# MODULOK



## Szignatúra és struktúra

### Alapkonstrukciók

- szignatúra (`signature`): a struktúra *specifikációja*, *„típusa”*,
- struktúra (`structure`): a szignatúra *megvalósítása*.

### Szignatúra

- típuskonstruktorok *(type constructors)*,
- kivételkonstruktorok *(exception constructors)* és
- értékkötések *(value bindings)* specifikációjából állhat.

### A szignatúra alapváltozata

`sig` *`specs`* `end` alakú kifejezés,
ahol a *`specs`* specifikációsorozat az alábbi elemeket tartalmazhatja:

- típusspecifikáció `type` *`(tyvar`*$_1$*`, ..., tyvar`*$_n$*`) tycon`* `[ = typ ]` alakban, ahol *`typ`* opcionális;
- adattípus-specifikáció (a `datatype`-deklarációval azonos alakban);
- kivételspecifikáció `exception` *`excon`* `of` *`typ`* alakban;
- értékspecifikáció `val` *`id`* `:` *`typ`* alakban.





## Szignatúra és struktúra (folyt.)

### Struktúra

- típuskonstruktorok *(type constructors)*,
- kivételkonstruktorok *(exception constructors)* és
- értékkötések *(value bindings)* specifikációjából állhat.

### A struktúra alapváltozata

`struct` *`decs`* `end` alakú kifejezés,
ahol a *`decs`* deklarációsorozat az alábbi elemeket tartalmazhatja:

- típuskonstruktort létrehozó típusdeklaráció;
- új (felhasználói) adattípust létrehozó adattípus-deklaráció (`datatype`-deklaráció);
- kivételkonstruktort (állandót vagy függvényt) létrehozó kivételdeklaráció;
- adott típusú új nevet definiáló értékdeklaráció.





## Szignatúra és struktúra (folyt.)

### A szignatúra-deklaráció `signature` *`sigid`* `=` *`sigexp`* alakú, ahol

- *`sigid`* egy szignatúranév,
- *`sigexp`* pedig egy szignatúrakifejezés.

### A struktúra-deklaráció egyszerű változata `structure` *`strid`* `=` *`strexp`* alakú, ahol

- *`strid`* egy struktúranév,
- *`strexp`* pedig egy struktúrakifejezés.

### A struktúra-deklaráció bonyolultabb változata: struktúra szignatúrához kötése

- áttetsző (opál): `structure` *`strid`* `:>` *`sigid`* `=` *`strexp`*
- átlátszó (transzparens): `structure` *`strid`* `:` *`sigid`* `=` *`strexp`*

## Példa: a féléves nagyházi KCsiga és Csiga szignatúrája

- A KCsiga keretprogram szignatúrája

```
signature KCsiga =
sig
  val csiga_be : string -> TCsiga.feladvany_leiro
  val csiga_ki : string * TCsiga.csigatabla list -> unit
  val megold   : string * string -> string
end
```

- A Csiga főmodul szignatúrája

```
signature Csiga =
sig
  val buvos_csiga :
        TCsiga.feladvany_leiro -> TCsiga.csigatabla list
end
```





## Példa: a féléves nagyházi TCsiga struktúrája és szignatúrája

- A TCsiga típusleíró-struktúra és *törzs*szignatúrája

```
structure TCsiga =                      (* TCsiga.sml
struct                                     törzsszignatúrája *)
  type meret          = int             type meret          = int
  type ciklus         = int             type ciklus         = int
  type sorszam        = int             type sorszam        = int
  type oszlopszam     = int             type oszlopszam     = int
  type ertek          = int             type ertek          = int
  type adott_elem     =                 type adott_elem     =
       sorszam * oszlopszam * ertek           int * int * int

  type feladvany_leiro =                type feladvany_leiro =
       meret * ciklus * adott_elem list       int * int * (int * int * int) list

  type ertek_vagy_ures = int            type ertek_vagy_ures = int
  type sor             =                type sor             =
       ertek_vagy_ures list                   int list

  type csigatabla     =                 type csigatabla     =
       sor list                               int list list
end
```

- *Törzsszignatúra* (principal signature): egy struktúra összetevőinek legspecifikusabb leírása.





## Példa: változatok a TCsiga-struktúrára és szignatúrára

- Struktúra *átlátszó* (transzparens, transparent) szignatúrával

```
structure TCsiga : TCsiga =
struct
  type meret          = int
  ...
  type adott_elem     =
       sorszam * oszlopszam * ertek
  type feladvany_leiro =
       meret * ciklus * adott_elem list
  type ertek_vagy_ures = int
  type sor             =
       ertek_vagy_ures list
  type csigatabla     = sor list
end
```

- Struktúra *áttetsző* (opál, opaque) szignatúrával

```
structure TCsiga :> TCsiga =
struct
  type meret          = int
  ...
  type adott_elem     =
       sorszam * oszlopszam * ertek
  type feladvany_leiro =
       meret * ciklus * adott_elem list
  type ertek_vagy_ures = int
  type sor             =
       ertek_vagy_ures list
  type csigatabla     = sor list
end
```

- Szignatúra (a részleteket elrejtő) absztrakt adattípus megvalósításához

```
signature TCsiga =
sig
    type feladvany_leiro
    type csigatabla
end
```





## Példa: sort megvalósító szignatúra és struktúra

```
signature QUEUE =
sig
    type 'a queue
    exception Empty
    val empty : 'a queue
    val insert : 'a * 'a queue -> 'a queue
    val remove : 'a queue -> 'a * 'a queue
end

structure Queue_as_lists =
struct
    type 'a queue = 'a list * 'a list
    exception Empty
    val empty = (nil, nil)
    fun insert (x, (b,f)) = (x::b, f)
    fun remove (nil, nil) = raise Empty
      | remove (bs, nil) = remove (nil, rev bs)
      | remove (bs, f::fs) = (f, (bs, fs))
end
```

## Példa: sort megvalósító szignatúra és struktúra (folyt.)

- Struktúra áttetsző szignatúrával

```
structure Queue_as_lists :> QUEUE =
struct type 'a queue = 'a list * 'a list
       exception Empty
       val empty = (nil, nil)
       fun insert (x, (b,f)) = (x::b, f)
       fun remove (nil, nil) = raise Empty
         | remove (bs, nil) = remove (nil, rev bs)
         | remove (bs, f::fs) = (f, (bs, fs))
end
```

- Struktúra átlátszó szignatúrával

```
structure Queue_as_lists : QUEUE =
struct type 'a queue = 'a list * 'a list
       exception Empty
       val empty = (nil, nil)
       fun insert (x, (b,f)) = (x::b, f)
       fun remove (nil, nil) = raise Empty
         | remove (bs, nil) = remove (nil, rev bs)
         | remove (bs, f::fs) = (f, (bs, fs))
end
```





## Példa: sort megvalósító szignatúra és struktúra (folyt.)

- Struktúra *bővített* áttetsző szignatúrával

```
structure Queue_as_lists :>
            QUEUE where type 'a queue = 'a list * 'a list =
struct type 'a queue = 'a list * 'a list
       exception Empty
       val empty = (nil, nil)
       fun insert (x, (b,f)) = (x::b, f)
       fun remove (nil, nil) = raise Empty
         | remove (bs, nil) = remove (nil, rev bs)
         | remove (bs, f::fs) = (f, (bs, fs))
end
```

- Áttetsző szignatúra-kötésnél a típusok megvalósítása rejtve marad, vagyis a *látható szignatúra* **független** a struktúra megvalósításától (pl. `type 'a queue = 'a queue`);
- átlátszó szignatúra-kötésnél a típusok megvalósítása láthatóvá válik, vagyis a *látható szignatúra* **függ** a struktúra megvalósításától (pl. `type 'a queue = 'a list * 'a list`);
- A megvalósítástól független modulrendszer kialakításához áttetsző szignatúra-kötést kell alkalmazni. Ezt garantálja az `mosmlc` fordító `structure`-módban.





## Példa: sort megvalósító szignatúra és struktúra (folyt.)

- Szignatúra-öröklődés bővítéssel, 1. változat

```
signature QUEUE_AS_LISTS =
    QUEUE where type 'a queue = 'a list * 'a list
```

- Szignatúra-öröklődés bővítéssel, 2. változat (ekvivalens az 1. változattal)

```
signature QUEUE_AS_LISTS =
sig
    include QUEUE
end where type 'a queue = 'a list * 'a list
```

- Struktúra áttetsző szignatúrával

```
structure Queue_as_lists :> QUEUE_AS_LISTS =
struct
    type 'a queue = 'a list * 'a list
    exception Empty
    val empty = (nil, nil)
    fun insert (x, (b,f)) = (x::b, f)
    fun remove (nil, nil) = raise Empty
      | remove (bs, nil) = remove (nil, rev bs)
      | remove (bs, f::fs) = (f, (bs, fs))
end
```





## Példa: sort megvalósító szignatúra és struktúra (folyt.)

- Szignatúra-öröklődés bővítéssel

```
signature QUEUE_AS_LISTS_WITH_EMPTY =
sig
    include QUEUE
    val is_empty : 'a queue -> bool
end where type 'a queue = 'a list * 'a list
```

- Struktúra-öröklődés (hibás: `type 'a queue = 'a list * 'a list` nincs deklarálva a modulban)

```
structure Queue_as_listsWithEmpty :> QUEUE_AS_LISTS_WITH_EMPTY =
struct
   structure Q = Queue_as_lists :
     QUEUE where type 'a queue = 'a list * 'a list
end
```

- Struktúra-öröklődés (hibás: `val 'a is_empty :  'a list * 'a list -> bool` nincs deklarálva a modulban)

```
structure Queue_as_listsWithEmpty :> QUEUE_AS_LISTS_WITH_EMPTY =
struct
    structure Q = Queue_as_lists
    open Q
end
```

- Struktúra-öröklődés

```
structure Queue_as_listsWithEmpty :> QUEUE_AS_LISTS_WITH_EMPTY =
struct
    structure Q = Queue_as_lists
    open Q
    fun is_empty (nil, nil) = true
      | is_empty _ = false
end
```

- Struktúra-öröklődés, ekvivalens az előzővel

```
structure Queue_as_listsWithEmpty :> QUEUE_AS_LISTS_WITH_EMPTY =
struct
    structure Q : QUEUE_AS_LISTS = Queue_as_lists
    open Q
    fun is_empty (nil, nil) = true | is_empty _ = false
end
```

- Struktúra-öröklődés, ekvivalens az előzővel

```
structure Queue_as_listsWithEmpty :> QUEUE_AS_LISTS_WITH_EMPTY =
struct
    structure Q = Queue_as_lists : QUEUE_AS_LISTS
    open Q
    fun is_empty (nil, nil) = true | is_empty _ = false
end
```



# 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).**





## Összefésülő rendezések (ism.)

- 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
```

- Hatékonyságromlást okoz, hogy a részeredményeket a veremben tároljuk. Iteratív megoldás esetén meg kell fordítani az eredménylistát.

## Alulról fölfelé haladó összefésülő rendezés

- Az alulról fölfelé haladó összefésülő rendezés *(bottom-up merge sort)* legegyszerűbb változata az eredeti $k$ hosszúságú listát $k$ darab egyelemű listára bontja, majd a szomszédos listákat összefuttatja, így 2, 4, 8, 16 stb. elemű listákat állít elő.
- R. O´Keefe algoritmusa (1982) lépésről lépésre futtatja össze az egyforma hosszú részlistákat, de csak az utolsó lépésben rendezi az összeset. Az alábbi példában az összefuttatott részlistákat *egymás mellé írással* jelöljük:

```
AB  C D E F G H I J K
AB  CD  E F G H I J K
ABCD    E F G H I J K
ABCD    EF  G H I J K
ABCD    EF  GH  I J K
ABCD    EFGH    I J K
ABCDEFGH        I J K
ABCDEFGH        IJ  K
...
```





## Alulról fölfelé haladó összefésülő rendezés (folyt.)

- `bmsort` a `sorting` segédfüggvényt használja, amelynek
  - első argumentuma a rendezendő lista,
  - második argumentuma a már rendezett részlistákat gyűjti,
  - harmadik argumentuma az adott lépésben összefuttatandó elem sorszáma.

```
(* bmsort xs = az xs elemeinek a <= reláció szerint
                rendezett listája
   bmsort : int list -> int list
*)
fun bmsort xs = sorting(xs, [], 0)
```





## Alulról fölfelé haladó összefésülő rendezés (folyt.)

- Ha a rendezendő lista (`xs`) még nem fogyott el, soron következő eleméből `sorting` egyelemű listát (`[x]`) képez, és ezt a már rendezett részlisták listája (`lss`) elé fűzve meghívja a `mergepairs` segédfüggvényt. `mergepairs` az argumentumként átadott lista két azonos hosszúságú bal oldali részlistáját fűzi egybe, feltéve persze, hogy vannak ilyenek. `k` az éppen átadott elem sorszáma. Ha a rendezendő lista kiürült, `sorting` a kétszintű lista egyetlen elemét, a rendezett listát adja eredményül.

```
(* sorting(xs, lss, k) = a még rendezetlen xs lista elemeit
                         berakja a k elemet tartalmazó, már
                         rendezett lss listába
   sorting : int list * int list list * int -> int list
   PRE: k >= 0
*)
fun sorting (x::xs, lss, k) =
              sorting(xs, mergepairs([x]::lss, k+1), k+1)
  | sorting ([], lss, k) = hd(mergepairs(lss, 0))
```





## Alulról fölfelé haladó összefésülő rendezés (folyt.)

- `mergepairs` egyetlen listában gyűjti a már összefuttatott részlistákat. Az éppen átadott elem `k` sorszámából dönti el, hogy mit kell csinálnia a következő részlistával.

```
(* mergepairs(llss, n)= az n elemet tartalmazó, már
                        rendezett llss lista első két részlistáját,
                        ha egyforma a hosszuk, összefuttatja
   mergepairs : int list list -> int list list
   PRE: n >= 0
*)
fun mergepairs (llss as ls1::ls2::lss, n) =
      (* legalább kételemű a lista *)
      if n mod 2 = 1
      then llss
      else mergepairs(merge(ls1, ls2)::lss, n div 2)
  | mergepairs (lss, _) = lss (* egyelemű a lista *)
```

- Ha `n` páratlan, `mergepairs` a listát változtatás nélkül adja vissza, ha páros, akkor az `llss` lista elején álló két, egyforma hosszú listát egyetlen rendezett listává futtatja össze. `n=0`-ra `mergepairs` az összes listák listáját olyan listává futtatja össze, amelynek egyetlen eleme maga is lista.

## Alulról fölfelé haladó összefésülő rendezés (folyt.)

- A legrosszabb esetben $O(n \cdot log\, n)$ lépésre van szükség.
- A függvények működését egy példán is bemutatjuk. A kezdőhívás legyen

```
bmsort [1,2,3,4,5,6,7,8,9]
    ---> sorting ([1,2,3,4,5,6,7,8,9], [], 0)
```

- Amíg `sorting` első argumentuma a nem üres `(x::xs)` lista, `sorting` saját magát hívja meg. A rekurzív hívás
  - első argumentuma a lépésenként egyre rövidülő `xs` lista,
  - második argumentuma a `mergepairs([x]::lss, k+1)` függvényalkalmazás eredménye, ahol kezdetben `lss = []`,
  - harmadik argumentuma (`k+1`) a már feldolgozott listaelemek száma.

```
fun sorting (x::xs, lss, k) =
            sorting(xs, mergepairs([x]::lss, k+1), k+1)
  | sorting ([], lss, k) = hd(mergepairs(lss, 0))
```





## Alulról fölfelé haladó összefésülő rendezés (folyt.)

- A következő táblázatos elrendezés
  - `mergepairs` mindkét argumentumát,
  - a rekurzív `sorting` hívás itt `j`-vel jelölt 3. argumentumát, `k+1`-et, és
  - bináris számként `k`-t mutatja lépésről lépésre.
- A `sorting` függvény hívja `mergepairs`-t azokban a sorokban, amelyekben a `j` új értéket vesz föl, a többi helyen `mergepairs` hívása rekurzív.
- Ne feledjük, hogy `mergepairs`-nek listák listája az első argumentuma!
- A táblázat utolsó oszlopa a vonatkozó magyarázatra hivatkozik.
- Vegyük észre, hogy kapcsolat van az `llss` első eleme utáni listaelemek hossza és a `k` bitjei között! Ha `k` valamelyik bitje 1, akkor (balról jobbra haladva) az `llss` megfelelő listaelemének a hossza az adott bit helyiértékével egyenlő. A 0 értékű biteknek megfelelő listaelemek „hiányoznak” `llss`-ből.

```
fun sorting (x::xs, lss, k) =
            sorting(xs, mergepairs([x]::lss, k+1), k+1)
  | sorting ([], lss, k) = hd(mergepairs(lss, 0))
```





## Alulról fölfelé haladó összefésülő rendezés (folyt.)

| llss | n | j | k | |
|---|---|---|---|---|
| `[[1]]` | 1 | 1 | 0 | m1 |
| `[[2],[1]]` | 2 | 2 | 1 | m2 |
| `[[1,2]]` | 1 | | | m3 |
| `[[3],[1,2]]` | 3 | 3 | 10 | m3 |
| `[[4],[3],[1,2]]` | 4 | 4 | 11 | m2 |
| `[[3,4],[1,2]]` | 2 | | | m2 |
| `[[1,2,3,4]]` | 1 | | | m3 |
| `[[5],[1,2,3,4]]` | 5 | 5 | 100 | m3 |
| `[[6],[5],[1,2,3,4]]` | 6 | 6 | 101 | m2 |
| `[[5,6],[1,2,3,4]]` | 3 | | | m3 |
| `[[7],[5,6],[1,2,3,4]]` | 7 | 7 | 110 | m3 |
| `[[8],[7],[5,6],[1,2,3,4]]` | 8 | 8 | 111 | m2 |
| `[[7,8],[5,6],[1,2,3,4]]` | 4 | | | m2 |
| `[[5,6,7,8],[1,2,3,4]]` | 2 | | | m2 |
| `[[1,2,3,4,5,6,7,8]]` | 1 | | | m3 |
| `[[9],[1,2,3,4,5,6,7,8]]` | 9 | 9 | 1000 | m3 |
| `[[9],[1,2,3,4,5,6,7,8]]` | 0 | 0 | | m4 |
| `[[1,2,3,4,5,6,7,8,9]]` | | | | |

```
fun sorting (x::xs, lss, k) =
      sorting(
              xs,
              mergepairs([x]::lss, k+1),
              k+1
              )
  | sorting ([], lss, k) =
      hd(mergepairs(lss, 0))
```

**m1:** Az argumentumként átadott listának egyetlen eleme van (maga is lista), ezért az argumentumot mergepairs második klóza változtatás nélkül visszaadja az őt hívó `sorting`-nak.

**m2:** `n` páros, ez azt jelzi, hogy az argumentumként átadott lista első két eleme egyforma hosszú lista, amelyeket `merge` egyetlen rendezett listává futtat össze, majd az eredménnyel `mergepairs` első klóza meghívja saját magát.

**m3:** `n` páratlan, ez azt jelzi, hogy az argumentumként átadott lista első két eleme nem egyforma hosszú lista, ezért az argumentumot `mergepairs` első klóza változtatás nélkül visszaadja az őt hívó `sorting`-nak.

**m4:** `n=0`, az összes listák listáját olyan listává kell összefuttatni, amelynek egyetlen lista az eleme.





## Simarendezés

- Az applikatív simarendezés (*smooth sort)* algoritmusa O’Keefe alulról fölfelé haladó rendezéséhez hasonló, de nem egyelemű listákat, hanem növekvő *futamokat* állít elő.
- Ha a futamok száma $n$-től független, azaz a lista majdnem rendezve van, akkor az algoritmus végrehajtási ideje $O(n)$, és a legrosszabb esetben is legfeljebb csak $O(n \cdot log\, n)$.

```
(* nextrun : int list * int list -> int list * int list
   nextrun (run, xs) = ... *)
fun nextrun (run, x::xs) =
      if x < hd run
      then (rev run, x::xs)
      else nextrun(x::run, xs)
  | nextrun (run, []) = (rev run, [])
```

- `nextrun` eredménye egy pár, ennek
  - első tagja a futam (egy növekvő számsorozat),
  - a második tagja pedig a rendezendő lista maradéka.

## Simarendezés (folyt.)

- A futam csökkenő sorrendben bővül, kilépéskor a futamot meg kell fordítani. `smsorting` a futamokat ismételten előállítja és összefuttatja:

```
(* smsorting : int list * int list list * int -> int list
   smsorting (xs, lss, k) = ... *)
fun smsorting (x::xs, lss, k) =
        let val (run, tail) = nextrun([x], xs)
        in
           smsorting(tail, mergepairs(run::lss, k+1), k+1)
        end
  | smsorting ([], lss, k) = hd(mergepairs(lss, 0))
```

- 
```
(* smsort : int list -> int list
   smsort xs = az xs elemeinek a <= reláció szerint
               rendezett listája
*)
fun smsort xs = smsorting(xs, [], 0)
```

- A simarendezés egy változata `sort` néven megtalálható a `Listsort` könyvtárban.





## A futási idők összehasonlítása

```
fun futIdo2 (sort, sortFn) (xs, kind) =
      let val starttime = Timer.startCPUTimer()
          val zs = sort xs
          val usr=tim,... = Timer.checkCPUTimer starttime
      in  "Int sort with " ^ sortFn ^ ", length = " ^ Int.toString(length xs) ^
          " (" ^kind ^ "), time = " ^ Time.fmt 2 tim ^ " sec\n"
      end

val t101 = futIdo2 (tmsort, "tmsort")
            ((Random.rangelist (1, 100000) (100000, Random.newgen())), "random");
val t102 = futIdo2 (bmsort, "bmsort")
            ((Random.rangelist (1, 100000) (100000, Random.newgen())), "random");
val t103 = futIdo2 (smsort, "smsort")
            ((Random.rangelist (1, 100000) (100000, Random.newgen())), "random")

Int sort with tmsort, length = 100000 (random), time = 10.96 sec
Int sort with bmsort, length = 100000 (random), time = 7.69 sec
Int sort with smsort, length = 100000 (random), time = 7.70 sec
Int sort with quicksort2, Int.compare, length = 100000 (random), time = 11.98 sec
Int sort with Listsort.sort, Int.compare, length = 100000 (random), time = 14.17 sec
```