ABSZTRAKCIÓ ADATOKKAL

Adatabsztrakció modulokkal: racionális számok (folyt.)

- A `Rat` struktúrában definiált értékekre teljes nevükkel kell hivatkozni:

```
Rat.printRat(Rat.mulRat(Rat.oneHalf, Rat.oneThird));
Rat.printRat(Rat.addRat(Rat.oneThird, Rat.oneThird));
```

- `open`-nel – a szignatúra által korlátozott mértékben – láthatóvá tehetjük a struktúra tartalmát:

```
open Rat;
equRat(addRat(oneThird, oneThird), twoThird);
addRat(oneThird, oneThird) = twoThird;
```

- A láthatóvá tétel lehet lokális (deklaráció, ill. kifejezés lokális deklarációval):

```
local open Rat
    val q1 = addRat(oneThird, oneThird); val q2 = twoThird
in  val ratPair = (q1, q2)
end;

let open Rat
in  printRat(addRat(oneThird, oneThird));
end;
```

Adatabsztrakció modulokkal: racionális számok (folyt.)

- Válasszunk a matematikában megszokotthoz közelebb álló neveket a függvényeknek:

```
signature Rat = sig
                    eqtype rat
                    val rat : int * int -> rat
                    val num : rat -> int
                    val den : rat -> int
                    val ++ : rat * rat -> rat
                    val -- : rat * rat -> rat
                    val ** : rat * rat -> rat
                    val // : rat * rat -> rat
                    val == : rat * rat -> bool
                    val toString : rat -> string
                    val one : rat
                    val oneHalf : rat
                    val oneThird : rat
                    val twoThird : rat
                    val zero : rat
                end;
```

Adatabsztrakció modulokkal: racionális számok (folyt.)

```
structure Rat :> Rat =
struct
  type rat = int * int;
  fun rat (n, d) =
        let val g = Gcd.gcd(n, d) in (n div g, d div g) : rat end
  fun num (q : rat) = #1 q
  fun den q = #2 (q : rat)
  fun op++(x, y) = rat(num x * den y + num y * den x, den x * den y)
  fun op--(x, y) = rat(num x * den y - num y * den x, den x * den y)
  fun op**(x, y) = rat(num x * num y, den x * den y)
  fun op//(x, y) = rat(num x * den y, den x * num y)
  fun op==(x, y) = num x * den y = den x * num y
  fun toString r = makestring(num r) ^ "/" ^ makestring(den r)
  val one      = rat(1,1)
  val zero     = rat(0,1)
  val oneHalf  = rat(1,2)
  val oneThird = rat(1,3)
  val twoThird = rat(2,3) end;
```

Adatabsztrakció modulokkal: racionális számok (folyt.)

- Az új műveleti jelek prefix pozícióban használhatók:

```
let open Rat
in
   print(toString(++( **(oneThird, oneHalf),oneThird)) ^ "\n");
   ++(oneThird, oneThird) = twoThird
end;
```

A `(` és a `**` közé legalább egy szóköz kell, különben az mosml *megjegyzés* kezdetének veszi!

- Vagy akár infix pozíciójúvá alakíthatók:

```
let open Rat
    infix 6 ++ --
    infix 7 ** //
in
   print(toString(oneThird ** oneHalf ++ oneThird) ^ "\n");
   oneThird ++ oneThird = twoThird
end;
```

Adatabsztrakció modulokkal: racionális számok (folyt.)

- A szokásos alapműveleti jeleket is újradefiniálhatjuk.
- Eredeti jelentésük nem vész el, de a műveletek teljes nevét kell használnunk *prefix* pozícióban:

```
load "Int";
let open Rat
    val op+ = ++
    val op- = --
    val op* = **
    val op/ = //
in
   print(toString oneHalf ^ "\n");
   print(toString(oneHalf + oneThird) ^ "\n");
   print(toString(oneHalf * oneThird) ^ "\n");
   print(toString(oneThird - oneThird) ^ "\n");
   print(toString(twoThird / oneThird) ^ "\n");
   oneThird + oneThird = twoThird;
   Int.+(1,2);
   1 Int.+ 2   (* hibás! *)
end;
```

Adatabsztrakció modulokkal: racionális számok (folyt.)

- *Új típust* és *konstruktorokat* hozhatunk létre a `datatype` deklarációval:

```
structure Rat :> Rat =
struct
  datatype rat = Rat of int * int
  fun rat (n, d) = let val g = Gcd.gcd(n, d) in Rat(n div g, d div g)
  fun num (Rat q) = #1 q
  fun den (Rat q) = #2 q
  fun op++(x, y) = rat(num x * den y + num y * den x, den x * den y)
  fun op--(x, y) = rat(num x * den y - num y * den x, den x * den y)
  fun op**(x, y) = rat(num x * num y, den x * den y)
  fun op//(x, y) = rat(num x * den y, den x * num y)
  fun op==(x, y) = num x * den y = den x * num y
  fun toString r = makestring(num r) ^ "/" ^ makestring(den r);
  val one      = rat(1,1)
  val zero     = rat(0,1)
  val oneHalf  = rat(1,2)
  val oneThird = rat(1,3)
  val twoThird = rat(2,3)  end;
```

Adatabsztrakció modulokkal: racionális számok (folyt.)

- Az adatkonstruktor mintaillesztésre *szelektorként* is felhasználható (és használni is kell):

```
structure Rat :> Rat =
struct
  datatype rat = Rat of int * int;
  fun rat (n, d) = let val g = Gcd.gcd(n, d) in Rat(n div g, d div g)
  fun num (Rat(n, _)) = n
  fun den (Rat(_, d)) = d
  fun op++(x, y) = rat(num x * den y + num y * den x, den x * den y)
  fun op--(x, y) = rat(num x * den y - num y * den x, den x * den y)
  fun op**(x, y) = rat(num x * num y, den x * den y)
  fun op//(x, y) = rat(num x * den y, den x * num y)
  fun op==(x, y) = num x * den y = den x * num y
  fun toString r = makestring(num r) ^ "/" ^ makestring(den r);
  val one      = rat(1,1)
  val zero     = rat(0,1)
  val oneHalf  = rat(1,2)
  val oneThird = rat(1,3)
  val twoThird = rat(2,3)  end;
```

Adatabsztrakció modulokkal: racionális számok (folyt.)

- Az adatkonstruktorfüggvény *valóban* használható új érték létrehozására:

```
structure Rat :> Rat =
struct
  datatype rat = Rat of int * int;
  val rat = Rat;
  fun num (Rat(n, _)) = n
  fun den (Rat(_, d)) = d
  fun op++(x, y) = rat(num x * den y + num y * den x, den x * den y)
  fun op--(x, y) = rat(num x * den y - num y * den x, den x * den y)
  fun op**(x, y) = rat(num x * num y, den x * den y)
  fun op//(x, y) = rat(num x * den y, den x * num y)
  fun op==(x, y) = num x * den y = den x * num y
  fun toString r = makestring(num r) ^ "/" ^ makestring(den r);
  val one      = rat(1,1)
  val zero     = rat(0,1)
  val oneHalf  = rat(1,2)
  val oneThird = rat(1,3)
  val twoThird = rat(2,3)  end;
```

GYENGE ÉS ERŐS ABSZTRAKCIÓ

Összefoglalás: gyenge és erős adatabsztrakció

- Gyenge absztrakció: a név szinonima, az adatszerkezet részei továbbra is hozzáférhetők.
- Erős absztrakció: a név új dolgot (entitást, objektumot) jelöl, az adatszerkezet részeihez csak korlátok között lehet hozzáférni.
- `type`: gyenge absztrakció; pl. `type rat = {num : int, den : int}`
  - Új nevet ad egy típuskifejezésnek (vö. értékdeklaráció).
  - Segíti a programszöveg megértését.
- `abstype`: erős absztrakció
  - Új típust hoz létre: név, műveletek, ábrázolás, jelölés.
  - Túlhaladott, van helyette jobb: `datatype` + modulok
- `datatype`: modulok nélkül gyenge, modulokkal erős absztrakció;
  pl. `datatype 'a esetleg = Semmi | Valami of 'a`
  Belső változata az SML-ben: `datatype 'a option = NONE | SOME of 'a`
  - Új entitást hoz létre.
  - Rekurzív és polimorf is lehet.

Összefoglalás: gyenge és erős adatabsztrakció (folyt.)

- `datatype logi = Igen | Nem` Felsorolásos típus.
  `datatype logi3 = igen | nem | talan` Felsorolásos típus.
  `datatype 'a esetleg = Semmi | 'a Valami` Polimorf típus.
- *Adatkonstruktornak* nevezzük a létrehozott `Igen`, `Nem`, `igen`, `nem`, `talan`, `Semmi` és `Valami` értékeket. `Valami` ún. *adatkonstruktorfügvény,* az összes többi ún. *adatkonstruktorállandó.* Az adatkonstruktorok a többi értéknévvel azonos névtérben vannak.
- *Típuskonstruktornak* nevezzük a létrehozott `logi`, `logi3` és `esetleg` neveket; `esetleg` ún. postfix *típuskonstruktorfüggvény* (vagy típusoperátor), a másik kettő ún. *típuskonstruktorállandó* (röviden típusállandó). Típusnévként használható a típusállandó (pl. `logi`), valamint a típusállandóra vagy típusváltozóra alkalmazott típuskontruktorfüggvény (pl. `int list` vagy `'a esetleg`). A típuskonstruktorok más névtérben vannak, mint az értéknevek.
- Természetesen az adatkonstruktoroknak is van típusuk, pl.

```
Igen   : logi     | Semmi  : 'a esetleg
Nem    : logi     | Valami : 'a -> 'a esetleg
```

- Példa `datatype` deklarációval létrehozott adattípust kezelő függvényre

```
fun inverz Nem = Igen | Igen = Nem
```

Deklaráció lokális érvényű deklarációval: `local`-deklaráció

- Ún. `local`-deklarációt használunk, ha egyes deklarációkat fel akarunk használni más deklarációkban, miközben *el akarjuk rejteni* őket a program többi része elől.
- Szintaxisa:
  ```
  local d1
  in d2
  end
  ```
  ahol *d1* egy nemüres deklarációsorozat,
  d2 egy másik nemüres deklarációsorozat.
- Példa:

```
(* length : 'a list -> int
   length zs = a zs lista hossza
*)
local
      (* len : 'a list * int -> int
         len (zs, n) = az n és a zs lista hosszának összege
      *)
      fun len ([], n)     = n
        | len (_::zs, n) = len(zs, n+1)
in
   fun length zs = len(zs, 0)
end
```

OPCIONÁLIS ÉRTÉK

Opcionális érték kezelése (`'a option`)

```
datatype 'a option = NONE | SOME of 'a
```

Függvények az `Option` könyvtárból:

```
val getOpt      : 'a option * 'a -> 'a
val isSome      : 'a option -> bool
val valOf       : 'a option -> 'a
val filter      : ('a -> bool) -> 'a -> 'a option
val map         : ('a -> 'b) -> 'a option -> 'b option
val mapPartial  : ('a -> 'b option) -> ('a option -> 'b option)
```

getOpt (xopt, d) = x if xopt is SOME x, d otherwise.

isSome xopt = true if xopt is SOME x, false otherwise.

valOf xopt = x if xopt is SOME x, raises Option otherwise.

filter p x = SOME x if p x is true, NONE otherwise.

map f xopt = SOME(f x) if xopt is SOME x, NONE otherwise.

mapPartial f xopt = f x if xopt is SOME x, NONE otherwise.

Példák opcionális értékek kezelésére

- Egészlista legnagyobb elemének kiválasztása

  Üres listának nincs legnagyobb eleme; egyelemű lista egyetlen eleme a „legnagyobb”; legalább kételemű lista legnagyobb eleme az első elem és a maradéklista elemei közül a legnagyobb.

```
(* maxl : int list -> int option
   maxl ns = az ns egészlista legnagyobb eleme *)
fun maxl []       = NONE    (* üres *)
  | maxl [n]      = SOME n  (* egyelemű *)
  | maxl (n::ns) =          (* legalább kételemű *)
              SOME(Int.max(n, valOf(maxl ns)))
```

- Füzér elején álló karaktersorozat átalakítása egész számmá

```
val Int.fromString : string -> int option (* Overflow *)

Int.fromString s = SOME i if a decimal integer numeral can be scanned
          from a prefix of string s, ignoring any initial whitespace;
          NONE otherwise.  A decimal integer numeral, after any initial
          whitespace, must have the form: [+~-]?[0-9]+

Int.fromString "1234"; Int.fromString "-1234"; Int.fromString "~1234";
Int.fromString "+1234"; Int.fromString "+007"; Int.fromString "alma"
```

KIVÉTELKEZELÉS

Kivételkezelés

- Kivételt az `exception` kulcsszóval deklarálunk, a `raise` kulcsszóval jelzünk, a `handle` kulcsszóval bevezetett kifejezésben kezelünk.
- A kivételt általában hibák jelzésére használjuk, de használhatjuk visszalépés kezelésére is (az utóbbira példa a `valtas` függvényben látható a következő fóliák egyikén).
- A kivételdeklaráció az adattípus-deklarációra (`datatype`-deklarációra) emlékeztet: `exception` *name*`;` `exception` *name* `of` *ty*.
- Példák kivétel deklarálására: `exception Valt; exception Hiba of char * int`.
- A kivételkonstruktor állandó vagy függvény lehet. Példák: `Valt : exn`, `Hiba : char * int -> exn`.
- A kivételdeklaráció speciális adattípus-deklaráció, ui. az utóbbival ellentétben dinamikusan *bővíti* a kivételkonstruktorok halmazát.
- Kivétel jelzésére a `raise` kulcsszóval kezdődő speciális kifejezést kell használunk.
- Példák kivétel jelzésére: `raise Valt`, `raise Hiba(#"N", 4)`.
- `raise` (hipotetikus) típusa: `exn -> 'a`.

Kivételkezelés (folyt.)

- `raise` alkalmazásának eredménye az ún. *kivételcsomag*. Mivel a kivételcsomag polimorf típusú, bármely más típussal kompatíbilis.
- A kivétel kezelése a `case`-szerkezetre emlékeztet: `E handle P1 => E1 | ··· | Pn => En`
- Ha `E` „közönséges” értéket ad eredményül, a kivételkezelő egyszerűen továbbadja az eredményt.
- Ha `E` eredménye *kivételcsomag*, az SML megpróbálja illeszteni a `P1, ···, Pn` mintákra.
  - Ha `Pi` (`1 <= i <= n`) az első illeszkedő minta, akkor `Ei` a kivételkezelő eredménye.
  - Ha egyetlen minta sem illeszkedik a kivételcsomagra, a kivételkezelő továbbpasszolja.
- Példák kivétel kezelésére:
  - ```
    erme :: valtas (erme::ermelista) (osszeg-erme)
                handle Valt => valtas ermelista osszeg
    ```
  - `(fn i => kivKez i handle Hiba(c, i) => (print(str c); i-1)) 0`
- `handle` (hipotetikus) típusa: `exn -> 'a`.
- Legyen `Ex` `exn` típusú kivétel, `e` pedig tetszőleges kifejezés; ekkor az `e handle Ex => c` (kivételkezelőt tartalmazó) kifejezésben `c`-nek `e`-vel azonos típusúnak kell lennie.
    ```

Kivételkezelés (folyt.)

- A következő programrészlet példa kivétel deklarálására, jelzésére és kezelésére

```
exception Hiba of char * int;

fun kivKez  0 = raise Hiba(#"N", 4)
  | kivKez ~9 = raise Hiba(#"M", 9)
  | kivKez  n = n;

fun kivKezel i =
        kivKez i handle Hiba(#"N", i) => (print "N"; i)
                      | Hiba(#"M", i) => (print "M"; i-1);

kivKezel 0  = 4;
kivKezel ~9 = 8;
kivKezel 7  = 7;
```

Kivételkezelés (folyt.)

- Példa visszalépés programozására kivételkezeléssel

```
exception Valt;

(* valtas : int list -> int -> int list
   valtas ermelista osszeg = a lehető legkevesebb érmét tartalmazó olyan
                             érmelista, amely elemeinek összege osszeg
   PRE : ermelista = a váltásra használható érmék csökkenő értéksorrendben
         osszeg >= 0
*)
fun valtas _ 0  = []
  | valtas [] _ = raise Valt
  | valtas (erme::ermelista) osszeg =
      if (* ha az adott érme túl nagy, a következővel próbálkozunk *)
         erme > osszeg then valtas ermelista osszeg
         (* ha az adott érmétől kezdve sikerül felváltani, az jó;
            ha nem, a következő érmével kezdjük újra az adott ponttól *)
      else erme :: valtas (erme::ermelista) (osszeg-erme)
                                  handle Valt => valtas ermelista osszeg;

valtas [50, 20, 10, 5, 2] 197 = [50, 50, 50, 20, 20, 5, 2];
```

Kivételkezelés (folyt.)

- A leggyakoribb belső kivételek

Név	*Művelet, amely a kivételt kiválthatja*
`Bind`	Értékdeklarációban a jobb oldali kifejezés nem illeszkedik a bal oldali mintára.
`Chr`	`chr pred succ`
`Div`	`/ div mod`
`Domain`	Az érték kilóg az értelmezési tartományból.
`Empty`	`hd tl last`
`Fail`	`compile load loadOne  Fail : string -> exn`
`Interrupt`	Megszakítás ctrl/c-vel.
`Io`	Ki/beviteli hiba. `Io : {cause : exn, function : string, name : string}`
`Match`	Mintaillesztési hiba `case` és `handle` kifejezésben, vagy függvényalkalmazásban.
`Option`	Hiba egy `Option` könyvtárbeli függvény alkalmazásakor.
`Overflow`	`~ + - * / div mod abs ceil floor round trunc`
`Size`	`^ array concat fromList implode tabulate translate vector`
`Subscript`	`copy drop extract nth sub substring take update`

- `Fail` és `Io` kivételkonstruktorfüggvények, a többi `exn` típusú kivételkonstruktorállandó.
- `Option` csak `Option.Option` néven használható, ha nem nyitjuk meg az `Option` könyvtárat.

TÖBB MEGOLDÁS ELŐÁLLÍTÁSA VISSZALÉPÉSSEL

n vezér a sakktáblán

- Hányféleképpen rakható *n* vezér a sakktáblára úgy, hogy ne üssék egymást?
  - A vezéreket tartalmazó mezők sorának `j` sorszámát az egyes oszlopokon belül egy `n` hosszú sorvektor adott oszlophoz rendelt mezőjébe írt szám adja meg, ahol `j<=s< n`. Példa `n=4` esetén:

```
   +---+---+---+---+
   | 3 | 1 | 4 | 2 |
   +---+---+---+---+


     0   ---->  n-1
   +---+---+---+---+
 0 |   | q |   |   |
   +---+---+---+---+
 | |   |   |   | q |
 | +---+---+---+---+
 V | q |   |   |   |
   +---+---+---+---+
n-1|   |   | q |   |
   +---+---+---+---+
```

  - A sorvektort (egy egyre bővülő) listával valósítjuk meg. Egy listához balról könnyű új elemeket fűzni, ezért a táblát és a vezérek helyzetét leíró listát hossztengelye mentén tükrözzük.

```
   ...-+---+---+---+
       | 4 | 1 | 3 |
   ...-+---+---+---+

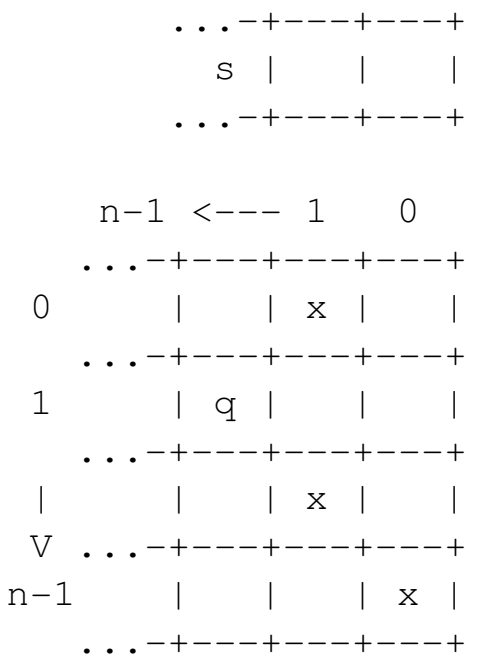

    n-1  <----   0
   ...-+---+---+---+
 0     |   | q |   |
   ...-+---+---+---+
 |     |   |   |   |
 | ...-+---+---+---+
 V     |   |   | q |
   ...-+---+---+---+
n-1    | q |   |   |
   ...-+---+---+---+
```

n vezér a sakktáblán (folyt.)

Azt, hogy az új vezért üti-e egy korábban a táblára rakott másik vezér, a sorvektor vizsgálatával dönthetjük el: a sorvektor azt adja meg, hogy a listaelem indexe által meghatározott oszlopban és a listaelem értéke által meghatározott sorban vezér van.

1. Az új vezér sorának sorszáma, azaz az új listalem értéke nem fordulhat elő a lista már felépített részében.
2. Az új vezér átlós irányban sem lehet egy vonalban más vezérrel a táblán. Ez azt jelenti, hogy ha a sorvektort jelentő lista elejére az `s` sorindexet akarjuk rakni, akkor az `i`-edik elemének az értéke, ha van ilyen eleme, nem lehet `s-(i+1)`, ill. `s+(i+1)`.
3. A következő példa segít megvilágítani az esetet.

Ha a `2`-es oszlopba és az `s=1`-es sorba akarjuk lerakni az új vezért, akkor az `x`-szel jelölt mezőket kell megvizsgálnunk. Az eddig létrehozott listának (sorvektornak) két eleme van, ahol a lista fejének az indexe 0. A listafej értéke nem lehet s-1, sem s+1. A lista rekurzív algoritmussal dolgozható fel.



n vezér a sakktáblán: „ütésben van”-vizsgálat

```
(* utesbenVan : int list -> bool
   utesbenVan zs = igaz, ha a (hd zs) vezért legalább egy
                   (tl zs)-beli vezér üti
*)
fun utesbenVan [] = false
  | utesbenVan (z::zs) =
      let (* uV : int -> int -> int list -> bool
             uV s1 s2 rs = igaz, ha a z vezért s1, s2 vagy r, vagy
                    egy másik rs-beli vezér közül legalább egy üti
          *)
          fun uV _ _ [] = false
            | uV s1 s2 (r::rs) = z = r orelse
                                 s1 = r orelse
                                 s2 = r orelse
                                 uV (s1-1) (s2+1) rs
      in
          uV (z-1) (z+1) zs
      end
```

n vezér a sakktáblán: egy megoldás előállítása

```
exception Zsakutca

(* vezerek0 : int -> int list
   vezerek0 n = a feladvány egy megoldása n vezér esetén
*)
fun vezerek0 n =
  let (* vez: int -> int list -> int list
         vez z zs: egy megoldás n vezér esetén
      *)
      fun vez z zs =
            if z = 0 andalso utesbenVan zs orelse z = n
            then raise Zsakutca
            else if length zs = n
            then rev zs
            else vez 0 (z::zs) handle Zsakutca => vez (z+1) zs
  in
      vez 0 []
  end
```

n vezér a sakktáblán: több megoldás előállítása visszalépéssel

```
(* vezerek1 : int -> int list list
   vezerek1 n = a feladvány összes megoldásának listája
                n vezér esetén
*)
fun vezerek1 n =
  let (* vez: int -> int list -> int list list
         vez z zs: az összes megoldás listája n vezér esetén
      *)
      fun vez z zs =
            if z = 0 andalso utesbenVan zs orelse z = n
            then raise Zsakutca
            else if length zs = n
            then [rev zs]
            else (vez 0 (z::zs) handle Zsakutca => []) @
                 (vez (z+1) zs handle Zsakutca => [])
  in
      vez 0 []
  end
```

n vezér a sakktáblán: több megoldás előállítása listák listájával

```
(* vezerek2 : int -> int list list
   vezerek2 n = a feladvány összes megoldásának listája
                n vezér esetén
*)
fun vezerek2 n =
  let (* vez: int -> int list -> int list list
         vez z zs: az összes megoldás listája n vezér esetén
      *)
      fun vez z zs =
            if z = 0 andalso utesbenVan zs orelse z = n
            then []
            else if length zs = n
            then [rev zs]
            else vez 0 (z::zs) @ vez (z+1) zs
  in
      vez 0 []
  end
```

n vezér a sakktáblán: több megoldás előállítása listák listájával (folyt.)

Akkumulátor alkalmazásával:

```
(* vezerek3 : int -> int list list
   vezerek3 n = a feladvány összes megoldásának listája
                n vezér esetén
*)
fun vezerek3 n =
  let (* vez: int -> int list -> int list list -> int list list
         vez z zs: az összes megoldás listája n vezér esetén
      *)
      fun vez z zs zss =
            if z = 0 andalso utesbenVan zs orelse z = n
            then zss
            else if length zs = n
            then rev zs :: zss
            else vez 0 (z::zs) (vez (z+1) zs zss)
  in
     vez 0 [] []
  end
```