FUNKCIONÁLIS PROGRAMOZÁS, II. RÉSZ

Az I. rész – az első hét FP-előadás – főbb témái

- A funkcionális programozás alapjai, λ-kalkulus
- Típus, típusváltozó, egyenlőségi típus, típuskifejezés, egyszerű és összetett típus (az SML *erősen típusos* nyelv)
- Név, érték, változó (nem frissíthető!), kifejezés (az SML-ben minden érték *teljes jogú érték*)
- Függvény (egyetlen paraméter!), részlegesen alkalmazható és magasabbrendű függvény; névtelen függvény (`fn`-jelölés vagy λ-kifejezés), mintaillesztés
- Egyszerű típus (`int`, `real`, `char`, `bool`, `word`, `string`, `unit`)
- Összetett típus: lista, rekord, ennes, nullas, `datatype`-deklaráció
- Listakezelő függvények (`hd`, `length`, `map`, `filter`, `maxl`, `@`, `rev`, `revAppend`, `take`, `drop`, `foldr`, `foldl` stb.)
- Lokális érvényű deklaráció: `let`-kifejezés, `local`-deklaráció
- Könyvtárak (`Char`, `Int`, `Real`, `String`, `List`, `Bool`, `Option`, `Time`, `Timer` stb.)
- Logikai műveletek (`andalso`, `orelse`, `not`), `if-then-else` kifejezés); lusta kiértékelés
- Polimorfizmus: paraméteres és többszörös terheléses
- Listák használata, futási idő mérése

Az I. rész – az első hét FP-előadás – főbb témái (folyt.)

- Kifejezések kiértékelése, mohó és lusta kiértékelés
- Jobbrekurzió, akkumulátor (gyűjtőargumentum)
- Példák `datatype`-deklarációra: felsorolásos típus, megkülönböztetett egyesítés, rekurzív típus; polimorf adattípus
- `datatype`-deklaráció és mintaillesztés; típuskonstruktorállandó és típuskonstruktorfüggény (típusoperátor), adatkonstruktorállandó és adatkonstruktorfüggény (operátor)
- Esetszétválasztás: `case`
- Opcionális értékek kezelése: `Option`, `getOpt`, `isSome`, `valOf` stb.
- Bináris és nem bináris fák és más gráfok `datatype`-deklarációval
- Egyszerű műveletek bináris fákon: `depth`, `nodes`, `fulltree`, `reflect` stb.
- Bináris fa bejárása (lista létrehozása bináris fa elemeiből): `preorder`, `inorder`, `postorder`
- Bináris fa létrehozása lista elemeiből: `balPreorder`, `balInorder`, `balPostorder`
- Elem törlése bináris fából: `join` és `remove`
- Bináris keresőfák: `blookup`, `binsert`, `bupdate`

KIÍRÁS

Kiírás

- `{TextIO.}print : string -> unit`

  `print s` = kiírja az `s` értékét a standard kimenetre, és azonnal kiüríti a puffert.

- `{General.}makestring : numtxt-> string`

  `makestring v` = eredménye a `v` érték *ábrázolása.*

- `{Meta.}printVal : 'a -> 'a`

  `printVal e` = kiírja az `e` kifejezés értékét a standard kimenetre pontosan úgy, ahogyan az SML értelmező írja ki a „legfelső szinten”, és azonnal kiüríti a puffert. Eredményül visszaadja az `e` kifejezés értékét. *Csak interaktív módban használható.*

1. megjegyzés. A `{` és `}` kapcsos zárójelek között opcionális modulnév áll. Pl. `{TextIO.}print` azt jelenti, hogy a függvény a `TextIO` modulban van definiálva, de az SML-értelmező a `print` nevet rövid alakban is felismeri.

2. megjegyzés. *numtxt* = `int` | `real` | `word` | `word8` | `char` | `string`

- Különböző típusú egyszerű értékeket alakítanak át füzérré a `toString` függvények:

```
Bool.toString   : bool -> string      String.toString  : string -> string
Char.toString   : char -> string      Time.toString    : time -> string
Date.toString   : date -> string      Word.toString    : word -> string
Int.toString    : int -> string       Word8.toString   : word8 -> string
Real.toString   : real -> string
```

Kiírás (folyt.)

- Példák:

```
- print("alma"^"Korte\n");
almaKorte
> val it = () : unit

- makestring ~5.8e~3;
> val it = "~0.0058" : string

- printVal("alma"^"Korte\n");
"almaKorte\n"> val it = "almaKorte\n" : string

- makestring("alma"^"Korte\n");
> val it = "\"almaKorte\\n\"" : string
```

- `printVal`-lal tetszőleges típusú érték íratható ki, például ennes, rekord vagy lista:

```
- printVal (3, 5.0);
(3, 5.0)> val it = (3, 5.0) : int * real

- printVal {m2 = 3, m1 = 5.0};
{m1 = 5.0, m2 = 3}> val it = {m1 = 5.0, m2 = 3} : {m1 : real, m2 : int}

- printVal [#"A",#"Z",#":"];
[#"A", #"Z", #":"]> val it = [#"A", #"Z", #":"] : char list
```

Kiírás (folyt.)

- `printVal`-lal `datatype`-deklarációval létrehozott típusú érték is kiíratható ki, pl.

```
- datatype t = L | B of t * t;
> New type names: =t
  datatype t = (t,con B : t * t -> t, con L : t)
  con B = fn : t * t -> t
  con L = L : t
- val fa = B(B(B(L,B(L,B(L,B(B(L,L),L)))),L),B(L,L));
> val fa = B(B(B(L, B(L, B(L, B(B(L, L), L)))), L), B(L, L)) : t
- printVal fa;
B(B(B(L, B(L, B(L, B(B(L, L), L)))), L), B(L, L))> val it = B(...
```

- A kiírt sor túl hosszú, a folytatását . . .-tal helyettesítettük.
- Törjük el a sort a > *válaszjel* előtt *szekvenciális* kifejezés alkalmazásával:

```
- (printVal fa; print "\n");
B(B(B(L, B(L, B(L, B(B(L, L), L)))), L), B(L, L))
> val it = () : unit
```

- Sajnos, az eredményül kapott érték nem `fa` értéke!

Kiírás (folyt.)

- Hogyan írjunk ki egy újsor-jelet úgy, hogy az eredmény mégis `fa` értéke legyen? Pl. így:

```
- let val res = printVal fa; val _ = print "\n"
  in
     res
  end;
B(B(B(L, B(L, B(L, B(B(L, L), L)))), L), B(L, L))
> val it = B(B(B(L, B(L, B(L, B(B(L, L), L)))), L), B(L, L)) : t
```

- Ez elég körülményes. A `before` operátort az ilyen esetek kezelésére vezették be:

```
- printVal fa before print "\n";
B(B(B(L, B(L, B(L, B(B(L, L), L)))), L), B(L, L))
> val it = B(B(B(L, B(L, B(L, B(B(L, L), L)))), L), B(L, L)) : t
```

- Szekvenciális kifejezés alkalmazásával további magyarázó szöveget írhatunk ki:

```
- (print "A fa változó értéke =\n"; printVal fa before print "\n");
A fa változó értéke =
B(B(B(L, B(L, B(L, B(B(L, L), L)))), L), B(L, L))
> val it = B(B(B(L, B(L, B(L, B(B(L, L), L)))), L), B(L, L)) : t
```

Kiírás (folyt.)

- Hosszú lista, ill. egymásba skatulyázott adatszerkezetek esetén `printVal` (és maga az SML-értelmező is) alapesetben csak az első 200 listaelemet, ill. legfeljebb 20 szintet ír ki. A hosszat a `printLength`, a szintek számát a `printDepth` *frissíthető változó* szabályozza. Mindkét érték felülírható.

```
printLength : int ref       printLength := 7;  !printLength;
printDepth  : int ref       printDepth  := 3;  !printDepth;
```

- Példák:

```
- printLength := 6; printVal [1,2,3,4,5,6,7,8,9,10] before print "\
[1, 2, 3, 4, 5, 6, 7, ...]
> val it = [1, 2, 3, 4, 5, 6, ...] : int list
- printDepth := 4; printVal fa before print "\n";
B(B(#, #), B(#, #))
> val it = B(B(#, #), B(#, #)) : t
```

- Figyelem: a `printLength` és a `!printLength` kifejezések különböznek!

```
- printLength;                 | - !printLength;
> val it = ref 7 : int ref     | > val it = 7 : int
```

NYOMKÖVETÉS KIÍRÁSSAL: LISTÁK

Nyomkövetés kiírással: `length` (nem iteratív)

- Az MOSML-ben nyomkövetés csak a program szövegébe beírt kiíró függvényekkel lehetséges.
- Példa: a `length` függvény két változatának kiértékelése
- A `length` „naív” változata

```
fun length (_::xs) = 1 + length xs
  | length []      = 0
```

- A `length` „naív” változata kiíró függvényekkel (**félkövér** szedéssel az eredeti szöveg látható)

```
fun length ((_ : int) :: xs) =
        printVal(1 + (print " & "; printVal(length(printVal xs))
                                     before print " $ "
                     )
                 )
        before print " #\n"
  | length []      = (print " * "; printVal 0
                                     before print " %\n")
```

Nyomkövetés kiírással: `lengthi` (iteratív)

- A `length` iteratív változata

```
fun lengthi xs = let fun len (i, _::xs) = len(i+1, xs)
                       | len (i, [])    = i
                 in len(0, xs)
                 end
```

- A `length` iteratív változata kiíró függvényekkel (**félkövér** szedéssel az eredeti szöveg látható)

```
fun lengthi xs =
    let fun len (i, (_ : int) :: xs) =
            len((print " "; printVal((printVal i
                                      before print " $ ") + 1)),
                (print " & "; printVal xs)
               )
                before print "#\n"
          | len (i, []) = (print " * "; printVal i
                                        before print " %\n")
    in len(0, xs)
    end
```

Nyomkövetés kiírással: `length` egy alkalmazása

- `length` egy alkalmazása

```
fun length ((_ : int) :: xs) =
        printVal(1 + (print " & "; printVal(length(printVal xs))
                                    before print " $ "
                     )
                 )
        before print " #\n"
  | length []       = (print " * "; printVal 0
                                    before print " %\n")

length [1,2,3];
 & [2, 3] & [3] & [] * 0 %
0 $ 1 #
1 $ 2 #
2 $ 3 #
```

Nyomkövetés kiírással: `lengthi` egy alkalmazása

- `lengthi` egy alkalmazása

```
fun lengthi xs =
    let fun len (i, (_ : int) :: xs) =
            len((print " "; printVal((printVal i
                                      before print " $ ") + 1)),
                (print " & "; printVal xs)
               )
                before print "#\n"
          | len (i, []) = (print " * "; printVal i
                                        before print " %\n")
    in len(0, xs)
    end

lengthi [1,2,3];
 0 $ 1 & [2, 3] 1 $ 2 & [3] 2 $ 3 & [] * 3 %
#
#
#
```

Nyomkövetés kiírással: `length` és `lengthi` összehasonlítása

- `length` és `lengthi` kiértékelésének összehasonlítása

```
length [1,2,3];

 & [2, 3] & [3] & [] * 0 %
0 $ 1 #
1 $ 2 #
2 $ 3 #
```

```
lengthi [1,2,3];

 0 $ 1 & [2, 3] 1 $ 2 & [3] 2 $ 3 & [] * 3 %
#
#
#
```

Hibakeresés Poly/ML-ben

- Példafüggvény a Poly/ML debugger használatához

```
(* length : 'a list -> int
   length zs = a zs elemeinek száma *)
fun length zs =
      let (* len : 'a list -> int
             len xs = az xs elemeinek száma *)
          fun len []         = 0
            | len (_ :: xs) = 1 + len xs
      in
          len zs
      end;
```

- Töréspontot elhelyezni egy olyan segédfüggvényben lehet, amelyet egy másik függvény törzsében egy `let`-kifejezésben definálunk.
- A fontosabb hibakereső függvényeket lásd a következő fólián (konkretizálva a `length` és a `len` függvényre utaló hivatkozásokkal).
- A `PolyML.profiling : int -> unit` függvénnyel egy kifejezés kiértékelési idejét, ill. egyes függvények futási idejét és helyfoglalását monitorozhatjuk (részletek a PolyML-leírásban).

Hibakeresés Poly/ML-ben (folyt.)

`PolyML.Compiler.debug := true;`	Hibakeresés engedélyezése; engedélyezett
`open PolyML.Debug;`	állapotban kell definiálni a vizsgálandó függvényt
`breakIn "len";`	Töréspont elhelyezése, rövid változat
`breakIn "length()len";`	Töréspont elhelyezése, teljes változat
`continue();`	Folytatás a töréspont következő előfordulásáig
`down();`	Áttérés az előző hívási szintre a veremben
`up();`	Áttérés a következő hívási szintre a veremben
`dump();`	A verem teljes tartalmának kiírása
`stack();`	A hívások kiírása a veremtartalom alapján
`variables();`	A változók értékének kiírása
`clearIn "length()len";`	Töréspont törlése, teljes változat
`trace true;`	Nyomkövetés bekapcsolása
`step();`	Adott hívási szinten tovább vagy beljebb
`stepOver();`	Adott hívási szinten tovább
`stepOut();`	Előző hívási szintre vissza

Részletesen ld. `<http://dp.iit.bme.hu/polyml/docs/Debugging.html>`. A példa megtalálható a `<http://dp.iit.bme.hu/sml/eloadas_anyagok/dp03s-fp08-10.sml>` címről letölthető fájlban.

Hibakeresés Poly/ML-ben (folyt.)

- A listaelemek értékének megjelenítéséhez *típusmegkötéssel* meg kell adni a típusukat.

```
(* length : real list -> int
   length zs = a zs elemeinek száma *)
fun length (zs : real list) =
      let (* len : real list -> int
             len xs = az xs elemeinek száma *)
          fun len []         = 0
            | len (_ :: xs : real list) = 1 + len xs
      in len zs
      end;
```

- `length` egy hibás változata a hibakeresés kipróbálásához

```
(* length : 'a list -> int
   length zs = a zs elemeinek száma *)
fun length zs =
      let (* len : 'a list * int -> int
             len xs = n + az xs elemeinek száma -- HIBÁS! *)
          fun len ([], n)       = n
            | len (_ :: xs, n) = len(xs, n)
      in len(zs,0)
      end;
```

Hibakeresés Poly/ML-ben (folyt.)

- `length` egy másik hibás változata a hibakeresés kipróbálásához

```
(* length : real list -> int
   length zs = a zs elemeinek száma *)
fun length (zs : real list) =
      let (* len : real list * int -> int
             len xs = n + az xs elemeinek száma -- HIBÁS! *)
          fun len ([], n)      = n
            | len (_ :: xs : real list, n : int) = len(xs, n)
      in len(zs,0)
      end;
```

- `length` egy jó változata a hibakeresés, ill. a nyomkövetés kipróbálásához

```
(* length : real list -> int
   length zs = a zs elemeinek száma *)
fun length (zs : real list) =
      let (* len : real list * int -> int
             len xs = n + az xs elemeinek száma *)
          fun len ([], n)      = n
            | len (_ :: xs : real list, n : int) = len(xs, n+1)
      in len(zs,0)
      end;
```

NYOMKÖVETÉS KIÍRÁSSAL: BINÁRIS FÁK

Nyomkövetés kiírással: bináris fák

- Korábban tárgyaltuk a `nodes` és `depth` függvényeket, valamint akkumulátort használó `nodesa` és `deptha` változatukat.
- A következő fóliákon e függvények kiértékelésének nyomkövetésére mutatunk megoldást.
- A szöveg olvashatóságát (szintenként növekvő) *behúzással* javítjuk. A megfelelő számú szóköz beszúrására szolgál a `tab` függvény. A változó számú szóközből álló füzért paraméterként adjuk át, ezért olyan segédfüggvényeket vezetünk be, amelyeknek az őket meghívó függvényhez képest eggyel több paraméterük van.
- A függvények *kiírást szolgáló részek nélküli* szövegét **félkövér** szedéssel jelöljük.

Nyomkövetés kiírással: nodes (akkumulátort nem használ)

```
(* tab : string -> string
   tab i = a sorok behúzásához használandó i füzér szóközökkel kiegészítve *)
fun tab i = i ^ "    "

fun nodes f =
      let (* nodes0 : string -> 'a tree -> int
             nodes0 i f = a csomópontok száma az f fában;
                          i a behúzáshoz használt füzér *)
          fun nodes0 i (N(a, t1, t2)) =
                (print("\n" ^ i ^ "<"); printVal a : int; print "> ";
                 printVal(1 +
                          nodes0 (tab i) (printVal t2 before print " *") +
                          nodes0 (tab i) (printVal t1 before print " %")
                          before print "$ "
                         )
                 before print(" #\n" ^ i)
                )
            | nodes0 i L = (print("\n" ^ i); 0)
      in
          nodes0 "" f
      end
```

Nyomkövetés kiírással: nodesa (akkumulátort használ)

```
fun nodesa f =
  let (* nodes0 i (f, n) = n + a csomópontok száma f-ben;
                           i a behúzáshoz használt füzér
         nodes0 : string -> 'a tree * int -> int
       *)
      fun nodes0 i (N(a, t1, t2), n) =
            (print("\n" ^ i ^ "<"); printVal a : int; print "> ";
             nodes0 (tab i) (printVal t1 before print(" %\n" ^ (tab i)),
                             nodes0 (tab i) (printVal t2 before print(" *\n" ^
                                                                      (tab i)),
                                             printVal(n+1) before print " $"
                                            )
                            )
             before print(" #" ^ i)
            )
        | nodes0 i (L, n) = (* (print("\n" ^ i); n) *) n
  in
      nodes0 "" (f, 0)
  end
```

- A következő fóliákon nodes-t és nodesa-t hét csomópontból álló teljes fára alkalmazzuk.

Nyomkövetés kiírással: `nodes` és `nodesa` egy alkalmazása

```
f7 = N(1, N(2, N(4, L, L), N(5, L, L)), N(3, N(6, L, L), N(7, L, L))) : int tree
```

```
- nodes f7;

<1> N(3, N(6, L, L), N(7, L, L)) *
    <3> N(7, L, L) *
        <7> L *
            L %
            $ 1 #
        N(6, L, L) %
        <6> L *
            L %
            $ 1 #
        $ 3 #
    N(2, N(4, L, L), N(5, L, L)) %
```

```
- nodesa f7;

<1> N(2, N(4, L, L), N(5, L, L)) %
    N(3, N(6, L, L), N(7, L, L)) *
    1 $
    <3> N(6, L, L) %
        N(7, L, L) *
        2 $
        <7> L %
            L *
            3 $ #
        <6> L %
            L *
            4 $ #          #
```

Folytatása a következő lapon.

Nyomkövetés kiírással: `nodes` és `nodesa` egy alkalmazása (folyt.)

```
f7 = N(1, N(2, N(4, L, L), N(5, L, L)), N(3, N(6, L, L), N(7, L, L))) : int tree
```

```
 (nodes f7)

    <2> N(5, L, L) *
        <5> L *
            L %
            $ 1 #
        N(4, L, L) %
        <4> L *
            L %
            $ 1 #
        $ 3 #
    $ 7 #

> val it = 7 : int
```

```
 (nodesa f7)

    <2> N(4, L, L) %
        N(5, L, L) *
        5 $
        <5> L %
            L *
            6 $ #
        <4> L %
            L *
            7 $ #          #     #

> val it = 7 : int
```

Nyomkövetés kiírással: `depth` (akkumulátort nem használ)

```
fun depth f =
  let (* depth0 i f = az f fa mélysége; i a behúzáshoz használt füzér
         depth0 : string -> 'a tree -> int
      *)
      fun depth0 i (N(a : int, t1, t2)) =
            (print("\n" ^ i ^ "<"); printVal a : int; print "> ";
             printVal(1 +
                      Int.max(depth0 (tab i) (printVal t2 before print " *"),
                              depth0 (tab i) (printVal t1 before print " %")
                             )
                     )
             before print(" #\n" ^ i))
        | depth0 i L = (print( "\n" ^ i) ; 0)
  in
      depth0 "" f
  end
```

Nyomkövetés kiírással: depth a (akkumulátort használ)

```
fun depth a f =
  let (* depth0 i (f, d) = d + az f fa mélysége; i a behúzáshoz használt füzér
         depth0 : string -> 'a tree * int -> int *)
      fun depth0 i (N(a : int, t1, t2), d) =
            (print("\n" ^ i ^ "<"); printVal a : int; print "> ";
             printVal(Int.max(depth0 (tab i) (printVal t2 before print(" *\n" ^
                                                                       (tab i)),
                                              printVal(d+1) before print " $ "
                                             ),
                              depth0 (tab i) (printVal t1 before print(" %\n" ^
                                                                       (tab i)),
                                              printVal(d+1) before print " & "
                                             )
                             )
                      )
             before print(" #\n" ^ i)
            )
        | depth0 i (L, d) = (print( "\n" ^ i) ; d)
  in
      depth0 "" (f, 0)
  end
```

- A következő fóliákon depth-t és depth a-t hét csomópontból álló teljes fára alkalmazzuk.

Nyomkövetés kiírással: `depth` és `deptha` egy alkalmazása

```
f7 = N(1, N(2, N(4, L, L), N(5, L, L)), N(3, N(6, L, L), N(7, L, L))) : int tree
```

```
- depth f7;

<1> N(3, N(6, L, L), N(7, L, L)) *
    <3> N(7, L, L) *
        <7> L *
            L %
            1 #
        N(6, L, L) %
        <6> L *
            L %
            1 #
        2 #
    N(2, N(4, L, L), N(5, L, L)) %
```

```
- deptha f7;

<1> N(3, N(6, L, L), N(7, L, L)) *
    1 $
    <3> N(7, L, L) *
        2 $
        <7> L *
            3 $
            L %
            3 &
            3 #
        N(6, L, L) %
        2 &
        <6> L *
            3 $
            L %
            3 &
            3 #
        3 #
    N(2, N(4, L, L), N(5, L, L)) %
    1 &
```

Nyomkövetés kiírással: `depth` és `deptha` egy alkalmazása (folyt.)

Folytatás az előző lapról.

```
f7 = N(1, N(2, N(4, L, L), N(5, L, L)), N(3, N(6, L, L), N(7, L, L))) : int tree
```

```
 (depth f7)

    <2> N(5, L, L) *
        <5> L *
            L %
            1 #
        N(4, L, L) %
        <4> L *
            L %
            1 #
        2 #
    3 #

> val it = 3 : int
```

```
 (deptha f7)

    <2> N(5, L, L) *
        2 $
        <5> L *
            3 $
            L %
            3 &
            3 #
        N(4, L, L) %
        2 &
        <4> L *
            3 $
            L %
            3 &
            3 #
        3 #
    3 #
> val it = 3 : int
```

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.

HALMAZMŰVELETEK

Halmazműveletek: „benne van-e?” (isMem) és „ha új, tedd bele” (newMem)

- isMem igaz értéket ad eredményül, ha a keresett elem benne van a listában.

```
(* isMem : ''a * ''a list -> bool
   isMem(x, ys) = x eleme-e ys-nek
*)
fun isMem (_, []) = false
  | isMem (x, y::ys) = x = y orelse isMem(x, ys)

infix isMem
```

- newMem egy új elemet rak be egy listába, ha még nincs benne.

```
(* newMem : ''a * ''a list -> ''a list
   newMem(x, xs) = [x] és xs listaként ábrázolt uniója
*)
fun newMem (x, xs) = if x isMem xs
                     then xs
                     else x::xs
```

newMem, ha a sorrendtől eltekintünk, halmazt hoz létre.

Halmazműveletek: „listából halmaz” (`setof`)

- `setof` halmazt készít egy listából úgy, hogy kiszedi belőle az ismétlődő elemeket. Rossz hatékonyságú.

```
(* setof : ''a list -> ''a list
   setof xs = xs elemeinek listaként ábrázolt halmaza
*)
fun setof []      = []
  | setof (x::xs) = newMem(x, setof xs)
```

- Öt halmazműveletet definiálunk:
  - unió (`union`, $S \cup T$),
  - metszet (`inter`, $S \cap T$),
  - részhalmaza-e (`isSubset`, $T \subseteq S$),
  - egyenlők-e (`isSetEq`, $S = T$),
  - hatványhalmaz (`powerSet`, pS).

Halmazműveletek: „unió” (union) és „metszet” (inter)

- Listaként kezeljük a halmazokat, később hatékonyabb ábrázolást választhatunk, pl. rendezett listát vagy bináris fát.
- Két halmaz uniója

```
(* union : ''a list * ''a list -> ''a list
   union(xs, ys) = az xs és ys elemeiből álló halmazok uniója
*)
fun union ([], ys)     = ys
  | union (x::xs, ys) = newMem(x, union(xs, ys))
```

- Két halmaz metszete

```
(* inter : ''a list * ''a list -> ''a list
   inter(xs, ys) = az xs és ys elemeiből álló halmazok metszete
*)
fun inter ([], _)      = []
  | inter (x::xs, ys) = let val zs = inter(xs, ys)
                        in
                           if x isMem ys then x::zs else zs
                        end
```

Halmazműveletek: „részhalmaza-e” (`isSubset`) és „egyenlők-e” (`isSetEq`)

- Részhalmaza-e egy halmaz egy másiknak?

```
(* isSubset : ''a list * ''a list -> bool
   isSubset (xs, ys) = az xs elemeiből álló halmaz részhalmaza-e
                       az ys elemeiből álló halmaznak
*)
fun isSubset ([], _)      = true
  | isSubset (x::xs, ys) = (x isMem ys) andalso isSubset(xs, ys)

infix isSubset
```

- Két halmaz egyenlősége (a listák egyenlőségvizsgálata beépített művelet az SML-ben, halmazokra mégsem használható, mert pl. `[3, 4]` és `[4, 3]` listaként ugyan különböznek, de halmazként egyenlők)

```
(* isSetEq : ''a list * ''a list -> bool
   isSetEq(xs, ys) = az xs elemeiből álló halmaz egyenlő-e
                     az ys elemeiből álló halmazzal
*)
fun isSetEq (xs, ys) = (xs isSubset ys) andalso (ys isSubset xs)
```

Halmazműveletek: „halmaz hatványhalmaza” (`powerSet`)

A hatványhalmaz megvalósítása SML-ben ezen és a következő két fólián csak olvasmány haladóknak, nem vizsgaanyag.

- Az S halmaz hatványhalmaza *összes* részhalmazának a halmaza, az S-t és a $\{\}$-t is beleértve.
- S hatványhalmaza úgy állítható elő, hogy kivesszük S-ből az x elemet, majd *rekurzív módon* előállítjuk az $S - \{x\}$ hatványhalmazát.
- Ha tetszőleges T halmazra $T \subseteq S - \{x\}$, akkor $T \subseteq S$ és $T \cup \{x\} \subseteq S$, így mind T, mind $T \cup \{x\}$ eleme S hatványhalmazának.
- Miközben a fenti elvet rekurzív módon alkalmazzuk, tehát fölsoroltatjuk az $S - \{x\}$ stb. részhalmazait, gyűjtjük a *már kiválasztott* elemeket. Egy-egy rekurzív lépésben a gyűjtő vagy változatlan (T), vagy kiegészül az x elemmel ($T \cup \{x\}$).
- A `pws` függvényben a `base` argumentumban gyűjtjük a halmaz *már kiválasztott* elemeit; kezdetben üres.
- pws(xs, base) = $\{S \cup \texttt{base} \mid S \subseteq \texttt{xs}\}$, azaz `xs` $\cup$ `base` azon részhalmazainak a listája, amelyek teljes egészében tartalmazzák a base halmazt.

Halmazműveletek: „halmaz hatványhalmaza” (folyt.)

- Ezzel a `pws` függvény:

```
(* pws : 'a list * 'a list -> 'a list list
   pws(xs, base) = mindazon halmazok listája, amelyek előállnak xs egy
                   részhalmazának és a base halmaznak az uniójaként
*)
fun pws ([], base) = [base]
  | pws (x::xs, base) = pws(xs, base) @ pws(xs, x::base)
```

- `pws(xs, base)` valósítja meg az $S - \{x\}$ rekurzív hívást (hiszen `x::xs` felel meg S-nek), azaz állítja elő az összes olyan halmazt, amelyekben `x` nincs benne.
- `pws(xs, x::base)` rekurzív módon `base`-ben gyűjti az `x` elemeket, vagyis előállítja az összes olyan halmazt, amelyben `x` benne van.
- `powerSet`-nek már csak megfelelő módon hívnia kell `pws`-t:

```
(* powerSet : 'a list -> 'a list list
   powerSet xs = az xs halmaz hatványhalmaza
*)
fun powerSet xs = pws(xs, [])
```

Halmazműveletek: „halmaz hatványhalmaza”, hatékonyabban

- `pws` rossz hatékonyságú, mert kétfelé ágazó rekurziót használ. Pl. egy 19 egész számból álló lista hatványhalmazának előállítását nem lehet kivárni. Írjunk hatékonyabb változatot.
- Az `insAll` segédfüggvény egy elemet szúr be egy listákból álló lista minden eleme elé.

```
(* insAll : 'a * 'a list list * 'a list list -> 'a list list
   insAll(x, yss, zss) = az yss lista ys elemeinek zss elé fűzött
                         listája, amelyben minden ys elem elé x van beszúrva *)
fun insAll (x, [], zss) = zss
  | insAll (x, ys::yss, zss) = insAll(x, yss, (x::ys)::zss)
```

- `powerSet` `insAll`-t használó rekurzív változata

```
fun powerSet [] = [[]]
  | powerSet (x::xs) = let val pws = powerSet xs
                       in  pws @ insAll(x, pws, [])
                       end
```

- `powerSet` `insAll`-t használó iteratív változata

```
fun powerSet [] = [[]]
  | powerSet (x::xs) = let val pws = powerSet xs
                       in insAll(x, pws, pws)
                       end
```

EGYIDEJŰ DEKLARÁCIÓ

Egyidejű deklaráció

- Típusok, ill. értékek *egyidejűleg* is deklarálhatók az `and` kulcsszó alkalmazásával.
- Vegyük a következő deklarációsorozatokat:

```
type sor = int; type osz = int;
datatype fa = L | B of fa * fa;
    datatype 'a verem = >| | >> of 'a * 'a verem;
val v1 = "a"; val v2 = "z";
fun f1 i = i +1; fun f2 i = i - 1;
```

Ezeket a deklarációkat az SML-értelmező a *megadott sorrendben* értékeli ki.

```
type sor = int and osz = int;
datatype fa = L | B of fa * fa and
    'a verem = >| | >> of 'a * 'a verem;
val v1 = "a" and v2 = "z";
fun f1 i = i +1 and f2 i = i - 1;
```

Az `and` szócskával elválasztott deklarációkat az SML-értelmező *egyidejűleg* értékeli ki.

Egyidejű deklaráció (folyt.)

- Egyidejű deklarációt kell használnunk kölcsönösen rekurzív függvények definiálására. Példa:

```
fun even 0 = true | even n = odd(n-1)
and odd 0 = false | odd n = even(n-1);
```

- Egyidejű deklarációt használhatunk két vagy több kötés egyidejű felcserélésére. Példa:

```
val v1 = "a"; val v2 = "z"; val v1 = v2 and v2 = v1;
```

- Egyidejű deklarációt használhatunk, ha fölülről lefelé haladva akarunk programot írni. Példa:

```
fun length zs = len zs 0
and len [] i = i | len (_ :: xs) i = len xs (i+1);
```

- A polimorf függvényeket a szekvenciális és az egyidejű deklaráció eltérően kezeli, mivel a típuslevezetést az SML-értelmező a teljes kifejezésre alkalmazza. Példa:

```
fun id x = x; fun hi () = id 3; fun nr () = id 4.0;
fun id x = x  and hi () = id 3  and nr () = id 4.0;
```

A első sor kiértékelésekor `id 'a -> 'a` típusú. A második sor kiértékelésekor `id int -> int` és `real -> real` típusú lenne egyszerre, ami lehetetlen.

AZ ORDER TÍPUS

Az order típus

Az order típus definíciója (ld. General.sig)

```
datatype order = LESS | EQUAL | GREATER

[order] is used as the return type of comparison functions.
```

Példák az SML-alapkönyvtárból (SML Basis Library)

```
Int.compare      : int * int -> order
Char.compare     : char * char -> order
Real.compare     : real * real -> order
String.compare   : string * string -> order
Time.compare     : time * time -> order
```

LISTÁK RENDEZÉSE

Listák rendezése

- **`inssort` (beszúró rendezés),**
- **`selsort` (kiválasztó rendezés),**
- **`quicksort` (gyorsrendezés),**
- **`tmsort` (felülről lefelé haladó összefésülő rendezés),**
- `bmsort` (alulról felfelé haladó összefésülő rendezés),
- `smsort` (simarendezés).

Beszúró rendezés

- Az `ins` segédfüggvény az `x` elemet a megfelelő helyre rakja be az `ys` listában:

```
(* ins : real * real list -> real list
   ins (x, ys) = ys kibővítve x-szel a <= reláció szerint
   PRE: ys a <= reláció szerint rendezve van *)
fun ins (x, y::ys) = if x <= y then x::y::ys else y::ins(x, ys)
  | ins (x : real, []) = [x]
```

- `inssort`-tal rekurzívan rendezzük a lista maradékát; végrehajtási ideje $O(n^2)$:

```
(* inssort : ('a * 'b list -> 'b list) -> 'a list -> 'b list
   inssort f xs = az xs elemeiből álló, az f felhasználásával
                  rendezett lista *)
fun inssort f (x::xs) = f(x, inssort f xs)
  | inssort _  [] = []
```

- Példa `inssort` alkalmazására:

```
inssort ins [4.24, 4.1, 5.67, 1.12, 4.1, 0.33, 8.0]
```

Beszúró rendezés, generikus változat

- Az `ins` függvényt generikussá tesszük:

```
(* ins : ('a * 'a -> bool) -> 'a * 'a list -> 'a list
   ins cmp (x, ys) = ys kibővítve x-szel a cmp reláció szerint
   PRE: ys a cmp reláció szerint rendezve van *)
fun ins cmp (x, ys) =
      let fun ins0 (y::ys) =
                 if cmp(x, y) then x::y::ys else y::ins0 ys
            | ins0 [] = [x]
      in  ins0 ys
      end
```

- Ezzel `inssort` egy újabb változata:

```
(* inssort : ('a * 'a -> bool) -> 'a list -> 'a list
   inssort cmp xs = az xs elemeiből álló, a cmp reláció
                    szerint rendezett lista *)
fun inssort cmp (x::xs) = ins cmp (x, inssort cmp xs)
  | inssort _ [] = []
```

Beszúró rendezés, generikus változat (folyt.)

- `inssort` eddigi változatai előbb elemeire szedik szét a rendezendő listát, majd hátulról visszafelé haladva, rendezés közben építik fel az újat.
- A jobbrekurziót és akkumulátort használó változatnak (`inssort2`) kisebb veremre van szüksége, mivel a listáról leválasztott elemeket balról jobbra haladva azonnal berakja a helyükre az eredménylistában. (A két megoldás futási idejét később összehasonlítjuk).

```
(* inssort2 : ('a * 'a -> bool) -> 'a list -> 'a list
   inssort2 cmp xs = az xs elemeiből álló, a cmp reláció
                     szerint rendezett lista *)
fun inssort2 cmp xs =
      let (* sort :  'a list -> 'a list -> 'a list
             sort xs zs = zs kibővítve az xs-nek a cmp reláció
                          szerint rendezett elemeivel
             PRE: zs cmp szerint rendezve van *)
          fun sort (x::xs) zs = sort xs (ins cmp (x, zs))
            | sort [] zs = zs
      in
          sort xs []
      end
```

Beszúró rendezés `foldr`-rel és `foldl`-lel

- A második argumentumát akkumulátorként használó `foldl` kisebb vermet használ `foldr`-nél, ezért `inssortL` hosszabb listákat tud rendezni:

```
fun inssortR cmp = foldr (ins cmp) []
fun inssortL cmp = foldl (ins cmp) []
```

- Példák `insort`-tal és `insort2`-vel:

```
inssort op<= [4.24, 4.1, 5.67, 1.12, 4.1, 0.33, 8.0];
inssort2 op>= [4, 4, 5, 1, 0, 8];
inssort op< (explode "qwerty")
```

- Példák `foldr` és `foldl` felhasználásával:

```
fun inssortRi cmp = foldr (ins cmp)[];
fun inssortLr cmp = foldl (ins cmp)([] : real list)

inssortRi op>= [4, 4, 5, 1, 0, 8];
inssortLr op>= [4.24, 4.1, 5.67, 1.12, 4.1, 0.33, 8.0]
```

A futási idők mérése, összehasonlítása

- 2000 elemet tartalmazó, véletlenszerűen előállított, illetve eredetileg éppen fordított sorrendű listák rendezéséhez szükséges futási időt mérünk.
  - Véletlen eloszlású egészlistát állít elő a `Random` könyvtárbeli `rangelist` függvény:

```
val xs2000R =
      Random.rangelist (1, 100000) (2000, Random.newgen());
```

  - Növekvő sorrendű egészlistát állít elő a `--` operátor:

```
infix --;
fun fm -- to =
      let fun upto to zs =
                if to < fm then zs else upto (to-1) (to::zs)
      in
          upto to []
      end;

val xs2000N = 1 -- 2000;
```

A futási idők mérése, összehasonlítása (folyt.)

- A futási időt az alábbi függvénnyel mérhetjük:

```
fun futIdo (sort, sortFn) (cmp, cmpFn) (xs, kind) =
      let val starttime = Timer.startCPUTimer()
          val zs = sort cmp xs
          val usr=tim,... = Timer.checkCPUTimer starttime
      in
          "Int sort with " ^ sortFn ^ ", " ^ cmpFn ^
          ", length = " ^ Int.toString(length xs) ^ " (" ^
          kind ^ "), time = " ^ Time.fmt 2 tim ^ " sec\n"
      end;

val t1N =
      futIdo (inssort, "inssort") (op>=, "op>=") (xs2000N, "increasing");
val t2N =
      futIdo (inssort2, "inssort2") (op>=, "op>=") (xs2000N, "increasing");
val t1R =
      futIdo (inssort, "inssort") (op>=, "op>=") (xs2000R, "random");
val t2R =
      futIdo (inssort2, "inssort2") (op>=, "op>=") (xs2000R, "random");
```

A futási idők mérése, összehasonlítása (folyt.)

- A 2000 elemű, fordított sorrendű lista rendezése az akkumulátort nem használó `inssort`-változatokkal több mint 5 s-ig, az akkumulátort használó változatokkal csak 0.01 s-ig tart (linux, 233 MHz-es Pentium).

```
Int sort with inssort, op>=, length = 2000 (increasing), time = 5.18 sec
Int sort with inssort2, op>=, length = 2000 (increasing), time = 0.01 sec
Int sort with inssortRi, op>=, length = 2000 (increasing), time = 5.14 sec
Int sort with inssortLi, op>=, length = 2000 (increasing), time = 0.01 sec
```

- Eltűnik a különbség, ha ugyanolyan hosszú, de véletlenszerűen előállított listákat rendezünk.

```
Int sort with inssort, op>=, length = 2000 (random), time = 2.39 sec
Int sort with inssort2, op>=, length = 2000 (random), time = 2.26 sec
Int sort with inssortRi, op>=, length = 2000 (random), time = 2.40 sec
Int sort with inssortLi, op>=, length = 2000 (random), time = 2.24 sec
```

Kiválasztó rendezés

```
(* selsort : ('a * 'a -> order) -> 'a list -> 'a list
   selsort cmp xs = az xs elemei cmp szerint növekvő sorrendben
*)
fun selsort cmp xs =
      let
          (* max : 'a * 'a -> 'a
             max (x, y) = x és y közül cmp szerint a nagyobb
          *)
          fun max (x, y) = if cmp(x, y) = GREATER then x else y

          (* min : 'a * 'a -> 'a
             min (x, y) = x és y közül cmp szerint a kisebb
          *)
          fun min (x, y) = if cmp(x, y) = LESS then x else y

          (* maxSelect : 'a * 'a list * 'a list -> 'a * 'a list
             maxSelect (x, ys, zs) = pár, amelynek első tagja az
                (x::ys) cmp szerinti legnagyobb eleme, második
                tagja az x::ys többi eleméből és a zs
                elemeiből álló lista                              *)
          fun maxSelect (x, [], zs) = (x, zs)
            | maxSelect (x, y::ys, zs) =
                          maxSelect(max(x, y), ys, min(x,y)::zs)
```

Kiválasztó rendezés (folyt.)

```
        (* sSort : 'a list * 'a list -> 'a list
           sSort (xs, ws) = az xs elemei cmp szerint növekvő
                            sorrendben a ws elé fűzve          *)
        fun sSort ([], ws) = ws
          | sSort (x::xs, ws) =
              let val (z, zs) = maxSelect(x, xs, [])
              in
                 sSort (zs, z::ws)
              end
    in
        sSort (xs, [])
    end

app load ["Int","Char","Real"];

selsort Int.compare [1,2,3,4,5,6,7,8,9];
selsort Int.compare [9,8,7,6,5,4,3,2,1];
selsort Real.compare [4.5,6.7,3.6,4.3,1.2,0.9,8.9,9.8,2.0];
selsort Char.compare (explode "Ej mi a ko tyukanyo");
```

Gyorsrendezés akkumulátor nélkül

```
(* quicksort1 cmp xs = az xs elemeinek cmp szerint rendezett listája
   quicksort1 : ('a * 'a -> order) -> 'a list -> 'a list
*)
fun quicksort1 cmp xs =
      let (* qs : 'a list -> 'a list
             qs ys = ys elemeinek cmp szerint rendezett listája
          *)
          fun qs (m::ys) =
                let (* partition : 'a list * 'a list * 'a list -> 'a list
                       partition (xs, ls, rs) = olyan pár, amelynek első tagja
                         az xs m-nél kisebb elemeinek a listája ls elé fűzve,
                         második tagja pedig az xs többi eleme rs elé fűzve *)
                    fun partition (x::xs, ls, rs) =
                          if cmp(x, m) = LESS then partition(xs, x::ls, rs)
                                             else partition(xs, ls, x::rs)
                      | partition ([], ls, rs) = qs ls @ (m::qs rs)
                in
                    partition (ys, [], [])
                end
            | qs [] = []
      in
          qs xs
      end;
```

Gyorsrendezés akkumulátorral

```
(* quicksort2 cmp xs = az xs elemeinek cmp szerint rendezett listája
   quicksort2 : ('a * 'a -> order) -> 'a list -> 'a list
*)
fun quicksort2 cmp xs =
      let (* qs : 'a list -> 'a list -> 'a list
             qs ys zs = ys elemeinek cmp szerint rendezett listája zs elé fűzve
           *)
          fun qs (m::ys) zs =
                let (* partition : 'a list * 'a list * 'a list -> 'a list
                       partition (xs, ls, rs) = olyan pár, amelynek első tagja
                         az xs m-nél kisebb elemeinek a listája ls elé fűzve,
                         második tagja pedig az xs többi eleme rs elé fűzve *)
                    fun partition (x::xs, ls, rs) =
                          if cmp(x, m) = LESS then partition(xs, x::ls, rs)
                                             else partition(xs, ls, x::rs)
                      | partition ([], ls, rs) = qs ls (m :: qs rs zs)
                in
                    partition (ys, [], [])
                end
            | qs [] zs = zs
      in
          qs xs []
      end;
```

A futási idők mérése, összehasonlítása

```
val t1 = futIdo (inssort2, "inssort2") (op>=, "op>=") (xs2000R, "random");
                                                  (* ~ 2 M összehasonlítás! *)
val t3 = futIdo (quicksort2, "quicksort2")
                (Int.compare, "Int.compare") (xs20000R, "random");
val t4 = futIdo (Listsort.sort, "Listsort.sort")
                (Int.compare, "Int.compare") (xs20000R, "random");
                                                  (* ~ 300 E összehasonlítás *)

Int sort with inssort2, op>=, length = 2000 (random), time = 2.30 sec

Int sort with quicksort1, Int.compare, length = 20000 (random), time = 2.18 sec
Int sort with quicksort2, Int.compare, length = 20000 (random), time = 1.72 sec
Int sort with Listsort.sort, Int.compare, length = 20000 (random), time = 1.76 sec

Int sort with quicksort2, Int.compare, length = 200000 (random), time = 27.13 sec
Int sort with quicksort1, Int.compare, length = 200000 (random), time = 32.59 sec

val t7 = futIdo (Listsort.sort, "Listsort.sort") (Int.compare, "Int.compare")
         (Random.rangelist (1, 100000) (200000, Random.newgen()), "random");
! Uncaught exception:
! Out_of_memory
```

Összefésülő rendezések

- Az összefésülő rendezéshez kell egy olyan függvény, amely két listát növekvő sorrendben egyesít.

```
(* merge(xs, ys) = xs és ys elemeinek <= szerint
                   egyesített listája
   merge : int list * int list -> int list
*)
fun merge (xxs as x::xs, yys as y::ys)=
      if x <= y
      then x::merge(xs, yys)
      else y::merge(xxs, ys)
  | merge ([], ys) = ys
  | merge (xs, []) = xs;
```

- Korlátot jelent, ha a részeredményeket a veremben tároljuk.
- Akkumulátor használata esetén meg kell fordítani az eredménylistát.

Fölülről lefelé haladó összefésülő rendezés

- A fölülről lefelé haladó összefésülő rendezés *(top-down merge sort)* akkor hatékony, ha közel azonos hosszúságú az a két lista, amelyekre a rendezendő listát szétszedjük.

```
(* tmsort xs = az xs elemeinek a <= reláció szerint
               rendezett listája
   tmsort : int list -> int list
*)
fun tmsort xs = let val h = length xs
                    val k = h div 2
                in
                   if h > 1
                   then merge(tmsort(List.take(xs, k)),
                              tmsort(List.drop(xs, k)))
                   else xs
                end;
```

- A legrosszabb esetben $O(n \cdot log\, n)$ lépésre van szükség.