```
fun gcd(x:int, y:int):int =
    if y = 0
    then x
    else gcd(y, x mod y);

 fun sestej(a: int * int, b: int * int): int * int =
    let
        val imenovalec_a = #2 a
        val stevec_a = #1 a
        val imenovalec_b = #2 b
        val stevec_b = #1 b

        fun okrajsaj(ulomek: int * int): int * int =
            let val nsd = gcd ulomek
            in
                (#1 ulomek div nsd, #2 ulomek div nsd)
            end
    in
        okrajsaj(stevec_a*imenovalec_b + stevec_b*imenovalec_a, imenovalec_a*imenovalec_b)
    end;

fun vsebuje(el: int, sez: int list): bool =
    if null sez
    then false
    else el = hd sez orelse vsebuje(el, tl sez);

fun zadnji(sez: int list): int =
    if null (tl sez)
    then hd sez
    else zadnji (tl sez);

fun dolzina(sez: int list): int =
    if null sez
    then 0
    else 1 + dolzina(tl sez);

fun vrni_ntega(sez: string list, n: int): string =
    if n = 1
    then hd sez
    else vrni_ntega(tl sez, n - 1);

fun obrni(sez: int list): int list =
    if null sez
    then []
    else obrni(tl sez) @ [hd sez];

fun brisi(el: int, sez: int list): int list =
    if null sez then []
    else if el = hd sez
         then brisi(el, tl sez)
         else hd sez :: brisi(el, tl sez);

fun vsota(s: real list, p: real list): real list =
    if null p
    then s
    else if null s
         then p
         else hd s + hd p :: vsota(tl s, tl p);

fun jePalindrom(sez: int list): bool =
    null sez orelse let
                        val zes = obrni sez

                        fun enaka(s1: int list, s2: int list): bool=
                            null s1 orelse hd s1 = hd s2 andalso enaka(tl s1, tl s2)
                    in
                        enaka(sez, zes)
                    end;

fun zdruzi(s1: int list, s2: int list): int list =
    if null s1
    then s2
    else if null s2
         then s1
         else if hd s1 > hd s2
              then hd s2 :: zdruzi(s1, tl s2)
              else hd s1 :: zdruzi(tl s1, s2);
```

```
fun polozaj(n:int, sez: int list) =
    if null sez
    then NONE
    else
        if hd sez = n
        then SOME 1
        else
            let
                val x = polozaj(n, tl sez)
            in
                if isSome(x)
                then SOME(valOf(x) + 1)
                else NONE
            end;

(*
    x = sqrt(n)
    x^2 = n
    (y + e)^2 = n
    y^2 + 2ey + e^2 = n
    e(2y + e) = n - y^2
    e = (n - y^2)/(2y - e)
    e << y
    e = (n - y^2)/(2y)

    y + e = y + (n - y^2)/(2y) = ... = 0.5 * (y + n/y)
*)
fun koren(n: real, eps: real) =
    if n < 0.0
    then NONE
    else
        let
            fun racunaj(y: real) =
                if abs((n - y*y) / (2.0*y)) < eps
                then y
                else racunaj(0.5 * (y + n/y))
        in
            SOME(racunaj(1.0))
        end;

(********************************** number **************************************************)
datatype number =
    Int of int
    | Real of real;

fun sestej(a: number, b: number) =
    case (a, b) of
        (Int ai, Int bi)      => Int(ai + bi)
        | (Real ai, Real bi)    => Real(ai + bi)
        | (Int ai, Real bi)     => Real(Real.fromInt(ai) + bi)
        | (Real ai, Int bi)     => Real(ai + Real.fromInt(bi));

fun toString(n: number) =
    case n of
        Int x  => Int.toString x
        |Real x => Real.toString x;

(************************************** BSTrees = BSForest? *****************************************)

datatype bstree = Nil
    | Node of {key:int, left: bstree, right: bstree};

fun vstavi(n:int, tree:bstree): bstree =
    case tree of
        Nil                           => Node {key = n, left = Nil, right = Nil}
        |Node {key=k, left=l, right=r} => case Int.compare(n, k) of
                                           EQUAL    => tree
                                          |LESS     => Node {key=k, left=vstavi(n, l), right=r}
                                          |GREATER => Node {key=k, left=l, right=vstavi(n, r)};

(*
    levo podrevo je urejeno, desno poddrevo je urejeno,
    zato samo zdruzimo poVrsti od levo, k in poVrsti od desno
*)
fun poVrsti(tree: bstree) =
    case tree of
        Nil                              => []
        | Node {key=k, left=l, right=r} => poVrsti(l) @ [k] @ poVrsti(r);

fun visina(tree: bstree) =
    case tree of
        Nil => 0
        | Node {key=k, left=l, right=r} => Int.max(visina(l), visina(r)) + 1;

fun jeBstree(tree: bstree) =
    let
        fun veljavno(t: bstree, min: int, max: int) =
            case t of
                Nil                              => true
                | Node {key=k, left=l, right=r} => min < k andalso
                                                   k < max andalso
                                                   veljavno(l, min, k) andalso
                                                   veljavno(r, k, max)

    in
        veljavno(tree, valOf(Int.minInt), valOf(Int.maxInt))
    end;

fun praznoDrevo(tree: bstree) =
    case tree of
        Nil       => true
        | Node nd => false;

fun stElementov(tree: bstree) =
    case tree of
        Nil       => 0
        | Node nd => 1 + stElementov(#left nd) + stElementov(#right nd);

fun stListov(tree: bstree) =
    case tree of
        Nil       => 0
        | Node nd => if praznoDrevo(#left nd) andalso praznoDrevo(#right nd)
                     then 1
                     else stListov(#left nd) + stListov(#right nd);

(* test bstree *)
(*
val t1 = Node {key=4, left=Nil, right=Node({key=6, left=Nil, right=Nil})};
val t2 = vstavi(3, t1);
val t2sez = poVrsti(t2);
val t3 = vstavi(2,t2);
val t3sez = poVrsti(t3);

visina(t2);
visina(t3);

val falseT = Node {key=4, left=Nil, right=Node({key=3, left=Nil, right=Nil})};
jeBstree(t3);
jeBstree(falseT);

praznoDrevo(t3);
praznoDrevo(Nil);

stElementov(t3);
stListov(t3);
*)

(* fun sestej *)
fun sestavi(sez : int list, vsota : int) =
    let
        fun izracunajVsoto(sez : int list, vsota : int, rez : int list) =
            if vsota = 0
            then rez
            else if null sez
```

```
                 then []
                 else let
                          val prviSeznam = izracunajVsoto(tl sez, vsota, rez)
                      in
                          if prviSeznam <> []
                          then prviSeznam
                          else izracunajVsoto(tl sez, vsota - hd sez, hd sez :: rez)
                      end
    in
        let
            val rez = izracunajVsoto(sez, vsota, [])
        in
            if null rez
            then NONE
            else SOME rez
        end
    end;

(* test sestavi *)
(*
sestavi([2,3,4,1], 6);
*)

(* fun casovniRazpon *)
type cas = (int * int * int)

fun casovniRazpon(c1 : cas, c2 : cas) =
        let
            val h1 = (#1 c1) val h2 = (#1 c2)
            val m1 = (#2 c1) val m2 = (#2 c2)
            val s1 = (#3 c1) val s2 = (#3 c2)

            fun odstejMinute(c1 : cas, c2 : cas) =
                let
                    val h1 = (#1 c1) val h2 = (#1 c2)
                    val m1 = (#2 c1) val m2 = (#2 c2)
                    val s1 = (#3 c1) val s2 = (#3 c2)
                in
                    if (m2 - m1) < 0
                    then ((h2 - (h1 + 1)) mod 24, (m2 - m1) mod 60, s1)
                    else ((h2 - h1) mod 24, (m2 - m1) mod 60, s1)
                end

        in
            if (s2 - s1)  < 0
            then odstejMinute((s1, m1 + 1, (s2 - s1) mod 60), c2)
            else odstejMinute((s1, m1, (s2 - s1) mod 60), c2)
        end;



(* test casovniRazpon *)
(* razlicne relacije med urami, minutami in sekundami *)
(*casovniRazpon((0, 0, 0),(1, 1, 1));
casovniRazpon((0, 0, 1),(1, 1, 0));
casovniRazpon((0, 1, 0),(1, 0, 1));
casovniRazpon((0, 1, 1),(1, 0, 0));
casovniRazpon((1, 0, 0),(0, 1, 1));
casovniRazpon((1, 0, 1),(0, 1, 0));
casovniRazpon((1, 1, 0),(0, 0, 1));
casovniRazpon((1, 1, 1),(0, 0, 0));*)

(* se nekaj enakih polj *)
(*casovniRazpon((0, 0, 0),(0, 0, 1));
casovniRazpon((0, 0, 1),(0, 0, 0));
casovniRazpon((0, 0, 1),(1, 1, 0));*)

(* se nekaj bolj normalnih ur *)
(*casovniRazpon((8, 0  , 0),(7, 33, 33));
casovniRazpon((6, 35, 0),(7, 33, 33));
casovniRazpon((8, 35, 0),(7, 33, 33));
casovniRazpon((7, 35, 0),(7, 33, 33));*)
```

```
(* zapiski - ponovitev
        exception Cons
                Cons_2 of Type

(raise Cons_2 r) handle p1 =>
                                                 | p2 =>
                                                 | Cons_2 x => x *)

fun sin (x, eps) =
        let fun racunaj (prejsnji, i, acc) =
                let
                        val naslednji = ((~1.0) * prejsnji * (x*x)) / (Real.fromInt(i+1) * Real.fromInt
(i+2))
                in
                        if prejsnji < eps
                        then acc
                        else racunaj(naslednji, i+1, acc+prejsnji)
                end
        in racunaj(x, 1, 0.0)
        end


fun fib n =
   let fun fib1 (n1, n2, i) =
       if i = n
       then n1 + n2
       else fib1(n2, n1 + n2, i + 1)
   in if n < 1
       then 0
       else fib1(1, 0, 1)
   end

exception NapacnaDolzina

fun sestavi3 terka =
        case terka of
                ([], [], [])                   => []
                | (h1::t1, h2::t2, h3::t3) => (h1, h2, h3)::sestavi3 (t1, t2, t3)
                |_                                               => raise NapacnaDolzina;

(* Racunanje tipov
        sestavi3: T1 -> T2
        terka: T3; T1 = T3
        T3 = T4 list * T5 list * T6 list
        [] -> T2 = T7 list
        h1: T8, h2:T9, h3:T10
        t1: T8 list, t2: T9 list, t3: T10 list
        ----- iz tega sledi:
        T8 = T4, T9 = T5, T10 = T6

        (T4 list * T5 list * T6 list) -> (T4 * T5 * T6) list
        ('a list * 'b list * 'c list) -> ('a * 'b * 'c) list
*)

fun razstavi3 sez =
        let fun razstavljaj (sez, s1, s2, s3) =
                case sez of
                        []                      => razstavljaj ([], rev s1, rev s2, rev s3)
                        |(a,b,c)::t => razstavljaj (t, a::s1, b::s2, c::s3)
        in razstavljaj (sez, [], [], [])
        end

fun naLihih (f, sez) =
        case sez of
                []                      => []
                | h::[]          => [f h]
                | h1::h2::t => (f h1)::naLihih (f, t);

fun veljaNaVseh (f, sez) =
        case sez of
                []          => true
                | a::t => (f a) andalso (veljaNaVseh (f, t))
```

```
val vsiPozitivni = fn sez =>
        veljaNaVseh (fn x => x > 0, sez)

val vsiLihi = fn sez =>
        veljaNaVseh (fn x => x mod 2 = 1, sez)

fun map (f, sez) =
        case sez of
                []         => []
                | h::t => f h::map (f, t)

fun filter (f, sez) =
        case sez of
                []         => []
                | h::t => if f h
                                  then h::filter (f, t)
                                  else filter (f, t)

fun fold (f, init, sez) =
        case sez of
                []         => init
                | h::t => fold (f, f(init, h), t)

val preslikaj = fn sez =>
        map (fn sez1 =>
                        fold (fn (x, sum) => x + sum, 0, sez1),
                sez);
```

```
fun obstaja (f, list) =
    case list of
        []        => false
        | x :: xs => f(x) orelse obstaja(f, xs)

fun zaVse (f, list) =
    case list of
        []        => true
        | x :: xs => f(x) andalso zaVse(f, xs)

fun clan (element, list) =
    obstaja (fn (x) => element = x, list)

fun vstavi (new, list) =
    if clan (new, list)
    then list
    else new :: list

fun jePodmnozica (list1, list2) =
    zaVse (fn el => clan (el, list1), list2)

fun jePodmnozica (list1, list2) =
    zaVse (fn x => obstaja (fn el => el = x, list1), list2)

fun staLoceni (list1, list2) =
    zaVse (fn x => not (clan (x, list2)), list1)

fun staLoceni (list1, list2) =
    zaVse (fn x => zaVse (fn y => x <> y, list2) , list1)

fun map (f, sez) =
    case sez of
        []       => []
        | h :: t => (f h) :: map (f, t)

fun filter (f, sez) =
    case sez of
        []       => []
        | h :: t => if f(h)
                    then h :: filter (f, t)
                    else filter (f, t)

fun presek (list1, list2) =
    filter (fn x => clan(x, list1), list2)

fun razlika (list1, list2) =
    filter (fn x => not (clan(x, list2)), list1)

fun zamenjaj (oldelement, newelement, list) =
    map (fn a => if a = oldelement then newelement else a, list)

fun kartprod(list1, list2) =
    map (fn x => map (fn y => (x,y), list1), list2)

fun unija (list1, list2) =
    list1 @ razlika (list2, list1)

fun odvod (list) =
    filter ( fn (a,b) => b > ~1 andalso a <> 0, map (fn (x, y) => ((x * y), (y - 1)), list))
```

```
fun obseg a b =
        if a > b then []
        else a::obseg (a+1) b

(* delna aplikacija funcije obseg *)
val stejNavzgor = obseg 1

val obrniNiz = implode o rev o explode

fun enDva sez =
        let fun has1 s1 =
                case s1 of
                        [] => true
                        | 1::t => has2 t
                        | _ => false
                and has2 s2 =
                        case s2 of
                                [] => true
                                | 2::t => has1 t
                                | _ => false
        in has1 sez end

fun stevec () =
        let
                val st = ref 0
                fun naslednji () = (st:= (!st + 1); !st)
                fun ponastavi () = (st:=0)
                val vrednost = !st
        in
                { naslednji = naslednji, ponastavi = ponastavi, vrednost = vrednost}
        end

(********************** M_STACK ********************)

signature M_STACK =
        sig
            type 'a mstack
            val new : 'a -> 'a mstack
            val push : 'a mstack * 'a -> unit
            val pop : 'a mstack -> 'a option
        end

structure mstack :> M_STACK =
        struct
                type 'a mstack = 'a list ref
                val new = fn x => ref [x]
                val push = fn (stack, x) => stack := x::(!stack)
                val pop = fn stack =>
                                        case !stack of
                                                [] => NONE
                                                | x::xs => (stack := xs; SOME x)
        end;

(* test:
        val s = mstack.new 2;
        mstack.push(s,2);
        mstack.pop(s)
*)

(************************ pcl *********************)
datatype 'a pcl = Pcl of 'a pcell ref
and 'a pcell = Nil | Cons of 'a * 'a pcl;

fun cons (h, t) =
        Pcl (ref (Cons (h,t)))

fun nill () = Pcl (ref Nil)

fun car (Pcl (ref (Cons (h,_)))) = h
fun cdr (Pcl (ref (Cons (_,t)))) = t

(*fun stl (Pcl (r as ref (Cons (h, t))), u) = (r := Cons (h, u));
```