

AZ SML MODULNYELVE

Az SML modulnyelve FP-254

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, (bs, fs)) = (x::bs, fs)
  fun remove (nil, nil) = raise Empty
    | remove (bs, nil) = remove (nil, rev bs)
    | remove (bs, f::fs) = (f, (bs, fs))
end
```

típusspecifikáció
kivételspecifikáció
értékspecifikációk

Minden típus és érték polimorf!

típuskonstruktor deklarálása
kivételkonstruktor deklarálása
névkonstruktorok deklarálása

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, (bs, fs)) =
    (x::bs, fs)
  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, (bs, fs)) =
    (x::bs, fs)
  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, (bs, fs)) = (x::bs, fs)
  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 specializációval, 1. változat

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

- Szignatúra-öröklődés specializációval, 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, (bs, fs)) = (x::bs, fs)
  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 inklúzióval (és az eredeti szignatúra bővítésével)

```
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: a Q-beli nevek rövid változata nem látszik Queue_as_listsWithEmpty-ben, pedig specifikálva vannak Queue_as_lists-ben)

```
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 Queue_as_lists-ben)

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

Példa: sort megvalósító szignatúra és struktúra (folyt.)

• 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
```

Modulfogalmak és elnevezések

• Eddig kétféle modulkonstrukcióval foglalkoztunk, a szignatúrával és a struktúrával:

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

• Az SML egy harmadik modulkonstrukciót is ismer, a funktort (functor):

- olyan generikus konstrukció, amelynek *struktúra* a paramétere (és az eredménye is);
- akkor használjuk, amikor a polimorfizmus kevés újrafelhasználható algoritmusok írásához.

• Az MOSML további két elnevezést használ (ld. *Moscow ML Language Overview*):

- modul (*module*): a struktúra és a funktor *közös* megnevezése;
- (fordítási, ill. lefordított) *egység* (compilation, ill. compiled *unit*): egy struktúra vagy szignatúra lefordítható, ill. lefordított (tárgykódú) változata.

• Állománynév-kiterjesztések

- `.sml` (SML, opcionális): struktúra vagy funktor,
- `.sig` (SML, kötelező): szignatúra,
- `.ui` (MOSML, kötelező): szignatúra lefordított változata (unit interface code),
- `.uo` (MOSML, kötelező): struktúra lefordított változata (unit object code).

Szignatúra-illeszkedés

- Mikor tekinthető egy struktúra egy szignatúra megvalósításának?
Akkor, ha a struktúra *az összes olyan komponenst definiálja és az összes olyan típusdefiníciót kielégíti*, amelyeket a szignatúra elvár. Másszóval definiálja a szignatúrában megadottal
 - ekvivalens típusú *kivételkomponenseket*,
 - kompatibilis (azaz legalább annyira általános) típusú *értékkomponenseket*,
 - azonos aritású (paraméterszámú) és – ha definiálja – ekvivalens definíciójú *típuskomponenseket*.
- Csakhogy egy struktúra a szignatúrájához képest, többek között
 - *több komponenst* definiálhat;
 - *általánosabb típusú értékeket* definiálhat (ami az újrafelhasználást segíti elő);
 - *datatype-deklarációt* használhat *type-deklaráció* helyett, *értékkonstruktort* definiálhat *érték* helyett (ami az adatabsztrakciót teszi lehetővé);
 - a *deklarációk sorrendje* tetszőleges lehet (ami növeli a flexibilitást).
- A feltett kérdésre ezért pontosabb a következő válasz:
Egy struktúra akkor és csak akkor tekinthető egy szignatúra megvalósításának, ha a struktúra ún. *törzsszignatúrája* illeszkedik az adott szignatúrára.

Törzstípus, törzsszignatúra

- Egy érték *törzstípusa* (principal type) az adott értékhez rendelhető *legáltalánosabb* típus.
- Minden jóldefiniált értéknek van törzstípusa.
Pl. ha $\text{fun } I \ x = x$, akkor $I : 'a \rightarrow 'a$.
- Egy struktúra *törzsszignatúrája* (principal signature) a komponenseihez rendelhető *legszelektívusabb* leírás.
- Minden jóldefiniált struktúrának van törzsszignatúrája.
- A típusellenőrzéshez elegendő a törzsszignatúra ismerete, a struktúrát magát nem kell vizsgálni.
- Egy struktúra törzsszignatúrája a következőképpen állítható elő: ha a deklaráció
 - $\text{type } (tyvar_1, \dots, tyvar_n) \ tycon = typ$ alakú, akkor a törzsszignatúra az ezzel azonos specifikációt tartalmazza;
 - $\text{datatype } (tyvar_1, \dots, tyvar_n) \ tycon = con_1 \text{ of } typ_1 \mid \dots \mid con_k \text{ of } typ_k$ alakú, akkor a törzsszignatúra az ezzel azonos specifikációt tartalmazza;
 - $\text{exception } id \text{ of } typ$ alakú, akkor a törzsszignatúra az ezzel azonos specifikációt tartalmazza;
 - $\text{val } id = exp$ alakú, akkor a törzsszignatúra a $\text{val } id : typ$ specifikációt tartalmazza, ahol typ az exp kifejezés törzstípusa.

Szignatúra-illeszkedés (folyt.)

- Egy *szignatúra-jelölt* (candidate signature) akkor és csak akkor illeszkedik egy *célszignatúrára* (target signature), ha az összes olyan komponenst és típusdefiniációt tartalmazza, amelyet a *célszignatúra* specifikál.
- Pontosabban, a szignatúra-jelöltnek tartalmaznia kell a célszignatúra
 - összes típuskonstruktorát, mégpedig azonos aritással (paraméterszámmal) és – ha definiálja – ekvivalens definícióval;
 - összes datatype-deklarációját, mégpedig úgy, hogy az adatkonstruktoroknak ekvivalens típusúaknak kell lenniük;
 - összes exception deklarációját, mégpedig úgy, hogy az argumentumaiknak, ha vannak, ekvivalens típusúaknak kell lenniük;
 - minden értékdeklarációját, mégpedig úgy, hogy a típusuknak legalább annyira általánosnak kell lenniük, mint a célszignatúrában.
- A szignatúra-jelöltnek a célszignatúránál lehet több komponense, és több típusdefiniációt tartalmazhat, de nem lehet benne kevesebb egyikből sem.
- A szignatúra-jelölt a célszignatúra *gyengítése*, mivel a célszignatúra összes tulajdonsága igaz a szignatúra-jelöltre is.

Szignatúra-illeszkedés: QUEUE, QUEUE_WITH_EMPTY, QUEUE_AS_LISTS

- ```

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

signature QUEUE_WITH_EMPTY =
sig include QUEUE
 val is_empty : 'a queue -> bool
end

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

```
- QUEUE\_WITH\_EMPTY illeszkedik QUEUE-ra, mert kielégíti QUEUE összes elvárását. QUEUE azonban a hiányzó is\_empty miatt nem illeszkedik QUEUE\_WITH\_EMPTY-re.
  - QUEUE\_AS\_LISTS illeszkedik QUEUE-ra, csak abban különbözik tőle, hogy 'a queue-t specifikálja. QUEUE azonban nem illeszkedik QUEUE\_AS\_LISTS-re, mert a QUEUE-beli 'a queue nem ekvivalens 'a list \* 'a list-tel.

## Szignatúra-illeszkedés: QUEUE, QUEUE\_AS\_LIST

---

```

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

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

```

- Úgy vélhetjük, hogy `QUEUE_AS_LIST` nem illeszkedik `QUEUE`-ra, annyira különbözik tőle.
- Csakhogy az előzővel ekvivalens az alábbi definíció:

```
signature QUEUE_AS_LIST =
 QUEUE where type 'a queue = 'a list
```

és az utóbbi nyilvánvalóan illeszkedik `QUEUE`-ra.

## Szignatúra-illeszkedés: MERGEABLE\_QUEUE, MERGEABLE\_INT\_QUEUE

---

- Említettük, hogy a szignatúra-jelöltben az értékek típusa általánosabb lehet, mint a célszignatúrában.
- A szignatúra-illeszkedés együtt járhat azzal, hogy a polimorf típusokat konkrét típusokra cseréljük.

```

signature MERGEABLE_QUEUE =
sig
 include QUEUE
 val merge : 'a queue * 'a queue -> 'a queue
end

signature MERGEABLE_INT_QUEUE =
sig
 include QUEUE
 val merge : int queue * int queue -> int queue
end

```

- A `MERGEABLE_QUEUE` szignatúra-jelölt illeszkedik a `MERGEABLE_INT_QUEUE` célszignatúrára, mert az előbbiben specifikált polimorf `merge` függvény típusát leíró típuskifejezés típusváltozója leköthető az `int` típusal.

## Szignatúra-illeszkedés: RBT\_DT, RBT

---

```

signature RBT_DT =
 sig datatype 'a rbt = Empty
 | Red of 'a rbt * 'a * 'a rbt
 | Black of 'a rbt * 'a * 'a rbt
 end

signature RBT =
 sig type 'a rbt
 val Empty : 'a rbt
 val Red : 'a rbt * 'a * 'a rbt -> 'a rbt
 end

```

- Az RBT\_DT szignatúra-jelölt illeszkedik az RBT célszignatúrára, mert az RBT\_DT-ben a datatype deklarációval specifikált típus és adatkonstruktorai illeszkednek az RBT-ben specifikált 'a rbt absztrakt típusra és a két értékspecifikációra (Empty és Red). Fordítva nem igaz.
- RBT\_DT ugyanis a következő típust, ill. adatkonstruktorokat specifikálja:

```

type 'a rbt
con 'a Empty : 'a rbt
con 'a Red : 'a rbt * 'a * 'a rbt -> 'a rbt
con 'a Black : 'a rbt * 'a * 'a rbt -> 'a rbt

```

## Szignatúra-illeszkedés (folyt.)

---

- Most már még pontosabban válaszolhatunk a kérdésre:
- Mikor tekinthető egy *struktúra-jelölt* egy *célszignatúra* megvalósításának?
- Akkor és csak akkor, ha a struktúra-jelölt *törzsszignatúrája* illeszkedik a célszignatúrára.
- Nyilvánvaló, hogy minden struktúra kielégíti a törzsszignatúráját (az illeszkedési reláció reflexív).
- Bármely szignatúra, amelyet egy struktúra megvalósít, *gyengébb* az adott struktúra törzsszignatúrájánál.
- A törzsszignatúra ezért a *legerősebb* szignatúra, amelyet egy struktúra megvalósíthat.



## Szignatúra-kötés

---

- *Szignatúra-kötéssel* (signature ascription) írjuk elő, hogy egy struktúra valósítson meg egy szignatúrát.
- A szignatúra-kötés *gyengíti* a struktúra szignatúráját az összes további felhasználás számára.
- Kétféle szignatúra-kötés van az SML-ben:
  - *átlátszó* vagy *leíró* (transparent, descriptive): a struktúra *látható szignatúrája* az adott struktúrában definiált típusokkal *bővített* célszignatúra lesz,
  - *áttetsző* vagy *korlátozó* (opaque, restrictive): a struktúra *látható szignatúrája* a célszignatúra lesz, bővítés nélkül.
- A szignatúra-kötés mindkét változata elrejt azokat a komponenseket, amelyek a célszignatúra nem specifikál.
- A moduláris programozás biztonsága megköveteli a típusinformációk gondos kezelését. A láthatóvá tételnek és az elrejtésnek egyformán fontos a szerepe.
- Az áttetsző szignatúra-kötéssel a típusinformáció láthatóságát korlátozzuk.
- Az átlátszó szignatúra-kötéssel a típusinformációt láthatóvá tesszük.

## Struktúra-deklaráció szignatúra-kötéssel

---

- Már láttuk a kétféle szignatúra-kötés használatát struktúra-deklarációkban:
  - *átlátszó*: `structure strid : sigexp = strex`
  - *áttetsző*: `structure strid :> sigexp = strex`
- A típusellenőrzés lépései szignatúrához kötött struktúra-deklaráció esetén a következők:
  - *strex* megvalósítja-e *sigexp*-et? Ennek eldöntéséhez a fordító
    - meghatározza *strex sigexp<sub>0</sub>* törzsszignatúráját, és megpróbálja illeszteni a *sigexp* célszignatúrára; valamint
    - előállítja a bővített *sigexp'* szignatúrát úgy, hogy *sigexp*-et bővíti a *sigexp<sub>0</sub>*-ban lévő típusdeklarációkkal;
  - a struktúranévhez köti a szignatúrát a kötés előírt módja szerint: a struktúra látható szignatúrája
    - *átlátszó* szignatúra-kötés esetén *sigexp'*,
    - *áttetsző* szignatúra-kötés esetén *sigexp* lesz.

## Struktúra-deklaráció szignatúra-kötéssel (folyt.)

- A szignatúrához kötött struktúra-deklaráció kiértékelését a fordító így folytatja:
  - kiértékeli *strex*-et;
  - előállítja az eredményül kapott érték egy *nézetét* úgy, hogy eldobja azokat az értékeket, amelyeket a *sigexp* célszignatúra nem tartalmaz;
  - a *strid* nevet ehhez a nézethez köti.
- A leírtakból is kitűnik, hogy az átlátszó szignatúra-kötés az áttetsző szignatúra-kötés *speciális esete*: a bővített szignatúrát a programozó maga is előállíthatná (technikai nehézségektől eltekintve, ui. néha nem férhet hozzá a szükséges információhoz).
- Az *áttetsző szignatúra-kötés* legfontosabb célja az adatabsztrakció elősegítése. Tekintsük a példát a következő dián!
- Az áttetsző szignatúra-kötés garantálja 'a `Queue_as_lists.queue` *absztrakt* voltát, így *kizárólag* az `empty`, `insert` és `remove` műveleteket lehet alkalmazni ilyen típusú értékekre.
- A programozó *nem használhatja ki*, hogy most a 'a `Queue_as_lists.queue` típust listákból álló párral valósítjuk meg.
- Ezért a szignatúrát megvalósító struktúra szabadon, a többi programrész konzisztenciájának megsértése nélkül módosítható.

## Struktúra-deklaráció áttetsző szignatúra-kötéssel: `QUEUE`, `Queue_as_lists`

- Az elmondottak illusztrálására nézzük a már látott példát:

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

## Struktúra-deklaráció áttetsző szignatúra-kötéssel (folyt.)

---

- A típusinformáció elrejtése a reprezentáció (ábrázolás) invariánsait is elszigeteli az absztrakció megvalósításától.
  - Az 'a Queue\_as\_lists.queue típust egy olyan absztrakt gép állapotípusának tekinthetjük, amelynek csak három parancs adható: empty (amely a kezdőállapotot hozza létre), insert és remove.
  - A Queue\_as\_lists struktúrán belül invariáns állításokkal jellemezhetjük az absztrakt gép belső állapotát.
  - Az adatabsztrakció elegáns eljárást nyújt az invariáns állítások alkalmazásához; az *assume-ensure* vagy *rely-guarantee* néven ismert eljárásához két követelményt kell kielégíteni:
    - minden inicializáló parancsnak *garantálnia kell* az invariáns teljesülését a végrehajtása után;
    - minden állapotmódosító parancs *felteheti*, hogy az invariáns teljesül a parancs végrehajtásának kezdetén, és minden ilyen parancsnak *garantálnia kell* az invariáns teljesülését a végrehajtása után.
  - Teljes indukcióval belátható, hogy az invariáns állítás az összes állapotra teljesül, azaz valóban invariáns!

## Struktúra-deklaráció áttetsző szignatúra-kötéssel: prioritási sor

---

- Olyan absztrakt prioritásisor-típust akarunk létrehozni, amely tetszőleges típusú elemekből állhat.
- A műveletek (függvények) nem lehetnek politípusúak, mert az elemek relatív prioritását összehasonlítással tudjuk megállapítani. Ezt a függőséget fejezi ki az alábbi szignatúra:

```
signature PQ =
sig
 type elt
 val lt : elt * elt -> bool
 type queue
 exception Empty
 val empty : queue
 val insert : elt * queue -> queue
 val remove : queue -> elt * queue
end
```

- Egy lehetséges megvalósítás vázlatát mutatja a következő példa, ahol az elemek `string` típusúak.

## Struktúra-deklaráció áttetsző szignatúra-kötéssel: prioritási sor (folyt.)

- A megvalósítástól független absztrakt típus *áttetsző szignatúrát* igényel:

```
structure PrioQueue :> PQ =
struct
 type elt = string
 val lt : string * string -> bool = (op <)
 type queue = ...
end
```

- Csakhogy így `PrioQueue.queue` mellett `PrioQueue.elt` is absztrakt típus lett, így nem tudunk `PrioQueue.elt` típusú értéket létrehozni, és pl. nem hívhatjuk a `PrioQueue.insert` függvényt. Ezért `PrioQueue.elt`-nek nem kellene absztrakt típusnak lennie.
- Egy lehetséges megoldás az, hogy a PQ szignatúrát bővítjük, és a bővített szignatúrát kötjük a struktúrához:

```
signature STRING_PQ = PQ where type elt = string
structure PrioQueue :> STRING_PQ = ...
```

vagy

```
structure PrioQueue :> PQ where type elt = string = ...
```

- A tanulság: megfontolást igényel, hogy mely típusokat válasszuk absztraktnak, és melyeket ne.

## Struktúra-deklaráció átlátszó szignatúra-kötéssel: `ORDERED`, `MyString`

- Átlátszó szignatúra-kötéssel csökkenthető az explicit típusspecifikációk száma a szignatúrában. De az se jó, ha túl sokat használjuk, ui. a típusinformációk láthatóvá tételével csökken a modulok függetlensége.
- Az átlátszó szignatúra-kötés tipikusan arra való, hogy egy struktúra *nézetét* állítsuk elő vele. E nézet célja, hogy elrejtse azokat a komponenseket, amelyek az adott szöveggörnyezetben feleslegesek, de ne rejtse el azokat a típusdefiníciókat, amelyekre szükség van.
- Az `ORDERED` szignatúra specifikálja a `t` típust és a `t` típusú értékekből álló párokra alkalmazható `lt` összehasonlító műveletet.

```
signature ORDERED =
sig
 type t
 val lt : t * t -> bool
end
```

- Az ilyen szignatúrát, mint láttuk, *csak átlátszóan* érdemes egy struktúrához kötni, különben nem tudnánk `t` típusú értékeket létrehozni.

## Struktúra-deklaráció átlátszó szignatúra-kötéssel: ORDERED, MyString (folyt.)

---

- Nézzük a következő példát:

```
structure MyString : ORDERED =
struct
 type t = string
 val clt = Char.<
 fun lt (s, t) = ... clt ...
end
```

- MyString a füzérek összehasonlítását karakterek összehasonlítására vezeti vissza, clt-t elrejti. String.t a külvilág számára is ekvivalens string-gel, bár ez az ORDERED szignatúrából nem látszik: MyString tényleges, látható szignatúrája ugyanis:

```
ORDERED where type t = string
```

- Arra is érdemes átlátszó szignatúra-kötést használni, hogy dokumentáljuk egy típus jelentését (anélkül, hogy absztrakttá tennénk).

## Struktúra-deklaráció átlátszó szignatúra-kötéssel: ORDERED, IntLt, IntDiv

---

- Tegyük föl, hogy egészeket kétféle alapon akarunk összehasonlítani, de nem akarjuk elrejteni, hogy egészekekről van szó:
- Összehasonlítás aritmetikai alapon

```
structure IntLt : ORDERED =
struct
 type t = int
 val lt = (op <)
end
```

- Összehasonlítás oszthatóság alapján

```
structure IntDiv : ORDERED =
struct
 type t = int
 fun lt (m, n) = (n mod m = 0)
end
```

- Mind IntLt.t, mind IntDiv.t ekvivalens int-tel.

## Átlátszóság, áttetszőség, függőség

---

- Az átlátszó szignatúra-kötés a típuslevezetéshez hasonlóan megkönnyíti a programozó dolgát: kevesebbet kell írnia. *De ára van:*
- átlátszó szignatúra-kötés esetén a szignatúra önmagában nem fordítható le, csak a struktúrával együtt: csak így állítható elő a struktúra tényleges, *látható* szignatúrája.
- Vagyis az összes olyan programrész, amely e struktúra látható szignatúrájára hivatkozik, *függ e struktúra megvalósításától!*
- Amíg az átlátszó szignatúra-kötés függőséget okoz, az áttetsző szignatúra-kötés kiküszöböli a függőséget.
- Ha egy struktúrához áttetsző módon kötjük a szignatúrát, a rá hivatkozó programrészek megbízhatnak a szignatúrában (a struktúra látható szignatúrája ui. ekvivalens az áttetsző szignatúrával).
- A megvalósítástól való függés gátolja, nehezíti a modularitást. A modularitás célja ui. az, hogy elszigetelje egymástól az egyes programrészeket, csökkentse az egyes programrészek hatását a többire. Ekkor egymástól függetlenül legyenek fejleszthetők, módosíthatók. Minél kevésbé függenek egymástól a modulok, annál könnyebben rakhatók össze a végén egyetlen rendszerré.

## MODULOK HIERARCHIÁJA

---

## Modulok hierarchiája

---

- Egy nagy program architektúrája rendszerint nem lineáris, hanem faszerkezetű (hierarchikus). Az SML modulnyelv faszerkezetű modulrendszer leírását is lehetővé teszi.
  - A modulok egymásba skatulyázhatók; a beágyazott struktúrát *alstruktúrának* (substructure) nevezzük.
  - Egy struktúra más struktúra-deklarációkat tartalmazhat (akár átlátszó, akár áttetsző szignatúra-kötéssel).
  - Egy szignatúrában egy struktúra `structure strid : sigexp` alakban specifikálható (szignatúráról lévén szó, itt nincs különbség átlátszó és áttetsző kötés között).
  - Az alstruktúrákra a struktúrák típusellenőrzési és a kiértékelési szabályait rekurzív módon alkalmazza a fordítóprogram.
  - Ebben a részben arról lesz szó, hogyan lehet alstruktúrákkal kifejezni az egyes absztrakciók egymástól való függését.
- 
- A következő példák egy polimorf szótárat megvalósító programból valók.

---

Deklaratív programozás. BME VIK, 2003. tavaszi félév

(Funkcionális programozás)

Modulok hierarchiája FP-282

## Polimorf szótár `string` típusú keresési kulccsal

---

- Az első változatban a *keresési kulcs* `string` típusú. A szignatúra és lehetséges megvalósítása:

```
signature MY_STRING_DICT =
sig
 type 'a dict
 val empty : 'a dict
 val insert : 'a dict * string * 'a -> 'a dict
 val lookup : 'a dict * string -> 'a option
end

structure MyStringDict :> MY_STRING_DICT =
struct
 datatype 'a dict = Empty
 | Node of 'a dict * string * 'a * 'a dict
 val empty = Empty
 fun insert (d, k, v) = ...
 fun lookup (d, k) = ...
end
```

- A hiányzó függvénydefiníciók a füzérek lexikografikus összehasonlító műveleteit használják.

## Polimorf szótár *int* típusú keresési kulccsal

---

- A második változatban a *keresési kulcs* *int* típusú. A szignatúra és lehetséges megvalósítása:

```
signature MY_INT_DICT =
sig
 type 'a dict
 val empty : 'a dict
 val insert : 'a dict * int * 'a -> 'a dict
 val lookup : 'a dict * int -> 'a option
end

structure MyIntDict :> MY_INT_DICT =
struct
 datatype 'a dict = Empty
 | Node of 'a dict * int * 'a * 'a dict
 val empty = Empty
 fun insert (d, k, v) = ...
 fun lookup (d, k) = ...
end
```

- A hiányzó függvénydefiníciók az egészek aritmetikai összehasonlító műveleteit használják.

## Polimorf szótár *absztrakt* típusú keresési kulccsal

---

- A két változat, a típustól és az összehasonlító műveletektől eltekintve, azonos.
- A harmadik változatban a *keresési kulcs absztrakt* típusú. A *generikus* szignatúra és két leszármazottja (példánya, instanciája):

```
signature MY_GEN_DICT =
sig
 type key
 type 'a dict
 val empty : 'a dict
 val insert : 'a dict * key * 'a -> 'a dict
 val lookup : 'a dict * key -> 'a option
end

signature MY_STRING_DICT =
 MY_GEN_DICT where type key = string

signature MY_INT_DICT =
 MY_GEN_DICT where type key = int
```



## Polimorf szótár `string` típusú keresési kulccsal (folyt.)

---

- A szignatúra egy megvalósítása `string` típusú kulcsokra:

```
structure MyStringDict :> MY_STRING_DICT =
struct
 type key = string
 datatype 'a dict = Empty
 | Node of 'a dict * key * 'a * 'a dict
 val empty = Empty
 fun insert (None, k, v) = Node (Empty, k, v, Empty)
 fun lookup (Empty, _) = NONE
 | lookup (Node (dl, l, v, dr), k) =
 if k < l then (* string comparison *)
 lookup (dl, k)
 else if k > l then (* string comparison *)
 lookup (dr, k)
 else
 SOME v
end
```

- A `MY_INT_DICT` szignatúrájú `MyIntDict` hasonlóan valósítható meg.

## Polimorf szótár `int` típusú kulccsal, *oszthatóságon alapuló összehasonlítással*

---

```
structure MyIntDivDict :> MY_INT_DICT =
struct
 type key = int
 datatype 'a dict = Empty
 | Node of 'a dict * key * 'a * 'a dict
 fun divides (k, l) = (l mod k = 0)
 val empty = Empty
 fun insert (None, k, v) = Node (Empty, k, v, Empty)
 fun lookup (Empty, _) = NONE
 | lookup (Node (dl, l, v, dr), k) =
 if divides (k, l) then (* divisibility test *)
 lookup (dl, k)
 else if divides (l, k) then (* divisibility test *)
 lookup (dr, k)
 else
 SOME v
end
```

- Függetlenítsük a megvalósítást a keresési kulcs típusától és az összehasonlító műveletektől!

## Egy rendezett absztrakt típus és néhány megvalósítása

- A `t` típus és két összehasonlító művelet

```
signature ORDERED =
sig
 type t
 val lt : t * t -> bool
 val eq : t * t -> bool
end
```

- Füzetek lexikografikus összehasonlítása

```
structure LexString : ORDERED =
struct
 type t = string
 val lt = (op <)
 val eq = (op =)
end
```

- Egészek aritmetikai összehasonlítása

```
structure LessInt : ORDERED =
struct
 type t = int
 val lt = (op <)
 val eq = (op =)
end
```

- Egészek oszthatóságon alapuló összehasonlítása

```
structure DivInt : ORDERED =
struct
 type t = int
 fun lt (m, n) = (n mod m = 0)
 fun eq (m, n) = lt (m, n)
 andalso lt (n, m)
end
```

- Ezekben a példákban indokolt az átlátszó szignatúra-kötés alkalmazása.

## Polimorf szótár generikus szignatúrája

- A `DICT` szignatúra „paramétere” az `ORDERED` szignatúrájú `Key` absztrakt keresési kulcs (a `DICT` szignatúra örökli a keresési kulcs `ORDERED` szignatúráját!):

```
signature DICT =
sig
 structure Key : ORDERED
 type 'a dict
 val empty : 'a dict
 val insert : 'a dict * Key.t * 'a -> 'a dict
 val lookup : 'a dict * Key.t -> 'a option
end
```

- A szignatúra két specializált változata segíti az absztrakciót:

```
signature STRING_DICT =
 DICT where type Key.t = string

signature INT_DICT =
 DICT where type Key.t = int
```

## Polimorf szótár: a specializált szignatúra megvalósítása string kulccsal

---

```

structure StringDict :> STRING_DICT =
struct
 structure Key : ORDERED = LexString
 datatype 'a dict = Empty
 | Node of 'a dict * Key.t * 'a * 'a dict
 val empty = Empty
 fun insert (None, k, v) = Node (Empty, k, v, Empty)
 fun lookup (Empty, _) = NONE
 | lookup (Node (dl, l, v, dr), k) =
 if Key.lt (k, l) then
 lookup (dl, k)
 else if Key.lt (l, k) then
 lookup (dr, k)
 else
 SOME v
end

```

(Félkövér szedéssel e változat és a következő két változat közötti **különbséget** emeljük ki.)

## Polimorf szótár: a specializált szignatúra megvalósítása int kulccsal (1. változat)

---

```

structure LessIntDict :> INT_DICT =
struct
 structure Key : ORDERED = LessInt
 datatype 'a dict = Empty
 | Node of 'a dict * Key.t * 'a * 'a dict
 val empty = Empty
 fun insert (None, k, v) = Node (Empty, k, v, Empty)
 fun lookup (Empty, _) = NONE
 | lookup (Node (dl, l, v, dr), k) =
 if Key.lt (k, l) then
 lookup (dl, k)
 else if Key.lt (l, k) then
 lookup (dr, k)
 else
 SOME v
end

```

## Polimorf szótár: a specializált szignatúra megvalósítása `int` kulccsal (2. változat)

---

```
structure DivIntDict :> INT_DICT =
struct
 structure Key : ORDERED = DivInt
 datatype 'a dict = Empty
 | Node of 'a dict * Key.t * 'a * 'a dict
 val empty = Empty
 fun insert (None, k, v) = Node (Empty, k, v, Empty)
 fun lookup (Empty, _) = NONE
 | lookup (Node (dl, l, v, dr), k) =
 if Key.lt (k, l) then
 lookup (dl, k)
 else if Key.lt (l, k) then
 lookup (dr, k)
 else
 SOME v
end
```

A funktorok tárgyalásakor látni fogjuk, hogyan írhatjuk meg e három struktúra közös generikus (azaz paraméterezhető) változatát.

## Paraméterezzhető, más néven generikus modulok

- Az újrafelhasználhatóságot segíti elő a *paraméterezzhető*, más néven *generikus* modul, azáltal hogy a megvalósítás egy vagy több elemét specifikálatlanul hagyja. A specifikálatlan elemek specifikálása a modul egy *példányát* hozza létre. A közös részt *csak egyszer* kell megírni.
- Az SML-ben az ilyen generikus modult *funktornak* (functor) nevezik. A funktornak struktúra a paramétere is, az eredménye is. A funktor egy *példányát* úgy hozzuk létre, hogy alkalmazzuk egy (létező) struktúrára.
- A *funktordeklarációnak* (vagy *funktorkötésnek*) két változata van, az *átlátszó*:

```
functor funid (decs) : sigexp = strexp
```

- és az *áttetsző*:

```
functor funid (decs) :> sigexp = strexp
```

- A funktor típusának ellenőrzéséhez a fordító megvizsgálja, hogy a funktor törzse megfelel-e a szignatúra-kötés által előírt szignatúrának, feltéve hogy a funktor paramétereinek megfelelő a szignatúrája.
- Mint tudjuk, az *áttetsző* szignatúra-kötés a megadott szignatúrát eredményezi, az *átlátszó* szignatúra-kötés pedig ennek a törzsszignatúra szerinti típusokkal bővített változata.

## Példa: polimorf szótár megvalósítása funktorral

- Egy korábbi példánk olyan polimorf szótár volt, ahol a keresési kulcsot és a rajta végrehajtható műveleteket egy *alstruktúra* specifikálta. A félkövér szedés a változatok közötti eltérésre utal.

```
structure StringDict :> DICT where type Key.t = string =
struct
 structure Key : ORDERED = LexString
 datatype 'a dict = Empty
 | Node of 'a dict * Key.t * 'a * 'a dict
 val empty = Empty
 fun insert (None, k, v) = Node (Empty, k, v, Empty)
 fun lookup (Empty, _) = NONE
 | lookup (Node (dl, l, v, dr), k) =
 if Key.lt (k, l) then lookup (dl, k)
 else if Key.lt (l, k) then lookup (dr, k)
 else SOME v
end
```

- Látható, hogy különbség csak a keresési kulcs típusában és a rajta végrehajtható műveletekben van; mindezt az ORDERED szignatúra specifikálja.

## Példa: polimorf szótár megvalósítása funktorral (DictFun)

- Az alstruktúrát, ha funktort deklarálunk, paraméterként adhatjuk át. A struktúradeklaráció és a funktordeklaráció közötti különbségeket most is félkövér szedéssel emeljük ki.

```

functor DictFun (structure K : ORDERED) :>
 DICT where type Key.t = K.t =
struct
 structure Key : ORDERED = K
 datatype 'a dict = Empty
 | Node of 'a dict * Key.t * 'a * 'a dict
 val empty = Empty
 fun insert (None, k, v) = Node (Empty, k, v, Empty)
 fun lookup (Empty, _) = NONE
 | lookup (Node (dl, l, v, dr), k) =
 if Key.lt(k, l) then lookup (dl, k)
 else if Key.lt(l, k) then lookup (dr, k)
 else SOME v
end

```

- A DICT where type Key.t = K.t szignatúra az 'a dict típus absztrakt voltát megőrzi. A Key.lt összehasonlító műveletet a paraméterként átadott struktúra valósítja meg.

## Funktoralkalmazás szignatúrája, funktor generativitása, ill. applikativitása

- A funktoralkalmazás *funid (binds)* alakú kifejezés, ahol *binds* a funktorargumentumok kötésének egy sorozata.
- Egy *funktoralkalmazás szignatúrája* a következő eljárással határozható meg. Feltesszük, hogy ismerjük a funktorparaméterek szignatúráját, valamint a funktor látható szignatúráját (a megadottat áttetsző, a bővítettet átlátszó szignatúrakötés esetén).
  - Minden argumentum szignatúráját illesztjük a funktor megfelelő paraméterének szignatúrájára. Ezzel minden argumentumra megkapjuk a paraméterszignatúrák egy bővített változatát.
  - Ha az eredmény szignatúrája hivatkozik a funktorparaméter valamely típuskomponensére, akkor a bővített paraméterszignatúrában lévő típusdefiníciónak meg kell jelennie az eredmény szignatúrájában.
- Az így előállított szignatúra átlátszó kötéssel kapcsolódik a funktoralkalmazáshoz. Ez azt jelenti, hogy ha a funktor eredményszignatúrája egy típust absztraktként specifikál, akkor e funktor minden alkalmazása e típusból egy új példányt hoz létre. Ezt a viselkedést a funktor *generativitásának* nevezzük. (Ezzel szemben a funktor *applikativitása* azt jelenti, hogy a funktor összes példánya megosztva „használja” ugyanazt az absztrakt típust.)

## Példa: polimorf szótár (LtIntDict, LexStringDict, DivIntDict)

---

- A szótár három változatát a DictFun funktor alkalmazásával könnyű előállítani:

```
structure LtIntDict = DictFun (structure K = LessInt)
structure LexStringDict = DictFun (structure K = LexString)
structure DivIntDict = DictFun (structure K = DivInt)
```

- Idézzük föl LexString, LessInt és DivInt egy-egy megvalósítását:

```
structure LexString : ORDERED =
struct type t = string
 val lt = (op <)
 val eq = (op =)
end

structure LessInt : ORDERED =
struct type t = int
 val lt = (op <)
 val eq = (op =)
end

structure DivInt : ORDERED =
struct type t = int
 fun lt (m, n) = (n mod m = 0)
 fun eq (m, n) = lt (m, n) andalso lt (n, m)
end
```

- Például LessInt bővített szignatúrája ez: ORDERED where type t = int.
- Ha a K paraméter aktuális értéke LessInt, akkor K.t és int ekvivalensek lesznek, és így DictFun aktuális szignatúrája ez: ORDERED where type Key.t = int.

## Lusta lista

---

- Olyan lista, amelynek a farka függvény, ezáltal késleltetjük a kiértékelését.
- Ily módon *végtelen listákat* hozhatunk létre.
- A lusta listának hátrányai, veszélyei is vannak, pl.
  - egy lusta lista bármely részét megjeleníthetjük, de sohasem az egészet;
  - két lusta lista elemeiből páronként képezhetünk egy harmadikat, de nem számíthatjuk ki egy lusta lista elemeinek az összegét, nem kereshetjük meg benne a legkisebbet, nem fordíthatjuk meg az elemek sorrendjét;
  - úgy kell rekurziót definiálnunk, hogy nincs alapeset;
  - egy program befejeződése helyett csak azt igazolhatjuk, hogy az eredmény tetszőleges véges része véges idő alatt előáll.
- A lusta listát sorozatnak (*sequence*) nevezzük, és a `seq` típusoperátort használjuk a létrehozására.

```
datatype 'a seq = Nil | Cons of 'a * (unit -> 'a seq)
```

## Lusta lista (folyt.)

---

- Egy sorozat fejét adja eredményül a `head` függvény; abortál, ha üres sorozatra alkalmazzuk.

```
(* head : 'a seq -> 'a
*)
fun head (Cons(x, _)) = x
```

- Egy sorozat farkát adja eredményül a `tail` függvény; abortál, ha üres sorozatra alkalmazzák.

```
(* tail : 'a seq -> 'a seq
*)
fun tail (Cons(_, xf)) = xf()
```

A sorozat farka `unit -> 'a seq` típusú *függvény*, erre illesztjük az `xf` mintát `tail` fejében; `tail` törzsében `xf`-et a `()` argumentumra kell alkalmazni.



## Lusta lista (folyt.)

---

Most további függvényeket definiálunk. `consq(x, xq)` az `x`-et berakja az `xq` sorozatba:

```
(* consq : 'a * 'a seq -> 'a seq
*)
fun consq (x, xq) = Cons(x, fn () => xq)
```

- Ha a `consq` függvényt alkalmazzuk, mondjuk, az  $(x, E)$  argumentumra, az SML a `consq(x, E)` kifejezést *nem lustán* értékeli ki, hiszen alapvetően mohó kiértékelésű.
- Ha  $E$  kiértékelésének eredményét `xq`-val jelöljük, akkor `consq(x, E)` kiértékelése a fenti definíció szerint `Cons(x, fn () => xq)`-t eredményez.
- A `consq`-beli `fn () => xq` függvény nem késlelteti a fark (a példában  $E$ ) kiértékelését `consq` alkalmazásakor.
- A lusta kiértékelés érdekében a híváskor is a `Cons(x, fn () => E)` alakot kell használnunk, `consq(x, E)` nem jó.
- Az explicit `fn () => E` alak késlelteti a kiértékelést: *szükség szerinti hivatkozást* valósít meg.

## Lusta lista (folyt.)

---

- Példaként a korábban megismert `from` és `take` függvények lusta változatait mutatjuk be.
- A `fromq k` sorozat egészek  $k$ -tól induló végtelen sorozata.

```
(* fromq : int -> int seq
*)
fun fromq k = Cons(k, fn () => fromq(k+1))
```

- `takeq(xq, n)` az `xq` sorozat első  $n$  eleméből képzett listát adja vissza:

```
(* takeq : 'a seq * int -> 'a list
*)
fun takeq (xq, 0) = []
 | takeq (Nil, n) = []
 | takeq (Cons(x, xf), n) = x :: takeq(xf(), n-1)
```

- Az `'a seq` típus nem egészen lusta kiértékelésű: ***egy nemüres sorozat fejét a futtatórendszer mindig feldolgozza.***

## Egyszerű függvények lusta listákra

---

- A kiszámíthatóság érdekében egy függvény eredményének tetszőleges véges része az argumentum véges részétől függhet csak.
- Amikor az eredményre szükség van, akkor ez az igény váltja ki az argumentum feldolgozását.
- Első példánkban egészeket egyesével emelünk négyzetre. Amikor szükség van rá, az eredmény farka (egy függvény) alkalmazza a `squareq` függvényt az argumentum farkára.

```
(* squareq : int seq -> int seq
*)
fun squareq Nil: int seq = Nil
 | squareq (Cons (x, xf)) = Cons(x * x, fn () => squareq(xf()))
```

- Két lusta lista hasonlóan adható össze.

```
(* addq : (int seq * int seq) -> int seq
*)
fun addq (Cons (x, xf), Cons(y, yf)) =
 Cons(x+y, fn () => addq(xf(), yf()))
 | addq _: int seq = Nil
```

## Egyszerű függvények lusta listákra (folyt.)

---

- Az `appendq` függvény addig nem nyúl `yq`-hoz, amíg `xq` ki nem ürül – vagyis csak akkor nyúl hozzá, ha `xq` véges. Véges sorozatot `consq`-val készíthetünk.

```
(* appendq : 'a seq * 'a seq -> 'a seq
*)
fun appendq (Nil, yq) = yq
 | appendq (Cons (x, xf), yq) =
 Cons(x, fn () => appendq (xf(), yq))
```

- Most érthetjük meg, hogy miért kellett a típusdefinícióban a `Nil` konstruktorállandót definiálni.

## Magasabb rendű függvények lusta listákra

---

- A map lusta változata:

```
(* mapq : ('a -> 'b) -> 'a seq -> 'b seq
*)
fun mapq f Nil = Nil
 | mapq f (Cons (x, xf)) = Cons(f x, fn () => mapq f (xf()))
```

- A filter lusta változata:

```
(* filterq : ('a -> bool) -> 'a seq -> 'a seq
*)
fun filterq p Nil = Nil
 | filterq p (Cons (x, xf)) =
 if p x
 then Cons(x, fn () => filterq p (xf()))
 else filterq p (xf())
```

## Magasabb rendű függvények lusta listákra (folyt.)

---

- squareq a korábban látottnál sokkal egyszerűbben definiálható mapq-val:

```
val squareq = mapq (fn i => i * i)
```

- Olyan számsorozatot állítunk elő, amelyben 50-nél nagyobb, 7-esre végződő egészek vannak:

```
filterq (fn n => n mod 10 = 7) (fromq 50)
```

- Az iterateq függvény – a fromq egy általánosítása – a következő sorozatot állítja elő:  
 $[x, f(x), f(f(x)), \dots, f^k(x), \dots]$ .

```
(* iterateq : ('a -> 'a) -> 'a -> 'a seq
*)
fun iterateq f x = Cons(x, fn () => iterateq f (f x))
```

- fromq-t iterateq-val így definiálhatjuk:

```
(* fromq : int -> int seq
*)
val fromq = iterateq (fn i => i+1)
```

## Prímszámok előállítás *eratoszteni* szitával

---

1. Vegyük az egészek 2-vel kezdődő sorozatát: (2, 3, 4, 5, 6, 7, ...).
2. Töröljük az összes 2-vel osztható számot: (3, 5, 7, 9, 11, ...).
3. Töröljük az összes 3-mal osztható számot: (5, 7, 11, 13, 17, 19, ...).
4. Töröljük az összes ...

- A sorozat első eleme mindig a következő prím. A sorozatban azok a számok maradnak benne, amelyek az eddig előállított prímeikkel nem oszthatók.

```
(* sift : int -> int seq -> int seq *)
fun sift p = filterq (fn n => n mod p <> 0)
```

- A `sift` a `p` argumentum többszöröseit törli egy lusta listából.
- A `sieve`-nek már csak ismételten alkalmaznia kell `sift`-et a megfelelő lusta listára. Mivel ez a lusta lista sohasem üres, nem kell az üres lusta listára illeszkedő változatot írunk.

```
(* sieve : int seq -> int seq *)
fun sieve Nil = Nil
 | sieve (Cons (p, nf)) = Cons(p, fn () => sieve(sift p (nf())));

takeq(sieve(fromq 2), 10)
```

## Álvéletlen számok

---

- Hagyományos álvéletlenszám-generátorok: olyan eljárások, amelyek egy *frissíthető változóban* tárolják a *seed* (mag) értéket – ebből állítják elő egy következő hívásnál a következő álvéletlen számot.
- Lusta listaként megvalósítva: a következő álvéletlen szám csak szükség esetén áll elő.

```
(* randseq : int -> real seq
*)
local val a = 16807.0 and m = 2147483647.0
 (* nextrandom : real -> real
 *)
 fun nextrandom seed =
 let val t = a * seed
 in t - real(floor(t/m)) * m
 end
in
 fun randseq s =
 mapq (fn x => x / m) (iterateq nextrandom (real s))
end
```

## Álvéletlen számok (folyt.)

- Ha a `nextrandom`-ot 1.0 és 21474836467.0 közötti `seed`-re alkalmazzuk, ugyanebbe a tartományba eső más értéket állít elő az  $a * \text{seed} \bmod m$  művelettel. (A valós számokat a túlsordulás elkerülésére használjuk.)
- A lusta lista előállítására `iterateq`-t `nextrandom`-ra és `seed` valós számmá alakított kezdőértékére alkalmazzuk. `mapq` gondoskodik arról, hogy a lusta listában minden értéket elosszunk `m`-mel, és így `randseq 0.0`-nál nem kisebb és `1.0`-nél kisebb értékeket adjon eredményül. Látható, hogy a lusta lista a megvalósítás részleteit szépen elrejt a felhasználó elől.
- Az előállított álvéletlen-számok `0.0`-nál nem kisebb és `1.0`-nél kisebb valós számok; `mapq`-val alakíthatjuk át őket 0 és 1 közötti egészekké:

```
mapq (floor o (fn x => 10.0 * x)) (randseq 1)
```

## Négyzetgyökvonás Newton-Raphson módszerrel

*Innentől az előadáson nem hangzott el, csak olvasmány, nem vizsgaanyag!*

- `nextapprox`  $x_k$ -ből  $x_{k+1}$ -et számítja ki az  $x_{k+1} = \frac{\frac{a}{x_k} + x_k}{2}$  képlet alapján.

```
(* nextapprox : real -> real -> real
*)
fun nextapprox a x = (a/x + x)/2.0
```

- A befejeződés megállapítására egyszerű tesztet írunk:

```
(* within : real -> real seq -> real
*)
fun within (eps: real) (Cons (x, xf)) =
 let val Cons (y, yf) = xf()
 in
 if abs (x-y) <= eps then y
 else within eps (Cons (y, yf))
 end
```

A `(Cons (y, yf))` és az `xf()` lusta lista ugyanaz: az `else`-ágban azért használjuk az elsőt, mert `xf()` meghívása költségesebb.

## Négyzetgyökvonás Newton-Raphson módszerrel (folyt.)

---

- Ezzel

```
(* qroot : real -> real
*)
fun qroot a = within 1E~6 (iterateq (nextapprox a) 1.0)
```

- A példában világosan különválasztjuk a leállásvizsgálatot (termination test) a következő jelölt előállításától.
- Most az abszolút különbséget ( $|x - y| < \varepsilon$ ) teszteljük, de vizsgálhatnánk pl. a relatív különbséget ( $|\frac{x}{y} - 1| < \varepsilon$ ) vagy az  $\frac{|x-y|}{\frac{|x|+|y|}{2}+1} < \varepsilon$  feltételt.
- A feladat többi része független attól, hogy milyen leállásvizsgálatot alkalmazunk, és így is kell megfogalmazni a megoldást.

## Négyzetgyökvonás Newton-Raphson módszerrel (folyt.)

---

- Írjunk függvényt a következő jelölt előállítására, és rejtjük el a részleteket:

```
(* approxq : real -> real seq
*)
fun approxq a =
 let (* nextapprox : real -> real
 *)
 fun nextapprox x = (a/x + x) / 2.0
 in iterateq nextapprox 1.0
 end
```

- Ezzel qroot egy „tisztább” változata:

```
(* qroot : real -> real
*)
val qroot = within 1E~6 o approxq
```

## Keresztszorzatokból álló lista

---

- Legyen  $xq$  és  $yc$  egy-egy sorozat. Képezzünk új sorozatot az  $(x_i, y_j)$  párokból, ahol  $x_i \in xq$  és  $y_j \in yc$ !
- Először hagyományos listákra oldjuk meg a feladatot `map` és `pair` alkalmazásával.
- $xs$  és  $ys$  egy-egy lista. Képezzünk listát az  $(x_i, y_j)$  párokból, ahol  $x_i \in xs$  és  $y_j \in ys$ !
- `map-et`, `pair-t` és `List.concat`-ot alkalmazva juthatunk el a keresett függvényhez.

```
(* pair : 'a -> 'b -> ('a * 'b)
*)
fun pair x y = (x, y)
```

- A `pair-t` a `map`-pel az  $ys$  lista elemeire alkalmazva olyan párokból álló listát kapunk eredményül, amelyben a párok első tagja a rögzített  $x$  érték, a második tagja pedig az  $ys$  egy-egy eleme.

```
map (pair x) ys
```

## Keresztszorzatokból álló lista (folyt.)

---

- Hogyan érhetjük el, hogy az  $x$  végigfusson az  $xs$  lista összes elemén? Az eddig szabad  $x$ -et kössük le egy függvény argumentumaként:

```
fn x => map (pair x) ys
```

majd alkalmazzuk újból a `map-et` erre a függvényre és  $xs$ -re:

```
map (fn x => map (pair x) ys) xs
```

- Listák listáját kapjuk eredményül, mert a belső `map` már listát adott vissza, amelynek minden eleméből újabb listát képeztünk a külső `map`-pel. `List.concat` elvégzi a szükséges simítást:

```
(* pairs : 'a list -> 'b list -> ('a * 'b) list
*)
fun pairs xs ys = List.concat (map (fn x => map (pair x) ys) xs)
```

## Keresztszorzatokból álló lusta lista

---

- A `pairss`-hez hasonlóan állíthatjuk elő párok lusta listájának lusta listáját:

```
(* pairqq : 'a seq -> 'b seq -> ('a * 'b) seq seq
*)
fun pairqq xq yq = mapq (fn x => mapq (pair x) yq) xq
```

- Az eredmény véges része kiíratható `takeqq`-val, amely a bal felső saroktól számított első `m` sorból és `n` oszlopból álló téglalapot jeleníti meg az `xqq` lusta listából:

```
(* 'a takeqq : 'a seq seq * (int * int) -> 'a list list
*)
fun takeqq (xqq, (m, n)) =
 map (fn yq => takeq(yq, n)) (takeq(xqq, m))
```

- Példa: olyan lusta lista, amelyben a párok első tagja az egymás után következő egészek 30-tól kezdve, második tagja pedig a prímszámok 2-től kezdve:

```
- pairqq (fromq 30) (sieve(fromq 2));
> val it = Cons (Cons ((30, 2), fn), fn) : (int * int) seq seq
```

## Keresztszorzatokból álló lusta lista (folyt.)

---

- `- takeqq(pairqq (fromq 30) (sieve(fromq 2)), (3, 5));`

```
> val it = [[(30, 2), ..., (30, 11)],
 [(31, 2), ..., (31, 11)],
 [(32, 2), ..., (32, 11)]] : (int * int) list list
```

- Ha ki akarjuk símítani a lusta listát, egy `List.concat`-hoz hasonló, lusta listákra alkalmazható függvénnyel nem megyünk semmire:

ha `xq` végtelen, `appendq (xq, yq) = xq`.

Azonban két lusta lista elemei páronként egymásba ékelhetők:

```
(* interleaveq : 'a seq * 'a seq -> 'a seq
*)
fun interleaveq (Nil, yq) = yq
 | interleaveq (Cons (x, xf), yq) =
 Cons(x, fn () => interleaveq(yq, xf()))
```

- `interleaveq` a rekurzív hívásban váltogatja a két lusta listát.

- `- takeq(interleaveq(fromq 0, fromq 50), 10);`  
`> val it = [0, 50, 1, 51, 2, 52, 3, 53, 4, 54] : int list`



## Keresztszorzatokból álló lusta lista (folyt.)

---

- `enumerate`: lusta listák lusta listájából egyetlen lusta listát állít elő. Legyen a kétszeres mélységű lusta lista feje `xq` és a farka `xqf`; alkalmazzuk `enumerate`-et rekurzívan `xqf`-re, majd az eredményt ékeljük `xq`-ba:

```
(* enumerate : 'a seq seq -> 'a seq
*)
fun enumerate Nil = Nil
 | enumerate (Cons (xq, xqf)) =
 interleaveq (xq, enumerate(xqf()))
```

- Ez a „megoldás” nem jó, mert a „végtelen” lusta lista miatt a rekurzió nem ér véget: az SML-ben, amely alapvetően mohó kiértékelésű, a rekurzív hívást késleltetni kell. Több esetet kell megkülönböztetnünk:

```
fun enumerate Nil = Nil
 | enumerate (Cons (Nil, xqf)) = enumerate (xqf())
 | enumerate (Cons (Cons (x, xf), xqf)) =
 Cons(x, fn () =>
 interleaveq(enumerate(xqf()), xf()))
```

## Keresztszorzatokból álló lusta lista (folyt.)

---

- Ha a bemenő lusta lista üres, készen vagyunk. Ha nem üres, meg kell vizsgálni a lusta lista fejét: ha ez üres, akkor folytatni kell a rekurzív hívást, ha nem üres, akkor az explicit `fn () => ...` függvénydefinícióval *késleltetni kell* a rekurziót.
- Példa: pozitív egészekből álló párok egy lusta listáját!

```
- val posintqq = pairqq (fromq 1) (fromq 1);
> val posintqq = Cons (Cons ((1, 1), fn), fn):(int * int) seq seq
- takeq(enumerate posintqq, 15);
> val it = [(1,1), (2,1), (1,2), (3,1), (1,3), (2,2),
 (1,4), (4,1), (1,5), (2,3), (1,6), (3,2),
 (1,7), (2,4), (1,8)] : (int * int) list
```

# MODULOK SZIMMETRIKUS ÖSSZEKAPCSOLÁSA

## Típusmegosztás specifikálása

*Innentől az előadáson nem hangzott el, csak olvasmány, nem vizsgaanyag!*

- Ebben a részben *modulok szimmetrikus összekapcsolásával* foglalkozunk.
- A különböző modulokban (akár azonos néven) specifikált absztrakt típusok mind különbözők. Általában ezt akarjuk. De nem mindig.
- A különböző modulokban specifikált típusok azonosságát az ún. *típusmegosztási előírással* (type sharing constraint) adhatjuk meg.
- A következő példák egy mértani elemeket megvalósító programból valók.
- Csupán két térbeli elemet valósítunk meg: a pontot és gömböt.

```
signature GEOMETRY =
sig
 structure Point : POINT
 structure Sphere : SPHERE
end
```

- A mértani elemek ábrázolását a vektorra és a pontra alapozzuk.

## Példa: mértani alapelemek ábrázolása (VECTOR, POINT)

---

- A VECTOR szignatúra egy vektor skalárral való szorzatát (`scale`), két vektor összegét (`add`) és skalárszorzatát (`dot`), továbbá a vektorösszeadás egységelemét (`zero`) specifikálja.

```
signature VECTOR =
sig
 type vector
 val zero : vector
 val scale : real * vector -> vector
 val add : vector * vector -> vector
 val dot : vector * vector -> real
end
```

- A POINT szignatúra egy pont eltolását egy vektor mentén (`translate`) és egy végpontjaival megadott vektor előállítását (`ray`) specifikálja.

```
signature POINT =
sig
 structure Vector : VECTOR
 type point
 val translate : point * Vector.vector -> point
 val ray : point * point -> Vector.vector
end
```

## Példa: mértani alapelemek ábrázolása (SPHERE)

---

- A gömböt a középpontjával és a sugarával adjuk meg.
- A gömböt létrehozó függvényt (`sphere`) az alábbi szignatúra specifikálja:

```
signature SPHERE =
sig
 structure Vector : VECTOR
 structure Point : POINT
 type sphere
 val sphere : Point.point * Vector.vector -> sphere
end
```

- Emlékeztető: a típusneveket és az értékneveket különböző névterek tárolják, ezért a `sphere` azonosító egyszerre használható típusnévként és értéknévként.
- Vegyük észre, hogy tér dimenziója nem része a specifikációnak!
- A dimenziót majd csak a modul megvalósításakor rögzítjük, ezzel elősegítjük a specifikáció *újrafelhasználását*.

## Különböző modulokban specifikált absztrakt típusok különböző volta

- Két- és háromdimenziós mértant így kezdődő struktúra-deklarációkkal valósíthatunk majd meg:

```
structure Geom2D :> GEOMETRY = ...
structure Geom3D :> GEOMETRY = ...
```

- Az *áttetsző* szignatúrakötésnek köszönhetően a két struktúrának *különböző* lesz a látható szignatúrája: a típusellenőrzés gondoskodik róla, hogy pl. a háromdimenziós térben ábrázolt `Geom3D.Sphere.sphere` középpontja ne lehessen a kétdimenziós térben ábrázolt `Geom2D.Point.point` pont.
- Ez jó dolog, növeli a programozás biztonságát.
- Sajnos, nemcsak `Geom2D` különbözik `Geom3D`-től, hanem pl. `Geom2D.Sphere.Vector` is különbözik `Geom2D.Point.Vector`-től!
- Ezért típushibát jelez a fordító a következő sor fordításakor (ahol `p` és `q` adott, `Geom2D.Point.point` típusú pontok):
 

```
Geom2D.Sphere.sphere (p, Geom2D.Point.ray (p, q))
```
- `Geom2D.Point.ray (p, q)` eredménye `Geom2D.Point.Vector.vector` típusú, `Geom2D.Sphere.sphere` ugyanakkor `Geom2D.Sphere.Vector.vector` típusú értéket vár. Ezt nyilvánvalóan nem akarjuk. Mi lehet az oka, hogyan küszöbölhetjük ki?

## Különböző modulokban specifikált absztrakt típusok megosztása (SPHERE)

- Az ok az, hogy a szignatúrákban specifikáltuk azokat az alstruktúrákat, amelyektől e szignatúrák függenek, így a pont-absztrakciót *két*, a vektor-absztrakciót *három példányban* hoztuk létre!
- Mivel áttetsző szignatúrakötést használunk, a látható szignatúráik mind különböznek!
- *Általában ezt akarjuk, néha nem.* Az SML-ben előírhatjuk, hogy két alstruktúra valamely absztrakt típusa legyen azonos. Erre való a *típusmegosztási előírás* (type sharing constraint).
- SPHERE módosított specifikációja (a módosítást **félkövér szedés** jelöli):

```
signature SPHERE =
sig
 structure Vector : VECTOR
 structure Point : POINT
 sharing type Point.Vector.vector = Vector.vector
 type sphere
 val sphere : Point.point * Vector.vector -> sphere
end
```

- A *típusmegosztási előírás* egy változatával, a *struktúramegosztási előírással* (structure sharing constraint) előírhatjuk, hogy két alstruktúra *összes* absztrakt típusa azonos legyen.

```
... sharing Point.Vector = Vector ...
```

## Különböző modulokban specifikált absztrakt típusok megosztása (GEOMETRY)

- GEOMETRY módosított specifikációja (két változatban, a módosítást **félkövér szedés** jelöli):

```
signature GEOMETRY =
sig
 structure Point : POINT
 structure Sphere : SPHERE
 sharing type Point.point = Sphere.Point.point
 sharing type Point.Vector.vector = Sphere.Vector.vector
end

... sharing Point = Sphere.Point
 sharing Point.Vector = Sphere.Vector ...
```

- A megosztási előírás tehát garantálja, hogy
  - a típus egyenletek mindig teljesüljenek, amikor a GEOMETRY szignatúrát és összes komponensét megvalósítjuk;
  - a megosztási előírás által érintett összes absztrakt típus azonos legyen.
- VECTOR-t és POINT-ot *egy példányban* valósítjuk meg, és e példányokat *újra felhasználjuk* a magasabb szintű absztrakció során (ld. a következő fóliákon).

## Példa: VECTOR, POINT, SPHERE és GEOMETRY egy 3D-s megvalósítása

- `structure Vector3D : VECTOR = ...`
- `structure Point3D : POINT =`  
 `struct`  
 `structure Vector : VECTOR = Vector3D`  
 `...`  
 `end`
- `structure Sphere3D : SPHERE =`  
 `struct`  
 `structure Vector : VECTOR = Vector3D`  
 `structure Point : POINT = Point3D`  
 `...`  
 `end`
- `structure Geom3D :> GEOMETRY =`  
 `struct`  
 `structure Point = Point3D`  
 `structure Sphere = Sphere3D`  
 `end`

- Fordítási idejű típushibához vezetne egyes 2D-s elemek alkalmazása a 3D-s megvalósításban.  
Példa:

```
... structure Sphere = Sphere2D ...
```

## A típusmegosztás elkerülése, a megosztási előírások számának csökkentése

- Fölvethető a kérdés, hogy a típusmegosztás elkerülhető-e a pont- és a vektor-absztrakció következtében létrejött példányszámok csökkentésével.
- A válasz: igen; azon az áron, hogy az egész programstruktúrát erőszakosan megváltoztatjuk.
- Első lépésként SPHERE-ben `Vector.vector-t Point.Vector.vector-ra` cseréljük:

```
signature SPHERE =
sig
 structure Point : POINT
 type sphere
 val sphere : Point.point * Point.Vector.vector -> sphere
end
```

- Ezzel GEOMETRY-ben `sharing Point.Vector = Sphere.Vector` feleslegessé vált, a megosztási előírások száma eggyel csökkent:

```
signature GEOMETRY =
sig
 structure Point : POINT
 structure Sphere : SPHERE
 sharing Point = Sphere.Point
end
```

## A típusmegosztás elkerülése, a megosztási előírások számának csökkentése (folyt.)

- Ha a `Point` alstruktúra specifikációját is sikerülne feleslegessé tenni SPHERE-ben, egyáltalán nem kellene megosztási előírás. Ekkor ez maradna SPHERE-ből:

```
signature SPHERE =
sig
 type sphere
 val sphere : Point.point * Point.Vector.vector -> sphere
end
```

- Most SPHERE-ből hiányzik a `Point` specifikálása. Ha `Point` már definiálva van, akkor `Point` lefordítható.
- Csakhogy ettől kezdve a SPHERE szignatúra a `Point` struktúrától, azaz a POINT szignatúra *egy megvalósításától* függ. Pl. a 2D-s megvalósítástól, ami által a szignatúra dimenziótól való függetlensége megszűnt, az absztrakció csorbát szenvedett.
- Ez az út tehát járhatatlan, de semmiképpen nem javasolható.
- Az eset hasonló ahhoz, amikor egy függvénynek nem paraméterként, hanem globálisként adunk át egy értéket.
- A `Point` struktúra a SPHERE szignatúra *paraméterének* tekinthető az ismertetett értelemben.

## A típusmegosztás elkerülése, a megosztási előírások számának csökkentése (folyt.)

- Egyetlen lehetőség maradt, hogy megszabaduljunk a megosztási előírásoktól, az, hogy a GEOMETRY szignatúrából töröljük a Point alstruktúrát. Ez lehetséges éppen, csak nem megoldás, mert csupán elhalasztja a problémát, de nem oldja meg a következők miatt. Pl. egy mértani elemeket megvalósító igazi programban több olyan elem van, amely a pont-fogalomra épít. Ezeknek feltétlenül szükségük lesz a megosztási előírásra.
- Lássunk egy további példát ennek alátámasztására, a SEMI\_SPACE specifikációját. A side predikátummal vizsgálható meg, hogy *egy adott pont a tér melyik felében van* – feltéve, hogy ez lehetséges: ezért választjuk a bool option típusú eredményt.

```
signature SEMI_SPACE =
sig
 structure Point : POINT
 type semispace
 val side : Point.point * semispace -> bool option
end
```

- SEMI\_SPACE-ből, SPHERE-hez hasonlóan, nem törölhetjük a Point alstruktúra specifikációját, ezért Point-ből mégiscsak két újabb példányt hozunk létre, azonosságukat pedig majd megosztási előírással kell kifejeznünk a GEOMETRY szignatúra új változatában.

## A típusmegosztás elkerülése, a megosztási előírások számának csökkentése (folyt.)

- A Point alstruktúrák azonosságát tehát megosztási előírással kell kifejeznünk EXTENDED\_GEOMETRY-ben, GEOMETRY kiegészített, de a Point alstruktúra nélküli változatában:

```
signature EXTENDED_GEOMETRY =
sig
 structure Sphere : SPHERE
 structure SemiSpace : SEMI_SPACE
 sharing Sphere.Point = SemiSpace.Point
end
```

- Amiről itt szó van, nem más, mint a moduláris programozás alapvető belső ellentmondása.
  - Egyrészt el akarjuk *szigetelni* egymástól a modulokat, hogy egymástól függetlenül legyenek kezelhetők, és ne befolyásolja az egyik modul megváltoztatása a többit. (A globális értékektől, változóktól való függés is az elszigetelés ellen hat!)
  - Másrészt a modulok kombinálásával programokat hozunk létre; ekkor a különböző modulok egyes pramelemeinek azonosságát (SML: megosztási előírásokkal) ki kell kötnünk.
- A megosztási előírás olyan *eszköz*, amely valaminek a bekövetkezése (ti. a specifikáció rögzítése) után hoz létre kapcsolatot a különböző absztrakciók között, és teremti meg az egész program koherenciáját. Ez a megközelítés az SML – más nyelvekben ismeretlen – sajátossága.

## Funktorok és a típusmegosztás specifikálása egy példán

---

- Korábban specifikáltuk a GEOMETRY szignatúrát és komponenseit.
- Mivel a célunk többféle (2D és 3D) megvalósításuk, érdemes őket funktorként definiálnunk.

```

functor PointFun
 (structure V : VECTOR) : POINT = ...

functor SphereFun
 (structure V : VECTOR
 structure P : POINT) : SPHERE =
struct
 structure Vector = V
 structure Point = P
 ...
end

functor GeomFun
 (structure P : POINT
 structure S : SPHERE) : GEOMETRY =
struct
 structure Point = P
 structure Sphere = S
end

```

## Funktorok és a típusmegosztás specifikálása egy példán (folyt.)

---

- Az előző funktordefiníciókkal a 2D-s programcsomag így valósítható meg:

```

structure Vector2D : VECTOR = ...

structure Point2D : POINT =
 PointFun (structure V = Vector2D)

structure Sphere2D : SPHERE =
 SphereFun (structure V = Vector2D and P = Point2D)

structure Geom2D : GEOMETRY =
 GeomFun (structure P = Point2D and S = Sphere2D)

```



## Funktorok és a típusmegosztás specifikálása egy példán (folyt.)

---

- Egyetlen baj van csupán: SphereFun és GeomFun típushibás! A hiba javítható: típusmegosztást kell előírnunk.

```

functor SphereFun
 (structure V : VECTOR
 structure P : POINT
 sharing P.Vector = V) : SPHERE =
struct
 structure Vector = V
 structure Point = P
 ...
end

functor GeomFun
 (structure P : POINT
 structure S : SPHERE
 sharing P.Vector = S.Vector
 sharing P = S.Point) : GEOMETRY =
struct
 structure Point = P
 structure Sphere = S
end

```

## A típusmegosztás elkerülése funktor használatakor

---

- Most is felvetődik a kérdés: elkerülhető-e típusmegosztás?
- Igen, de azon az áron, hogy erőszakosan megváltoztatjuk a program szerkezetét.
- A típusmegosztás fő előnye, hogy közvetlenül és tömören fejezi ki az elvárt összefüggéseket, de a paraméterek szignatúrájának definiálásakor még nem kell velük foglalkozni. Ez a tulajdonság nagyon megkönnyíti az „előre gyártott” programrészek újrafelhasználását, hiszen ilyen esetekben a típusmegosztás konkrét igényét előre (azaz pl. a paraméterek definiálásakor) lehetetlen megmondani.
- Vegyük elő a már látott példát, amelyben a megosztási specifikációk számát egyre csökkentettük.

```

signature EXTD_GEOMETRY =
sig
 structure Sphere : SPHERE
 structure SemiSpace : SEMI_SPACE
 sharing Sphere.Point = SemiSpace.Point
end

```

## A típusmegosztás elkerülése funktor használatakor (folyt.)

---

- Az EXTD\_GEOMETRY szignatúrát ezzel a funktorral valósítjuk meg:

```
functor ExtdGeomFun
 (structure Sp : SPHERE
 structure Ss : SEMI_SPACE
 sharing Sp.Point = Ss.Point) =
struct
 structure Sphere = Sp
 structure SemiSpace = Ss
end
```

- Ahhoz, hogy a megosztási előírást elhagyhassuk a funktor paraméteréből, gondoskodnunk kell arról, hogy az EXTD\_GEOMETRY szignatúra által előírt típusmegosztás teljesüljön.
- Megoldás lehet a POINT szignatúrát megvalósító struktúra „kiemelése”.
- Kétféle módon járhatunk el.

## A típusmegosztás elkerülése funktor használatakor (folyt.)

---

- Az első lehetőség az, hogy ExtdGeomFun paraméterként SPHERE és SEMI\_SPACE egy-egy megvalósítása helyett a közös elem, azaz POINT egy megvalósítását kapja, és a funktor törzsében hozza létre SPHERE és SEMI\_SPACE egy-egy megvalósítását.
- Ehhez a SphereFun és a SemiSpaceFun funktorokat is megfelelően kell paraméterezni:

```
functor SphereFun
 (structure P : POINT) : SPHERE =
struct
 structure Vector = P.Vector
 structure Point = P
 ...
end

functor SemiSpaceFun
 (structure P : POINT) : SEMI_SPACE =
struct
 ...
end
```

## A típusmegosztás elkerülése funktor használatakor (folyt.)

- Az `ExtdGeomFun` funktor első változatával, `ExtdGeomFun_1`-gyel több gond van:

```
functor ExtdGeomFun_1
 (structure P : POINT) : GEOMETRY =
struct
 structure Sphere = SphereFun (structure P = Point)
 structure SemiSpace = SemiSpaceFun (structure P = Point)
end
```

- `ExtdGeomFun_1`-ben alstruktúra-definícióban fordul elő `SphereFun` és `SemiSpaceFun`, és ez olyan paraméterekre korlátozza `ExtdGeomFun_1`-et, amelyek e két funktorral állíthatók elő. – Ez erős korlátozás `ExtdGeomFun`-hoz képest, amely `SPHERE`, ill. `SEMI_SPACE` bármely megvalósításának alkalmazását lehetővé teszi.
- `ExtdGeomFun_1`-nek paraméterként meg kell kapnia komponenseinek közös elemét, ill. elemeit (a példában a `POINT` szignatúrát megvalósító `P` struktúrát). Ez a megoldás kényelmetlenné válik, ha a programunk sok rétegből áll: a „legtávolabbi” elemtől kezdve a teljes hierarchiát fel építünk minden alkalommal.
- Nincs igazi indok arra, hogy `ExtdGeomFun_1` miért éppen `POINT` egy megvalósítását kapja paraméterként. (Nem elég nyomós) oka az, hogy `ExtdGeomFun_1`-nek fel kell építenie az említett hierarchiát a megosztási előírások kielégítéséhez.

## A típusmegosztás elkerülése funktor használatakor (folyt.)

- A második lehetőség az, hogy a paraméterként átadott közös elem, azaz `POINT` megvalósításához kötjük `SPHERE`, ill. `SEMI_SPACE` (ugyancsak paraméterként átveendő) megvalósítását, és így érjük el a típusgyenletek teljesülését.
- `ExtdGeomFun_2` alábbi deklarációja kielégíti a követelményeket, de láthatóan nincs semmi előnye a megosztási előírásokat tartalmazó kiinduló változathoz, `ExtdGeomFun`-hoz képest.

```
functor ExtdGeomFun_2
 (structure P : POINT
 structure Sp : SPHERE where Point = P
 structure Ss : SEMI_SPACE where Point = P) =
struct
 structure Sphere = Sp
 structure SemiSpace = Ss
end
```

- Inkább hátránynak tekinthető, hogy az eredeti változathoz képest egy harmadik paramétert vezettünk be, amelynek az egyetlen szerepe az, hogy elhagyhassuk a megosztási előírást.

## A típusmegosztás elkerülése funktor használatakor (folyt.)

---

- ExtGeomFun\_2 rafináltabb változata ExtGeomFun\_3 és ExtGeomFun\_4. Mindkettőnek csak két paraméterre van szüksége azért, hogy a kettő közül valamelyiket bizonyos értelemben kiemeltük, és előírtuk, hogy a másiknak vele kompatibilisnek kell lennie.

```

functor ExtGeomFun_3
 (structure Sp : SPHERE
 structure Ss : SEMI_SPACE where Point = Sp.Point) =
struct
 structure Sphere = Sp
 structure SemiSpace = Ss
end

functor ExtGeomFun_4
 (structure Ss : SEMI_SPACE
 structure Sp : SPHERE where Point = Ss.Point) =
struct
 structure Sphere = Sp
 structure SemiSpace = Ss
end

```

## A típusmegosztás elkerülése funktor használatakor (folyt.)

---

- E két utóbbi megoldásnak megvannak ugyanazok az előnyei, mint a megosztási előírást alkalmazó megoldásnak. De meg kellett törnünk a megoldás természetes szimmetriáját. Ez mondanivalónk lényege:
- A megosztási előírással a programozó *szimmetrikus helyzetet szimmetrikus módon* oldhat meg.
- A programozó helyett a fordítóprogram törli meg a szimmetriát, amikor ilyen vagy olyan megvalósítást választ. A programozó nem kényszerül arra, hogy önkényes, semmivel alá nem támasztható döntést hozzon.