen in Listen umwandelt, *)
(* z.B. die Identität. Eine solche Funktion braucht man natürlich nur, wenn *)
(* man einen Mechanismus zur Datenabstraktion verwendet, etwa Klassen oder *)
(* Module. *)
(* ----- *)

# let list_of_set (s: 'a set): 'a list = s;;

val list_of_set : 'a set -> 'a list = <fun>

```

```

(* ----- *)
(* PS: Anstelle der Operation < kann man auch die Funktion compare benutzen. *)
(* compare x y liefert -1 falls x < y, 0 falls x = y und 1 falls x > y. Durch *)
(* pattern-matching mit diesen 3 Werten kann man sich dann die if-then-else- *)
(* Schachtelungen ersparen, z.B.: *)
(* ----- *)

# let rec union (s1: 'a set) (s2: 'a set): 'a set =
  match (s1, s2) with
  | ([], _) -> s2
  | (_, []) -> s1
  | (x1 :: s1', x2 :: s2') ->
    match compare x1 x2 with
    | 0 -> x1 :: union s1' s2'
    | 1 -> x2 :: union s1 s2'
    | _ -> x1 :: union s1' s2;;

val union : 'a set -> 'a set -> 'a set = <fun>

# let u12 = union s1 s2;;

val u12 : int set = [0; 1; 3; 4; 5; 6; 7; 47]

(* ===== *)
(* Aufgaben 3 und 2: *)
(* ===== *)

# type identifier = string;;

type identifier = string

# type expression =
  Id of identifier
| App of expression * expression
| Lambda of identifier * expression;;

type expression =
  Id of identifier
| App of expression * expression
| Lambda of identifier * expression

(* ----- *)
(* Die Funktion exp_to_string wandelt die abstrakte Syntax eines Ausdrucks e *)
(* in die (vollständig geklammerte) konkrete Syntax um. *)
(* ----- *)

# let rec exp_to_string (e: expression): string =
  match e with
  | Id id -> id
  | App (e1, e2) -> "(" ^ exp_to_string e1 ^ " " ^ exp_to_string e2 ^ ")"
  | Lambda (id, e1) -> "(lambda " ^ id ^ " . " ^ exp_to_string e1 ^ "));";

val exp_to_string : expression -> string = <fun>

(* ----- *)
(* Beispiel: Die Ausdrücke aus Aufgabe 2. Da wir keine Konstanten vorgesehen *)
(* haben, fassen wir 1, 2 und + als Namen auf. *)
(* ----- *)

# let sum = Lambda ("x", Lambda ("y", App (App (Id "+", Id "x"), Id "y")));;

val sum : expression =
  Lambda ("x", Lambda ("y", App (App (Id "+", Id "x"), Id "y")))

# let identity = Lambda ("x", Id "x");;

val identity : expression = Lambda ("x", Id "x")

# let exp_2a = App (App (sum, Id "2"), App (identity, Id "1"));;

val exp_2a : expression =
  App
  (App (Lambda ("x", Lambda ("y", App (App (Id "+", Id "x"), Id "y"))),
    Id "2"),
  App (Lambda ("x", Id "x"), Id "1"))

# let exp_2b = App (Lambda ("y", App (App (sum, Id "y"), Id "1")), Id "2");;

```

```

val exp_2b : expression =
  App
    (Lambda ("y",
      App
        (App (Lambda ("x", Lambda ("y", App (App (Id "+", Id "x"), Id "y"))),
          Id "y"),
          Id "1")),
      Id "2")

# exp_to_string exp_2a;;

- : identifier =
"(((lambda x . (lambda y . ((+ x) y))) 2) ((lambda x . x) 1))"

# exp_to_string exp_2b;;

- : identifier =
"((lambda y . (((lambda x . (lambda y . ((+ x) y))) y) 1)) 2)"

(* ----- *)
(* Mit den Mengenoperationen lässt sich die Funktion free implementieren, *)
(* indem man einfach die Definition abschreibt. *)
(* ----- *)

# let rec free (e: expression): identifier set =
  match e with
  | Id id      -> set [id]
  | App (e1, e2) -> union (free e1) (free e2)
  | Lambda (id, e) -> delete id (free e);;

val free : expression -> identifier set = <fun>

# free exp_2a;;

- : identifier set = ["+"; "1"; "2"]

# free exp_2b;;

- : identifier set = ["+"; "1"; "2"]

(* ----- *)
(* Zur Implementierung von subst benötigen wir eine Funktion, die uns bei Be- *)
(* darf einen neuen Namen liefert: new_name id s liefert den ersten Namen in *)
(* der Folge id, id', id'', ..., der nicht in s liegt, z.B. liefert der Aufruf *)
(* new_name y {x, y, y', z} den Namen y'' *)
(* ----- *)

# let rec new_name (id: identifier) (s: identifier set): identifier =
  if member id s then new_name (id ^ "'") s else id;;

val new_name : identifier -> identifier set -> identifier = <fun>

# new_name "y" (set ["x"; "y"; "y"; "z"]);;

- : identifier = "y'"

(* ----- *)
(* subst e' id e liefert den Ausdruck e'[e/id], der durch Substitution von id *)
(* durch e aus e' entsteht. Auch hier braucht man nur noch die Definition von *)
(* e'[e/id] abzuschreiben. *)
(* ----- *)

# let rec subst (e': expression) (id: identifier) (e: expression): expression =
  match e' with
  | Id id' -> if id = id' then e else e'
  | App (e1, e2) -> App (subst e1 id e, subst e2 id e)
  | Lambda (id', e1) ->
    if id = id'
    then e'
    else
      let id'' = new_name id' (insert id (union (free e) (free e')))
      in Lambda (id'', subst (subst e1 id' (Id id'')) id e);;

val subst : expression -> identifier -> expression -> expression = <fun>

# exp_to_string (subst exp_2a "+" (Id "*"));;

```

```

- : identifier =
"(((lambda x . (lambda y . ((* x) y))) 2) ((lambda x . x) 1))"

(* ----- *)
(* Die Funktion beta liefert zu einem beta-Redex (lambda id. e1) e2 die zuge- *)
(* h"orige rechte Seite el[e2/id] der beta-Regel. F"ur jeden anderen Ausdruck *)
(* wirft sie die exception Not_a_redex. *)
(* ----- *)

# exception Not_a_redex;;

exception Not_a_redex

# let beta e =
  match e with
  | App (Lambda (id, e1), e2) -> subst e1 id e2
  | _ -> raise Not_a_redex;;

val beta : expression -> expression = <fun>

# exp_to_string (beta exp_2a);;

Exception: Not_a_redex.

# exp_to_string (beta exp_2b);;

- : identifier = "(((lambda x . (lambda y . ((+ x) y))) 2) 1)"

(* ----- *)
(* beta-reducts e liefert die Liste aller Ausdr"ucke e', die durch EINEN beta- *)
(* Reduktionsschritt aus e entstehen (die sogenannten beta-REDUKTE). Im Falle *)
(* e = e1 e2 gibt es f"ur e' drei M"oglichkeiten: Entweder ist e' = e1' e2 f"ur *)
(* ein beta-Redukt e1' von e1 oder e' = e1 e2' f"ur ein beta-Redukt e2' von e2 *)
(* oder e ist selbst ein beta-Redex und e' ist die zugeh"orige rechte Seite. *)
(* Diese drei M"oglichkeiten sammeln wir in den Teillisten left_reducts, right- *)
(* reducts und new_reduct, die wir dann zur Gesamtliste zusammenf"ugen. *)
(* ----- *)

# let rec beta_reducts (e: expression): expression list =
  match e with
  | Id _ -> []
  | App (e1, e2) ->
    let left_reducts = List.map (fun e -> App (e, e2)) (beta_reducts e1)
        and right_reducts = List.map (fun e -> App (e1, e)) (beta_reducts e2)
        and new_reduct = try [beta (App (e1, e2))] with Not_a_redex -> []
    in left_reducts @ right_reducts @ new_reduct
  | Lambda (id, e1) ->
    List.map (fun e -> Lambda (id, e)) (beta_reducts e1);;

val beta_reducts : expression -> expression list = <fun>

# List.map exp_to_string (beta_reducts exp_2a);;

- : identifier list =
["((lambda y . ((+ 2) y)) ((lambda x . x) 1));"
 "(((lambda x . (lambda y . ((+ x) y))) 2) 1)"]

# List.map exp_to_string (beta_reducts exp_2b);;

- : identifier list =
["((lambda y . ((lambda y' . ((+ y) y')) 1)) 2);"
 "(((lambda x . (lambda y . ((+ x) y))) 2) 1)"]

(* ----- *)
(* Eine beta-Reduktion (= Folge von beta-Reduktionsschritten) stellen wir als *)
(* string der Form "e_1 -> e_2 -> ... -> e_n" dar. *)
(* ----- *)
(* beta-reductions e liefert die Liste aller beta-Reduktionen von e (was nur *)
(* m"oglich ist, wenn e keine unendlichen Reduktionen besitzt). Wenn e keine *)
(* beta-Redukkte besitzt, so ist die Folge e (mit 0 Reduktionsschritten) die *)
(* einzige beta-Reduktion von e. Andernfalls ist jede beta-Reduktion f"ur e von *)
(* der Form e -> e_1 -> ... -> e_n, wobei e_1 -> ... -> e_n beta-Reduktion f"ur *)
(* ein beta-Redukt e_1 von e ist. *)
(* ----- *)

# type reduction = string;;

type reduction = string

```

```

# let rec beta_reductions (e: expression): reduction list =
  match beta_reductions e with
  | [] -> [exp_to_string e]
  | es ->
      let l = List.concat (List.map beta_reductions es)
      in List.map (fun (s: reduction) -> exp_to_string e ^ "\n -> " ^ s) l;;

val beta_reductions : expression -> reduction list = <fun>

(* ----- *)
(* print_reductions e gibt die Liste der beta-Reduktionen für e in halbwegs *)
(* lesbarer Form aus. *)
(* ----- *)

# let print_reductions (e: expression) =
  print_string ("Alle beta-Reduktionen für " ^ exp_to_string e ^ ":\n\n" ^
    String.concat "\n\n" (beta_reductions e) ^ "\n\n");;

val print_reductions : expression -> unit = <fun>

# print_reductions exp_2a;;

Alle beta-Reduktionen für ((lambda x . (lambda y . ((+ x) y))) 2) ((lambda x . x) 1):

((lambda x . (lambda y . ((+ x) y))) 2) ((lambda x . x) 1)
-> ((lambda y . ((+ 2) y)) ((lambda x . x) 1))
-> ((lambda y . ((+ 2) y)) 1)
-> ((+ 2) 1)

((lambda x . (lambda y . ((+ x) y))) 2) ((lambda x . x) 1)
-> ((lambda y . ((+ 2) y)) ((lambda x . x) 1))
-> ((+ 2) ((lambda x . x) 1))
-> ((+ 2) 1)

((lambda x . (lambda y . ((+ x) y))) 2) ((lambda x . x) 1)
-> ((lambda x . (lambda y . ((+ x) y))) 2) 1)
-> ((lambda y . ((+ 2) y)) 1)
-> ((+ 2) 1)

- : unit = ()

# print_reductions exp_2b;;

Alle beta-Reduktionen für ((lambda y . (((lambda x . (lambda y . ((+ x) y))) y) 1)) 2):

((lambda y . (((lambda x . (lambda y . ((+ x) y))) y) 1)) 2)
-> ((lambda y . (((lambda y' . ((+ y) y')) 1)) 2)
-> ((lambda y . ((+ y) 1)) 2)
-> ((+ 2) 1)

((lambda y . (((lambda x . (lambda y . ((+ x) y))) y) 1)) 2)
-> ((lambda y . (((lambda y' . ((+ y) y')) 1)) 2)
-> ((lambda y' . ((+ 2) y')) 1)
-> ((+ 2) 1)

((lambda y . (((lambda x . (lambda y . ((+ x) y))) y) 1)) 2)
-> ((lambda x . (lambda y . ((+ x) y))) 2) 1)
-> ((lambda y . ((+ 2) y)) 1)
-> ((+ 2) 1)

- : unit = ()

#

```