

AZ SML MODULNYELVE

Modulok az SML-ben

- A modul: (1) *fordítási egység*, (2) a *csoportosítás* és az *elrejtés*, azaz az *absztrakció* eszköze.
- Szignatúra (*signature*): specifikációs modul; a struktúra *specifikációja*, „*típusa*”.
- Struktúra (*structure*): implementációs modul; a szignatúra *megvalósítása*.
- Funktor (*functor*): *struktúrával* paraméterezhető (generikus) modul; eredménye egy *struktúra*.
- Egy struktúra akkor és csak akkor valósít meg egy szignatúrát, ha a struktúra kielégíti a szignatúra által támasztott összes követelményt (később pontosítjuk).
- Modul-elnevezések az MOSML-ben (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 az MOSML-ben
 - `.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 ÉS STRUKTÚRA

Szignatúra és struktúra

A szignatúra alapváltozata

`sig specs end` alakú *szignatúrakifejezés*,

ahol a *specs* specifikációsorozat az alábbi elemeket tartalmazhatja:

- típusspecifikáció `type (tyvar1, ..., tyvarn) tycon [= typ]` alakban, ahol *typ* opcionális;
- adattípus-specifikáció (a `datatype`-deklarációval azonos alakban);
- kivételspecifikáció `exception excon of typ` alakban;
- értékspecifikáció `val id : typ` alakban.

A struktúra alapváltozata

`struct decs end` alakú *struktúrakifejezés*,

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;
- megadott típusú új nevet (névkonstruktort) létrehozó értékdeklaráció.

Szigatúra és struktúra (folyt.)

A szigatúra-deklaráció

- `signature sigid = sigexp` alakú, ahol
 - `sigid` egy szigatúranév,
 - `sigexp` pedig egy szigatúrakifejezés.
- A szigatúranév a szigatúrakifejezés rövidítése, szinonimája.

A struktúra-deklaráció egyszerű változata

- `structure strid = strexp` alakú, ahol
 - `strid` egy struktúranév,
 - `strex` pedig egy struktúrakifejezés.
- A struktúranév a struktúrakifejezés rövidítése, szinonimája.

A struktúra-deklaráció bonyolultabb változata

- a struktúra *szigatúrához kötése*, amely kétféle lehet:
 - áttetsző (opál, opaque): `structure strid :> sigid = strexp`
 - átlátszó (transzparens, transparent): `structure strid : sigid = strexp`

Példa: a KCsiga és a Csiga szigatúra

- A KCsiga struktúra (a keretprogram) szigatúrája

```
signature KCsiga =
sig
  val csigaBe : string -> TCsiga.feladvanyleiro
  val csigaKi : string * TCsiga.csigatabla list -> unit
  val megold : string * string -> string
end
```

- A Csiga struktúra (a főmodul) szigatúrája

```
signature Csiga =
sig
  val buvosCsiga :
    TCsiga.feladvanyleiro -> TCsiga.csigatabla list
end
```

Mindkét szigatúra csupán `val id : typ` alakú *értékspecifikációkat* tartalmaz.

Példa: a TCsiga struktúra és törzsszigatúrája

- A TCsiga struktúra (a típusleíró modul) és *törzsszigatúrája*

```

structure TCsiga =
struct
  type meret          = int
  type ciklus         = int
  type sorszam        = int
  type oszlopszam     = int
  type ertek          = int
  type adottElem      =
    sorszam * oszlopszam * ertek
end

(* TCsiga.sml
   törzsszigatúrája *)
type meret          = int
type ciklus         = int
type sorszam        = int
type oszlopszam     = int
type ertek          = int
type adottElem      =
  int * int * int

type feladvanyleiro =
  meret * ciklus * adottElem list

type ertekVagyUres  = int
type sor            =
  ertekVagyUres list

type csigatabla     =
  sor list

type feladvanyleiro =
  int * int * (int * int * int) list

type ertekVagyUres  = int
type sor            =
  int list

type csigatabla     =
  int list list

```

Példa: a TCsiga struktúra és törzsszigatúrája (folyt.)

- *Törzsszigatúra* (principal signature): egy struktúra összetevőinek *lehető legrspecifikusabb* leírása.
- TCsiga csupa típusspecifikációt tartalmaz


```

type (tyvar1, ..., tyvarn) tycon [ = typ ],

```

 pontosabban


```

type tycon = typ

```

 alakban.
- Egy struktúra szigatúra-megkötés nélkül nem rejti el a részleteket, azaz *gyenge absztrakciót* valósít meg.

Példa: változatok a TCsiga struktúrára és szigatúrára

• Struktúra *átlátszó* szigatúrával

```
structure TCsiga : TCsiga =
struct
  type meret          = int
  ...
  type adottElem     =
    sorszam * oszlopszam * ertek
  type feladvanyleiro =
    meret * ciklus * adottElem list
  type ertekVagyUres = int
  type sor           =
    ertekVagyUres list
  type csigatabla    = sor list
end
```

• Struktúra *áttetsző* szigatúrával

```
structure TCsiga :> TCsiga =
struct
  type meret          = int
  ...
  type adottElem     =
    sorszam * oszlopszam * ertek
  type feladvanyleiro =
    meret * ciklus * adottElem list
  type ertekVagyUres = int
  type sor           =
    ertekVagyUres list
  type csigatabla    = sor list
end
```

• Szigatúra (a részleteket *elrejtő*) absztrakt adattípus megvalósításához

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

Példa: sort megvalósító szigatúra 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
Minden típus és érték polimorf!

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
típuskonstruktor deklarálása
kivételkonstruktor deklarálása
névkonstruktorok deklarálása
```

Példa: sort megvalósító szigatúra és struktúra (folyt.)

• Struktúra áttetsző szigatúrával

```
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 szigatúrával

```
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 szigatúra és struktúra (folyt.)

• Struktúra *bővített* áttetsző szigatúrával

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

Példa: sort megvalósító szigatúra és struktúra (folyt.)

- Szigatúra-öröklődés specializációval, 1. változat

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

- Szigatúra-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 szigatúrával

```
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 szigatúra és struktúra (folyt.)

- Szigatúra-öröklődés inklúzióval (és az eredeti szigatúra 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: type 'a queue = 'a list * 'a list nincs specifikálva QUEUE-ban, ezért hiába van deklarálva 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 szigatúra 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
```


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 adatsztrakciót 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.

Szigatúra-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.

Szigatúra-illeszkedés: MERGEABLE_QUEUE, MERGEABLE_INT_QUEUE

- Említettük, hogy a szigatúra-jelöltben az értékek típusa általánosabb lehet, mint a célszigatúrában.
- A szigatúra-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 szigatúra-jelölt illeszkedik a MERGEABLE_INT_QUEUE célszigatúrára, mert az előbbiben specifikált polimorf merge függvény típusát leíró típuskifejezés típusváltozója leköthető az int típusal. Fordítva nem igaz.

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.)

- Összefoglalásként 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 (ui. az illeszkedési reláció reflexív).
- Bármely szignatúra, amelyet egy struktúra képes megvalósítani, nem lehet *gyengébb* az adott struktúra törzsszignatúrájánál.
- A törzsszignatúra a *leggyengébb* szignatúra, amelyet egy struktúra megvalósíthat.

SZIGNATÚRA-KÖTÉS

Szignatúra-kötés

- *Szignatúra-kötéssel* (signature ascription) írjuk elő, hogy egy struktúra valósítson meg egy szignatúrát.
- A szignatúra-kötés *erősíti* 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 = strexp`
 - áttetsző: `structure strid :> sigexp = strexp`
- A típusellenőrzés lépései szignatúrához kötött struktúra-deklaráció esetén a következők:
 - `strexp` megvalósítja-e `sigexp`-et? Ennek eldöntéséhez a fordító
 - meghatározza `strexp sigexp0` törzsszignatúráját, és megpróbálja illeszteni a `sigexp` célszignatúrára; valamint
 - előállítja a bővített `sigexp'` szignatúrát úgy, hogy `sigexp`-et bővíti a `sigexp0`-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 `strexp`-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 szigatúra-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 szigatúra-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 állapot típusának 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 eljáráshoz 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 szigatúra-kötéssel: prioritási sor

Lássuk a következő példát!

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

```
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 szigatúra-kötéssel: prioritási sor (folyt.)

- A megvalósítástól független absztrakt típus *áttetsző szigatúrát* 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 szigatúrát bővítjük, és a bővített szigatúrát 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 szigatúra-kötéssel: ORDERED, MyString

- Átlátszó szigatúra-kötéssel csökkenthető az explicit típus-specifikációk száma a szigatúrában. De az se jó, ha túl sokat használjuk, ui. a típusinformációk láthatóvá tételével csökken a modulok függetlensége.
- Az átlátszó szigatúra-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 típusdefiniókat, amelyekre szükség van.
- Az ORDERED szigatúra 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 szigatúrát, mint láttuk, *csak átlátszóan* érdemes egy struktúrához kötni, különben nem tudnánk `t` típusú értékeket létrehozni.

Struktúra-deklaráció átlátszó szigatúra-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 füzerek összehasonlítását karakterek összehasonlítására vezeti vissza, `clt`-t elrejt. `String.t` a külvilág számára is ekvivalens `string`-gel, bár ez az ORDERED szigatúrából nem látszik: `MyString` tényleges, *látható* szigatúrája ugyanis:

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

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

Struktúra-deklaráció átlátszó szigatúra-kötéssel: ORDERED, IntLt, IntDiv

- Tegyük föl, hogy egészeket kétféle alapon akarunk összehasonlítani, de nem akarjuk elrejteni, hogy egészekről 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 szigatúra-kötés a típuslevezetéshez hasonlóan megkönnyíti a programozó dolgát: kevesebbet kell írnia. *De ára van:*
- átlátszó szigatúra-kötés esetén a szigatúra önmagában nem fordítható le, csak a struktúrával együtt: csak így állítható elő a struktúra tényleges, *látható* szigatúrája.
- Vagyis az összes olyan programrész, amely e struktúra látható szigatúrájára hivatkozik, *függ* e struktúra *megvalósításától!*
- Amíg az átlátszó szigatúra-kötés függőséget okoz, az áttetsző szigatúra-kötés kiküszöböli a függőséget.
- Ha egy struktúrához áttetsző módon kötjük a szigatúrát, a rá hivatkozó programrészek megbízhatnak a szigatúrában (a struktúra látható szigatúrája ui. ekvivalens az áttetsző szigatúrával).
- 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 HIERARCHIKUS ÖSSZEKAPCSOLÁSA

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.

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

```

- Egészek aritmetikai összehasonlítása

```

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

```

- Füzek lexikografikus összehasonlítása

```

structure LexString : ORDERED =
struct
  type t = string
  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

```

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

MODULOK SZIMMETRIKUS ÖSSZEKAPCSOLÁSA

Típusmegosztás specifikálása

- 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- és a vektor-absztrakciót *két-két 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 SPHERE 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.

PARAMÉTEREZHETŐ MODULOK

Paramétezhető, más néven generikus modulok

- Az újrafelhasználhatóságot segíti elő a *paramétezhető*, 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*:
$$\text{functor } \textit{funid} (\textit{decs}) : \textit{sigexp} = \textit{stexp}$$
- és az *áttetsző*:
$$\text{functor } \textit{funid} (\textit{decs}) :> \textit{sigexp} = \textit{stexp}$$
- 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.
- A funktor structure K : ORDERED formális paraméterének specifikálása önmagában nem elég, a structure Key : ORDERED = K deklarációra is szükség van a funktor törzsében.

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: DICT where type Key.t = int.

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 építenünk 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.)

- ExtDGeomFun_2 rafináltabb változata ExtDGeomFun_3 és ExtDGeomFun_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 ExtDGeomFun_3
  (structure Sp : SPHERE
   structure Ss : SEMI_SPACE where Point = Sp.Point) =
struct
  structure Sphere = Sp
  structure SemiSpace = Ss
end

functor ExtDGeomFun_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.

A diák Robert Harper *Programming in Standard ML c. jegyzete Part III. The Module Language c. részének* felhasználásával készültek (ld. <http://www-2.cs.cmu.edu/rwh/smlbook>).