

FUNKCIONÁLIS PROGRAMOZÁS

TÍPUS ÉS FÜGGVÉNY

Történeti áttekintés

A λ -kalkulus rövid története

- 1910: Russel & Whitehead: Principia Mathematica
- 1935: Gödel: nem létezhet erős axióma- és tételrendszer, amely teljes
- 1930-40-es évek: Alonzo Church: λ -kalkulus

A Russel paradoxon

- R : az önmagukat nem tartalmazó halmazok halmaza
- R tartalmazza-e önmagát? \Rightarrow ellentmondás

A Russel paradoxon a λ -kalkulusban

- $R = \lambda x. \neg(x x)$
- $R R = \neg(R R) \Rightarrow$ ellentmondás

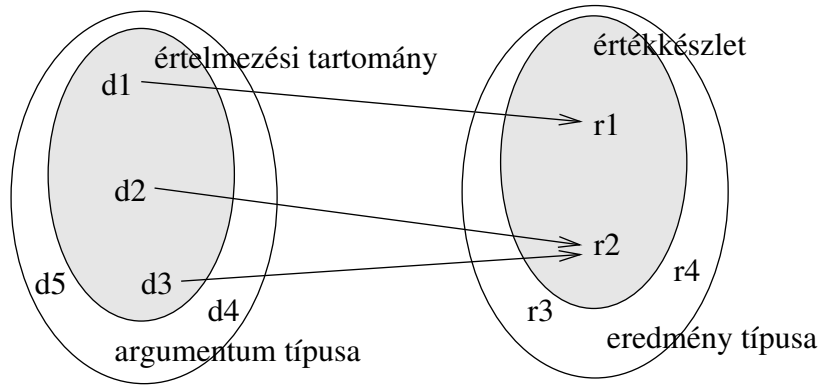
Típusos λ -kalkulus

- minden kifejezésnek van egy típusa
- f csak akkor alkalmazható e -re, ha a típusaik „passzolnak”
- R nem írható le

A típus és a függvény fogalma

- A típus fogalma
 - A típus értékek egy halmaza (pl. egész típus ~ az egész számok egy (rész)halmaza)
 - Tetszőleges típus jelölése az ún. *típuselméletben*: $\alpha, \beta \dots$
 - Típusállandó (azaz konkrét típus) jelölése az SML-ben: `int, real, char, string ...`
 - Típusváltozó jelölése az SML-ben $\alpha, \beta \dots$ helyett: `'a, 'b ...`
- A függvény fogalma
 - A függvény valamely D halmaznak valamely R halmazba való olyan *egyértelmű* leképezése, amelyet a (d, r) rendezett párok halmaza ad meg, ahol $d \in D$ és $r \in R$.
 - A d a függvény argumentuma (paramétere), az r az eredménye
 - A D a függvény értelmezési tartománya, az R az értékkészlete
 - A típusos nyelvekben d is, r is *meghatározott* típusú
 - A függvény értelmezési tartománya \subseteq az argumentum típusa
 - A függvény értékkészlete \subseteq az eredmény típusa

A függvény mint leképezés



Függvények osztályozása, ill. alkalmazása

- Függvények osztályozása
 - Parciális függvény: értelmezési tartomány \subset argumentum típusa (figyelem: hibák forrása lehet!)
 - Teljes függvény: értelmezési tartomány = argumentum típusa
 - Szürjektív függvény: értékkészlet = eredmény típusa
 - Nemszürjektív függvény: értékkészlet \subset eredmény típusa
 - Injektív függvény: a leképezés kölcsönösen egyértelmű
 - Az $f : \alpha \rightarrow \beta$ injektív függvény inverze: $f^{-1} : \beta \rightarrow \alpha$
 - Bijektív = injektív + szürjektív, azaz f bijektív, ha f^{-1} teljes függvény
- Függvények alkalmazása
 - *Függvényalkalmazást* jelöl az f és e jelek egymás mellé írása („*juxtapozicionálása*”): $f e$ azt jelenti, hogy az f -et alkalmazzuk az e -re.
 - Általánosabban: az $f e$ kifejezésben az e tetszőleges olyan kifejezés, amelynek az értéke az f értelmezési tartományába esik.
 - Még általánosabban: az $f e$ kifejezésben az f függvényértéket adó tetszőleges kifejezés, e pedig tetszőleges olyan kifejezés, amelynek értéke az f értelmezési tartományába esik.

A függvény mint érték

- A függvény „teljes jogú” (*first-class*) érték a funkcionális programozási nyelvekben
 - A függvény típusa általában: $\alpha \rightarrow \beta$, ahol az α az argumentum, a β az eredmény típusát jelöli
 - A függvény maga is: érték. *Függvényérték*.
 - Fontos: a függvényérték *nem* a függvény egy alkalmazásának az eredménye!
 - Példák függvényértékre
 - \sin (a típusa: *valós* \rightarrow *valós*)
 - round (a típusa: *valós* \rightarrow *egész*)
 - \circ (függvénykompozíció; a típusa: $((\beta \rightarrow \gamma) * (\alpha \rightarrow \beta)) \rightarrow (\alpha \rightarrow \gamma)$)
 - Példák függvényalkalmazásra
 - $\text{round } 5.4 = 5$, azaz egy *egész* típusú érték az eredménye ennek a függvényalkalmazásnak
 - $\text{round } \circ \sin$ (a típusa: *valós* \rightarrow *egész*)
 - $(\text{round } \circ \sin) 1.0 = 1$ (a típusa: *egész*)

Két- vagy többargumentumú függvények

- Függvény alkalmazása két- vagy több argumentumra
 1. Az argumentumokat *összetett adatnak* – párnak, rekordnak, listának stb. – tekintjük
 - pl. $f(1, 2)$ az f függvény alkalmazását jelenti az $(1, 2)$ *párra*,
 - pl. $f[1, 2, 3]$ az f függvény alkalmazását jelenti az $[1, 2, 3]$ *listára*.
 2. A függvényt több egymás utáni lépésben alkalmazzuk az argumentumokra, pl. $f 1 2 \equiv (f 1) 2$ azt jelenti, hogy
 - az első lépésben az f függvény alkalmazzuk az 1 értékre, ami egy *függvényt ad eredményül*,
 - a második lépésben az első lépésben kapott függvényt alkalmazzuk a 2 értékre, így kapjuk meg az $f 1 2$ függvényalkalmazás (vég)eredményét.
- Az $f 1 2$ esetben az f függvényt *részlegesen alkalmazható* függvénynek nevezzük.
- A programozó szabadon dönthet, hogy a függvényt részlegesen alkalmazható vagy pl. egy párra alkalmazható formában írja meg. A különbség *csak* a szintaxisban van. A részlegesen alkalmazható változat, mint látni fogjuk, rugalmasabban használható.
- Infix jelölés: $x \oplus y \equiv a \oplus$ függvény alkalmazása az (x, y) párra mint argumentumra

Függvények alkalmazása az SML-ben

- Az SML-ben az f és az e tetszőleges *név* lehet, amelyeket megfelelően *szeparálni* kell egymástól: $f e$, vagy $f(e)$, vagy $(f)e$
- Szeparátor: nulla, egy vagy több *formázó* karakter (\lfloor , $\backslash t$, $\backslash n$ stb.). Nulla db formázó karakter elegendő pl. a (előtt és a) után.
- FONTOS! A szeparátor a legerősebb balra kötő infix operátor az SML-ben.
- Példák:

$$\text{Math.sin } 1.00 \text{ (Math.cos) Math.pi round(3.17)}$$

$$2 + 3 \quad \text{(real) (3 + 2 * 5)}$$
- Függvények egy csoportosítása az SML-ben
 - Beépített függvények, pl. $+$, $*$ (mindkettő infix), real , round (mindkettő prefix)
 - Könyvtári függvények, pl. Math.sin , Math.cos , Math.pi (0 argumentumú!)
 - Felhasználó által definiálható függvények, pl. terulet , $/\backslash$, head

Deklaratív programozás. BME VIK, 2003. tavaszi félév

(Funkcionális programozás)

FÜGGVÉNYEK AZ SML-BEN

Függvények az SML-ben FP-11

SML-példa: Egyszeres Hamming-távolságú ciklikus kód

- A függvényt pl. *táblázattal* adhatjuk meg:

00	01	fn 00 => 01
01	11	01 => 11
11	10	11 => 10
10	00	10 => 00
- Változatok („klózek”): minden lehetséges esetre egy változat.
- Az fn (olvasd: *lambda*) névtelen függvényt, *függvénykifejezést* vezet be.
- A függvény néhány alkalmazása:
 - $(fn \ 00 \Rightarrow 01 \ | \ 01 \Rightarrow 11 \ | \ 11 \Rightarrow 10 \ | \ 10 \Rightarrow 00) \ 10$
 - $(fn \ 00 \Rightarrow 01 \ | \ 01 \Rightarrow 11 \ | \ 11 \Rightarrow 10 \ | \ 10 \Rightarrow 00) \ 11$
 - $(fn \ 00 \Rightarrow 01 \ | \ 01 \Rightarrow 11 \ | \ 11 \Rightarrow 10 \ | \ 10 \Rightarrow 00) \ 111$
- Mintaillesztés, egyesítés
- Érthető, de nem robosztus (vö. parciális a függvény!).

Deklaratív programozás. BME VIK, 2003. tavaszi félév

(Funkcionális programozás)

Függvények az SML-ben FP-12

SML-példa: modulo n alapú inkrementálás

- A függvényt általában *algoritmussal* adjuk meg (nem táblázattal), egyébként
 - az argumentum nem lehetne változó,
 - túl sok változatot kellene felírni stb.
- $fn \ i \Rightarrow (i + 1) \bmod n$
 - az i ún. *kötött változó*, a névtelen függvény argumentuma
 - az n ebben a kifejezésben *szabad változó*, és nincs értéke (!)
 - az n -et is *le kell kötni* mint a függvény argumentumát
- $fn \ n \Rightarrow fn \ i \Rightarrow (i + 1) \bmod n$
- A függvény néhány alkalmazása:
 - $(fn \ n \Rightarrow (fn \ i \Rightarrow (i + 1) \bmod n)) \ 128 \ 111$
 - $(fn \ n \Rightarrow (fn \ i \Rightarrow (i + 1) \bmod n)) \ 4 \sim 7$
 - $(fn \ n \Rightarrow (fn \ i \Rightarrow (i + 1) \bmod n)) \ 128 \ 6.0$ – Hiba!

Deklaratív programozás. BME VIK, 2003. tavaszi félév

(Funkcionális programozás)

Értékdeklaráció SML-ben: függvényérték deklarálása

- Név kötése függvényértékhez

- `val kovKod = fn 00 => 01 | 01 => 11 | 11 => 10 | 10 => 00`
- `val incMod = fn n => fn i => (i + 1) mod n`

- Szintaktikus édesítőszerral (`fun`)

- `fun kovKod 00 = 01`
`| kovKod 01 = 11`
`| kovKod 11 = 10`
`| kovKod 10 = 00`
- `fun incMod n i = (i + 1) mod n`

- Alkalmazásuk argumentumra

- `kovKod 01`
- `incMod 128 111`

LISTÁK

Fejkomment

Írjunk *deklaratív fejkommentet* minden (függvény)érték-deklarációhoz!

- (* kovKod cc = az egyszeres Hamming-távolságú, kétbites, ciklikus kódkészlet cc-t követő eleme

```
PRE: cc ∈ {00, 01, 11, 10}
```

*)
`fun kovKod 00 = 01`
`| kovKod 01 = 11`
`| kovKod 11 = 10`
`| kovKod 10 = 00`
- PRE = *precondition*, előfeltétel
- PRE: `cc ∈ {00, 01, 11, 10}` jelentése: a `kovKod` függvény `cc` argumentumának a `{00, 01, 11, 10}` halmazbeli értéknek kell lennie, ellenkező esetben a függvény eredménye nincs definiálva.
- (* `incMod n i = (i+1) modulo n` szerint

```
PRE: n > i >= 0
```

*)
`fun incMod n i = (i+1) mod n`

Lista: definíciók, adat- és típuskonstruktorok

- Definíciók

1. A *lista* azonos típusú elemek véges (de nem korlátos!) sorozata.
2. A lista olyan *rekurzív* lineáris adatszerkezet, amely azonos típusú elemekből áll, és
 - vagy üres,
 - vagy egy elemből és az elemet követő listából áll.

- Konstruktorok

- Az üres lista jele a `nil` *adatkonstruktorállandó*.
- A `nil` helyett általában a `[]` jelet használjuk (szintaktikus édesítőszerr).
- A `nil` típusa: `'a list`.
- Az `'a` *típusváltozót* jelöl, a `list`-et *típuskonstruktor*nak nevezzük.
- A `::` *adatkonstruktorfüggvény* új listát hoz létre egy elemből és egy (esetleg üres) listából.
- A `::` típusa `'a * 'a list -> 'a list`, infix pozíciójú, 5-ös precedenciájú, jobbra köt. Infix pozíciója miatt *adatkonstruktoroperátornak* is nevezzük.
- A `::`-ot *négyespontnak* vagy *cons*-nak olvassuk (vö. *constructor*, ami ennek az adatkonstruktorfüggvénynek a hagyományos neve a λ -kalkulusban és egyes funkcionális nyelvekben).

Lista: jelölések, minták

• Példák

• Lista létrehozása adatkonstruktorokkal

```
[] nil #"" :: nil
3 :: 5 :: 9 :: nil = 3 :: (5 :: (9 :: nil))
```

• Szintaktikus édesítőszer lista jelölésére

```
[3, 5, 9] = 3 :: 5 :: 9 :: nil
```

• Vigyázat! A Prolog listajelölése hasonló, de vannak lényeges különbségek:

SML	Prolog		SML	Prolog	
[]	[]	azonos	(x::xs)	[X Xs]	különböző
[1,2,3]	[1,2,3]	azonos	(x::y::ys)	[X,Y Ys]	különböző

• Minták

A [], nil adatkonstruktorállandóval és a :: adatkonstruktoroperátorral felépített kifejezések, valamint a [x1, x2, ..., xn] listajelölés mintában is alkalmazhatók.

Lista: hossz (length), elemek összege (isum), szorzata (rprod)

• Egy lista hosszát adja eredményül a length függvény (vö. List.length).

```
(* length : 'a list -> int
   length zs = a zs lista elemeinek száma *)
fun length [] = 0
  | length (_ :: zs) = 1 + length zs
```

• Egy egész számokból álló lista elemeinek összegét adja eredményül isum.

```
(* isum : int list -> int
   isum ns = az ns egészlista elemeinek összege *)
fun isum [] = 0
  | isum (n :: ns) = n + isum ns
```

• Egy valós számokból álló lista elemeinek szorzatát adja eredményül rprod.

```
(* rprod : real list -> real
   rprod xs = az xs valós lista elemeinek szorzata *)
fun rprod [] = 1.0
  | rprod (x :: xs) = x * rprod xs
```

Lista: fej (hd), fark (tl)

• A nemüres lista első eleme a lista feje.

```
(* hd : 'a list -> 'a
   hd xs = a nemüres xs első eleme (az xs feje)
*)
fun hd (x :: _) = x;
```

• A nemüres lista első utáni elemeiből áll a lista farka.

```
(* tl : 'a list -> 'a list
   tl xs = a nemüres xs első utáni elemeinek az eredetivel
   azonos sorrendű listája (az xs farka)
*)
fun tl (_ :: xs) = xs;
```

• hd és tl *parciális* függvények. Ha könyvtárbeli megfelelőiket (List.hd, List.tl) üres listára alkalmazzuk, Empty néven *kivételt* jeleznek.

• Az _ (aláhúzás) az ún. *mindenesjel*, azaz a mindenre illeszkedő minta. Figyelem: a mindenesjel kifejezésben – pl. egyenlőségjel jobb oldalán – nem használható!

Példák: hd, tl, length, isum, rprod

• hd, tl

A kifejezés	Az mosml válasza
List.hd [1, 2, 3];	> val it = 1 : int
List.hd [];	! Uncaught exception: ! Empty
List.tl [1, 2, 3];	> val it = [2, 3] : int list
List.tl [];	! Uncaught exception: ! Empty

• length, isum, rprod

A kifejezés	Az mosml válasza
length [1, 2, 3, 4];	> val it = 4 : int
length [];	> val it = 0 : int
isum [1, 2, 3, 4];	> val it = 10 : int
isum [];	> val it = 0 : int
rprod [1.0, 2.0, 3.0, 4.0];	> val it = 24.0 : real
rprod [];	> val it = 1.0 : real

Kifejezés lokális érvényű deklarációval: let-kifejezés

- Ún. let-kifejezést használunk, ha ismétlődő részkifejezéseket *csak egyszer* akarunk kiszámítani, vagy akkor, ha bizonyos értékeket *el akarunk rejtetni* a program többi része előtt.
- Szintaxisa:

```
let d      ahol d egy nemüres deklarációsorozat,
in e
end      e egy nemüres kifejezés.
```

- Példa:

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

Deklaratív programozás. BME VIK, 2003. tavaszi félév

(Funkcionális programozás)

Egyszerű adattípusok FP-24

Egyszerű adattípusok

LOKÁLIS ÉRVÉNYŰ DEKLARÁCIÓK

EGYSZERŰ ADATTÍPUSOK

Típusnév	Megnevezés	Könyvtár
int	előjeles egész	Int
real	racionális (valós)	Real
char	karakter	Char
bool	logikai	Bool
string	fűzér	String
word	előjel nélküli egész	Word
word8	8 bites előjel nélküli egész	Word8

Deklaratív programozás. BME VIK, 2003. tavaszi félév

(Funkcionális programozás)

Különleges állandók

- Előjeles egész állandó
Példák: 0 ~0 4 ~04 999999 0xFFFF ~0x1ff
Ellenpéldák: 0.0 ~0.0 4.0 1E0 -317 0xFFFF -0x1ff
- Racionális (valós) állandó
Példák: 0.7 ~0.7 3.32E5 3E~7 ~3E~7 3e~7 ~3e~7
Ellenpéldák: 23 .3 4.E5 1E2.0 1E+7 1E-7
- Előjel nélküli egész állandó
Példák: 0w0 0w4 0w999999 0wxFFFF 0wx1ff
Ellenpéldák: 0w0.0 ~0w4 -0w4 0w1E0 0wxFFFF 0wxXXXX
- Karakterállandó: a # jelet közvetlenül követő, egykarakteres füzérállandó (l. a következő lapon).
Példák: # "a" # "\n" # "\^Z" # "\255" # "\""
Ellenpéldák: # "a" #c # "" # 'a'
- Logikai állandó: csupán kétféle lehet.
Példák: true false
Ellenpéldák: TRUE False 0 1

NEVEK, OPERÁTOROK

Különleges állandók, escape-szekvenciák

- Füzérállandó: idézőjelek (") között álló nulla vagy több nyomtatható karakter, szóköz vagy \ jellel kezdődő *escape-szekvencia* (l. a táblázatot).
- Escape-szekvenciák

<code>\a</code>	Csengőjel (BEL, ASCII 7).
<code>\b</code>	Visszalépés (BS, ASCII 8).
<code>\t</code>	Vízszintes tabulátor (HT, ASCII 9).
<code>\n</code>	Újsor, soremelés (LF, ASCII 10).
<code>\v</code>	Függőleges tabulátor (VT, ASCII 11).
<code>\f</code>	Lapdobás (FF, ASCII 12).
<code>\r</code>	Kocsi-vissza (CR, ASCII 13).
<code>\^c</code>	Vezérlő karakter, ahol $64 \leq c \leq 95$ (@ ...), és <code>\^c</code> ASCII-kódja 64-gyel kevesebb <i>c</i> ASCII-kódjánál.
<code>\ddd</code>	A <i>ddd</i> kódú karakter (<i>d</i> decimális számjegy).
<code>\uxxxx</code>	Az <i>xxxx</i> kódú karakter (<i>x</i> hexadecimális számjegy).
<code>\"</code>	Idézőjel (").
<code>\\</code>	Hátratórt-vonal (\).
<code>\f...f\</code>	Figyelmen kívül hagyott sorozat. <i>f...f</i> nulla vagy több formázókaraktert (szóköz, HT, LF, VT, FF, CR) jelent.

Nevek képzési szabályai

- Alfánumerikus név: kis- és nagybetűk, számjegyek, percjelek (') és aláhúzás-jelek () olyan sorozata, amely betűvel vagy perccel kezdődik
 - Példák: tothGyorgy Toth_3_Gyorgy toth'gyorgy 'gyurika
 - Percjellel kezdődő név csak típusváltozót jelölhet.
- Írásjelekből álló név: az alábbi 20 jel tetszőleges, nem üres sorozata


```
! % & $ # + - / : < = > ? @ \ ~ ' ^ | *
```

 - Példák: ++ <-> ||| ## |=|
- Speciális a szerepe az alábbi fenntartott jeleknek


```
( ) [ ] { } , ; . ...
```
- Más jelentés nem rendelhető az alábbi fenntartott nevekhez és jelekhez

```
abstype and andalso as case do datatype else end eqtype exception
fn fun functor handle if in include infix infixr let local nonfix
of op open orelse raise rec sharing sig signature struct structure
then type val where with withtype while : :: :> _ | => -> #
```

A beépített operátorok és precedenciájuk

Az alábbi táblázatban *wordint*, *num* és *numtxt* az alábbi típusnevek helyett állnak.

```
wordint = int, word, word8
num = int, real, word, word8
numtxt = int, real, word, word8, char, string
```

Prec.	Operátor	Típus	Eredmény	Kivétel
7	*	<i>num</i> * <i>num</i> -> <i>num</i>	szorzat	Overflow
	/	<i>real</i> * <i>real</i> -> <i>real</i>	hányados	Div, Overflow
	div, mod	<i>wordint</i> * <i>wordint</i> -> <i>wordint</i>	hányados, maradék	Div, Overflow
	quot, rem	<i>int</i> * <i>int</i> -> <i>int</i>	hányados, maradék	Div, Overflow
6	+, -	<i>num</i> * <i>num</i> -> <i>num</i>	összeg, különbség	Overflow
	^	<i>string</i> * <i>string</i> -> <i>string</i>	egybeírt szöveg	Size
5	::	'a' * 'a list' -> 'a list	elemmel bővített lista (jobbra köt)	
	@	'a list' * 'a list' -> 'a list	összefűzött lista (jobbra köt)	
4	=, <>	'a' * 'a' -> bool	egyenlő, nem egyenlő	
	<, <=	<i>numtxt</i> * <i>numtxt</i> -> bool	kisebb, kisebb-egyenlő	
	>, >=	<i>numtxt</i> * <i>numtxt</i> -> bool	nagyobb, nagyobb-egyenlő	
3	:=	'a ref' * 'a' -> unit	értékkadás	
	o	('b -> 'c) * ('a -> 'b) -> ('a -> 'c)	két függvény kompozíciója	
0	before	'a' * 'b' -> 'a	a bal oldali argumentum	

div -∞, quot 0 felé kerekít. div és quot, ill. mod és rem eredménye csak akkor azonos, ha két operandusuk azonos előjelű (mindkettő pozitív, vagy mindkettő negatív).

map: adott függvény alkalmazása egy lista minden elemére

- Példa: vonjunk négyzetgyököt egy valós számokból álló lista minden eleméből!

```
map Math.sqrt [1.0, 4.0, 9.0, 16.0] = [1.0, 2.0, 3.0, 4.0]
```

- Általában: $\text{map } f [x_1, x_2, \dots, x_n] = [f x_1, f x_2, \dots, f x_n]$

- map definíciója (map polimorf függvény!):

```
(* map : ('a -> 'b) -> 'a list -> 'b list
   map f xs = az xs f-fel átalakított elemeiből álló lista
*)
fun map f [] = []
  | map f (x :: xs) = f x :: map f xs
```

- map típusa (mivel a -> típusoperátor jobbra köt!):

```
('a -> 'b) -> 'a list -> 'b list ≡ ('a -> 'b) -> ('a list -> 'b list)
```

- A map egy ún. *részlegesen alkalmazható*, magasabbrendű függvény: ha egy 'a -> 'b típusú függvényre alkalmazzuk, akkor egy 'a list -> 'b list típusú **függvényt** ad eredményül. A kapott függvényt egy 'a list típusú listára alkalmazva egy 'b list típusú listát kapunk.
- map – teljes nevén List.map – belső függvény az SML-ben.

LISTÁK

PROGRAMHELYESSÉG

A program helyességének (informális) igazolása a map példáján

- A rekurzív programról be kell látnunk, hogy
 - funkcionálisan helyes (azt kapjuk eredményül, amit várunk),
 - a kiértékelése biztosan befejeződik (nem „végtelen” a rekurzió).
- Bizonyítása hossz szerinti *strukturális indukcióval* lehetséges (vissavezethető a teljes indukcióra).

```
fun map f [] = []
  | map f (x :: xs) = f x :: map f xs
```

- Feltesszük, hogy a map jó eredményt ad az eggyel rövidebb listára (azaz a lista farkára).
- Alkalmazzuk az f-et a lista első elemére (a fejére).
- A fej transzformálásával kapott eredményt a fark transzformálásával kapott lista elé fűzve valóban a várt eredményt kapjuk.
- A kiértékelés véges számú lépésben befejeződik, mert
 - a lista véges,
 - a map függvényt a *rekurzív ágban* minden lépésben egyre rövidülő listára alkalmazzuk, és
 - gondoskodtunk a rekurzió leállításáról (a *triviális eset* kezeléséről, ui. van nem rekurzív ág).

Néhány belső, ill. könyvtári függvény

- `explode` : `string -> char list` – a fűzér karaktereiből álló lista
pl. `explode "abc" = ['#"a", #'b", #'c"]`
 - `implode` : `char list -> string` – a karakterlista elemeiből álló fűzér
pl. `implode ['#"a", #'b", #'c"] = "abc"`
 - `map`-nek más változatai is vannak, amelyek egyéb összetett adatokra alkalmazhatók. Például
 - `String.map` : `(char -> char) -> string -> string`
 - `Vector.map` : `('a -> 'b) -> 'a vector -> 'b vector`
 - A Char könyvtárban sok hasznos ún. *tesztelő* függvény található, például:
 - `Char.isLower` : `char -> bool` – igaz az angol ábécé kisbetűire
 - `Char.isSpace` : `char -> bool` – igaz a hat formázó karakterre
 - `Char.isAlpha` : `char -> bool` – igaz az angol ábécé betűire
 - `Char.isAlphaNum` : `char -> bool` – igaz az angol ábécé betűire és a számjegyekre
 - `Char.isAscii` : `char -> bool` – igaz a 128-nál kisebb ascii-kódú karakterekre
- ```
pl. Char.isSpace #"\" = true; Char.isAlphaNum #"!" = false
```

## NÉHÁNY BELSŐ, ILL. KÖNYVTÁRI FÜGGVÉNY

## LISTÁK

## filter: adott predikátumot kielégítő elemek kiválogatása

- Példa: gyűjtjük ki a kisbetűket egy karakterlistából!

```
List.filter Char.isLower (explode "VaLtOgAtVa") = ["#a", "#t", "#g", "#t", "#a"]
```

- Általában, ha  $p x_1 = \text{true}$ ,  $p x_2 = \text{false}$ ,  $p x_3 = \text{true}$ , ...,  $p x_{2k+1} = \text{true}$ , akkor  $\text{filter } p [x_1, x_2, x_3, \dots, x_{2k+1}] = [x_1, x_3, \dots, x_{2k+1}]$ .

- filter definíciója:

```
(* filter : ('a -> bool) -> 'a list -> 'a list
 filter p zs = a zs p-t kielégítő elemeiből álló lista
*)
fun filter _ [] = []
 | filter p (x :: xs) =
 if p x then x :: filter p xs else filter p xs
```

- filter típusa, ha egyargumentumú függvénynek tekintjük ( $\rightarrow$  jobbra köt!):

```
filter : ('a -> bool) -> ('a list -> 'a list).
```

Azaz ha `filter`-t egy `'a -> bool` típusú függvényre (predikátumra) alkalmazzuk, akkor egy `('a list -> 'a list)` függvényt ad eredményül. A kapott függvényt egy `'a list` típusú listára alkalmazva egy `'a list` típusú listát kapunk.

## POLIMORFIZMUS

## Lista legnagyobb elemének megkeresése

- Egy egészlista legnagyobb elemének kiválasztásához szükségünk van az `Int.max` függvényre.

- Üres listának nincs legnagyobb eleme,
- egyelemű listában az egyetlen elem a legnagyobb,
- legalább kételemű lista legnagyobb elemét úgy kapjuk, hogy az első elem és a maradéklista elemeinek legnagyobbika közül vesszük a nagyobbat.

|                                                                                                                                                                                          |                                                                                                                                                          |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>(* maxl : int list -&gt; int    maxl ns = az ns egészlista              legnagyobb eleme *) fun maxl [] = raise Empty     maxl [n] = n     maxl (n::ns) = Int.max(n, maxl ns)</pre> | <pre>maxl egy változata egészszekre: (* max : int * int -&gt; int    max (n,m) = n és m közül a nagyobb *) fun max (n,m) = if n&gt;m then n else m</pre> |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------|

- `maxl` egy másik változata (az előzővel szemben itt a klózok sorrendje közömbös!):

```
fun maxl' [] = raise Empty
 | maxl' [n] = n
 | maxl' (n::m::ns) = maxl' (Int.max(n,m)::ns)
```

- `maxl'` *jobbrekurzív*, ezért (talán) hatékonyabb, mint `maxl`. De `maxl` szebb, pontosan tükrözi a definíciót (és a rekurzív gondolkodásmódot), a helyességét is könnyebb belátni.

## Polimorfizmus

- Nézzük az identitásfüggvényt: `fun id x = x`.
- Mi az `x` típusa? Tetszőleges, típusát *típusváltozó* jelöli: `val 'a id = fn : 'a -> 'a`.
- `id` *polimorf* függvényt jelöl, `x` és `id` *politípusú* nevek.
- A *percjellel* kezdődő típusnév (pl. `'a`, olvasd *alfa*): *típusváltozó*.
- Nézzük az egyenlőségfüggvényt: `fun eq (x, y) = x = y`.
- Mi az `x` és `y` típusa? Tetszőleges, típusát szintén *típusváltozó* jelöli: `val "a eq = fn : "a * "a -> bool`.
- A *két* percjellel kezdődő típusnév (pl. `"a`, olvasd *alfa*) az ún. *egyenlőségi típus változója*.

Polimorfizmus többféle változatban fordul elő a programozásban.

- Egy *polimorf név egyetlen* olyan algoritmust azonosít, amely tetszőleges típusú argumentumra alkalmazható; ez a *paraméteres polimorfizmus*.
- Egy *többszörösen terhelt név több különböző* algoritmust azonosít: ahány típusú argumentumra alkalmazható, annyiféle; ez az *ad-hoc* vagy *többszörös terheléses* polimorfizmus.
- A polimorfizmus harmadik változata az *öröklődéses polimorfizmus* (vö. objektum-orientált programozás).

## LOGIKAI MŰVELETEK

### Logikai műveletek

- Típusnév: `bool`, adatkonstruktorok: `false`, `true`, beépített függvény: `not`.
- *Lusta kiértékelésű* beépített operátorok
  - Három argumentumú: `if b then e1 else e2`.  
Nem értékeli ki az `e2`-t, ha `a b` igaz, ill. az `e1`-et, ha `a b` hamis.
  - Két argumentumúak:
    - `e1 andalso e2`: nem értékeli ki az `e2`-t, ha az `e1` hamis.
    - `e1 orelse e2`: nem értékeli ki az `e2`-t, ha az `e1` igaz.
- *Lusta kiértékelés* = kiértékelés szükség szerint (*call-by-need*): csak akkor értékeli ki az értelmező, amikor szükség van rá.
- Mind három logikai operátor csupán szintaktikus édesítőszer!
  - `if b then e1 else e2`  $\equiv$  `(fn true => e1 | false => e2) b`
  - `e1 andalso e2`  $\equiv$  `(fn true => e2 | false => false) e1`
  - `e1 orelse e2`  $\equiv$  `(fn true => true | false => e2) e1`
- Tipikus hiba: `if exp then true else false !!!`

### Logikai műveletek (folyt.)

- Nyilvánvaló: `andalso` és `orelse` kifejezhető `if-then-else`-szel is.
  - `if e1 then e2 else false`  $\equiv$  `e1 andalso e2`
  - `if e1 then true else e2`  $\equiv$  `e1 orelse e2`
- Használjuk az `andalso`-t és az `orelse`-t az `if-then-else` helyett, ahol csak lehet: olvashatóbb lesz a program.
- *Lusta kiértékelésű* függvényt a programozó nem definiálhat az SML-ben. Az SML, mielőtt egy függvényt alkalmazna az (egyszerű vagy összetett) argumentumára, kiértékeli.
- Az `andalso` és az `orelse` *mohó kiértékelésű* megfelelői:
 

|                                                                                                                                                 |                                                                                                                |
|-------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------|
| <pre>(* &amp;&amp; (a, b) = a /\ b    &amp;&amp; : bool * bool -&gt; bool    *) fun op&amp;&amp; (a, b) = a andalso b; infix 2 &amp;&amp;</pre> | <pre>(*    (a, b) = a \/ b       : bool * bool -&gt; bool    *) fun op   (a, b) = a orelse b; infix 1   </pre> |
|-------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------|
- `infix prec név1 név2 ... : a név1 név2 ...` függvényeket `prec` precedenciaszintű, *infix* helyzetű *operátorra* alakítja.

## LISTÁK

## Lista legnagyobb elemének megkeresése (folyt.)

- Hogyan tehető *polimorffá* a max1 függvény? Úgy, hogy ún. *generikus* függvényként definiáljuk: *aktuális paraméterként* kapja azt a többszörösen terhelhető függvényt, amely két érték közül a nagyobbikat kiválasztja.

```
(* max1 : ('a * 'a -> 'a) -> 'a list -> 'a
 max1 max zs = a zs lista max szerint legnagyobb eleme
*)
fun max1 max [] = raise Empty
 | max1 max [z] = z
 | max1 max (z::zs) = max(z, max1 max zs)
```

- max mindig ugyanaz, mégis újra és újra átadjuk argumentumként a rekurzív ágban. Javíthatja a hatékonyságot, ha *let-kifejezést* használunk.

```
fun max1 max zs = let fun mxl [] = raise Empty
 | mxl [y] = y
 | mxl (y::ys) = max(y, mxl ys)
 in
 mxl zs
 end
```

## ÖSSZETETT ADATTÍPUSOK

## Változatok max-ra

### Változatok max-ra

- (\* charMax : char \* char -> char
 charMax (a, b) = a és b közül a nagyobbik
 \*)
 fun charMax (a, b) = if ord a > ord b then a else b;
 vagy egyszerűen (ord nélkül)
 fun charMax (a : char, b) = if a > b then a else b;
- (\* pairMax : ((int \* real) \* (int \* real)) -> (int \* real)
 pairMax (n, m) = n és m közül lexicografikusan a nagyobbik
 \*)
 fun pairMax (n as (n1 : int, n2 : real), m as (m1, m2)) =
 if n1 > m1 orelse n1 = m1 andalso n2 >= m2 then n else m;
- (\* stringMax : string \* string -> string
 stringMax (s, t) = s és t közül a nagyobbik
 \*)
 fun stringMax (s : string, t) = if s > t then s else t;

## Rekord és ennes

- Két különböző típusú értékből rekordot vagy párt képezhetünk. Pl.
  $\{x = 2, y = 1.0\} : \{x : int, y : real\}$  és  $(2, 1.0) : (int * real)$
- A pár is csak szintaktikus édesítőszert. Pl.
  $(2, 1.0) \equiv \{1 = 2, 2 = 1.0\} \equiv \{2 = 1.0, 1 = 2\} \neq \{1 = 1.0, 2 = 2\}$ .  
 Egy párban a tagok sorrendje meghatározó! Az 1 és a 2 is: *mezőnevek*.
- Rekordot kettőnél több értékből is összeállíthatunk. Pl.
  $\{nev = "Bea", tel = 3192144, kor = 19\} : \{kor : int, nev : string, tel : int\}$   
 Egy hasonló rekord egészszám-mezőnevekkel:  
 $\{1 = "Bea", 3 = 3192144, 2 = 19\} : \{1 : string, 2 : int, 3 : int\}$   
 Az *utóbbi* azonos az alábbi *ennessel* (n-tuple):  
 $("Bea", 19, 3192144) : (string * int * int)$   
 azaz  
 $(string * int * int) \equiv \{1 = string, 2 = int, 3 = int\}$
- Egy rekordban a tagok sorrendje közömbös, a tagokat a mezőnév azonosítja. Egy ennesben a tagok sorrendje nem közömbös, a tagokat a *pozícionális* mezőnév azonosítja.

## GYENGE ÉS ERŐS ABSZTRAKCIÓ

### Adattípusok: gyenge és erős absztrakció

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

Aadatabsztrakció, adattípusok: type, datatype FP-51

### Adattípusok: felsorolásos és polimorf típusok datatype deklarációval

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

|             |                           |
|-------------|---------------------------|
| Igen : logi | Semmi : 'a esetleg        |
| Nem : logi  | Valami : 'a -> 'a esetleg |
- Példa datatype deklarációval létrehozott adattípust kezelő függvényre
 

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

Deklaratív programozás. BME VIK, 2003. tavaszi félév

(Funkcionális programozás)

## LISTÁK

## Listák összefűzése (append) és megfordítása (nrev)

- Két lista összefűzése (append, infix változatban @)

$$[x_1, \dots, x_m] @ [y_1, \dots, y_n] = [x_1, \dots, x_{m-1}] @ (x_m :: [y_1, \dots, y_n]) = \dots = [x_1, \dots, x_m, y_1, \dots, y_n]$$

Az `xs`-t először az elemeire bontjuk, majd hátulról visszafelé haladva fűzzük az elemeket az `ys`-hez, ugyanis a listákat csak előlről tudjuk építeni. A lépések száma  $O(n)$ .

```
(* append : 'a list * 'a list -> 'a list
 append xs, ys = xs összes eleme ys elé fűzve *)
fun append ([] , ys) = ys
 | append (x::xs, ys) = x::append(xs, ys)
```

- Lista naív megfordítása (nrev)

$$nrev[x_1, x_2, \dots, x_m] = nrev[x_2, \dots, x_m] @ [x_1] = nrev[\dots, x_m] @ [x_2] @ [x_1] = \dots = [x_m, \dots, x_1]$$

A lista elejéről levett elemet egyelemű listaként tudjuk a végéhez fűzni. A lépések száma  $O(n^2)$ .

```
(* nrev : 'a list -> 'a list
 nrev xs = xs megfordítva *)
fun nrev [] = []
 | nrev (x::xs) = (nrev xs) @ [x]
```

## Listák összefűzése (revApp) és megfordítása (rev)

- Egy lista elemeinek egy másik lista elé fűzése fordított sorrendben (revApp)

```
(* revApp : 'a list * 'a list -> 'a list
 revApp xs, ys = xs elemei fordított sorrendben ys elé fűzve
 *)
fun revApp ([] , ys) = ys
 | revApp (x::xs, ys) = revApp(xs, x::ys)
```

`revApp` lépésszáma arányos a lista hosszával. Segítségével `rev` hatékonyan:

```
(* rev : 'a list -> 'a list
 rev xs = xs megfordítva
 *)
fun rev xs = revApp (xs, [])
```

Egy 1000 elemű listát `rev` 1000 lépésben, `nrev`  $\frac{1000 \cdot 1001}{2} = 500500$  lépésben fordít meg. Hatalmas a nyereség!

- `append` – `@` néven, infix operátorként – és `rev` beépített függvények, `List.revApp` pedig `List.revAppend` néven könyvtári függvény az SML-ben.

## Listák megfordítása: példa nrev alkalmazására

- Egy példa `nrev` egyszerűsítésére

```
fun nrev [] = []
 | nrev (x::xs) = (nrev xs) @ [x]

fun [] @ ys = ys
 | (x::xs) @ ys = x :: xs @ ys (* = x :: (xs @ ys) *)
```

`A ::` és `@` jobbra kötnek, precedenciaszintjük 5.

```
nrev([1,2,3,4]) → nrev([2,3,4])@ [1] → nrev([3,4])@ [2]@ [1]
→ nrev([4])@ [3]@ [2]@ [1] → nrev([])@ [4]@ [3]@ [2]@ [1]
→ []@ [4]@ [3]@ [2]@ [1] → [4]@ [3]@ [2]@ [1]
→ 4:: []@ [3]@ [2]@ [1] → 4:: [3]@ [2]@ [1]
→ [4,3]@ [2]@ [1] → 4:: ([3]@ [2])@ [1]
→ []@ [4]@ (3:: [2,1]) → []@ [4]@ [3,2,1] → ...
```

`nrev` rossz hatékonyságú: a lépések száma  $O(n^2)$ .

## LISTÁK HASZNÁLATA

## Példák listák használatára: számtani sorozatok előállítása

### Naív változat

```
(* upto1 : int -> int -> int list
 upto1 m n = 1 különbségű növekvő számtani sorozat
 az alulról zárt [m,n) tartományban
*)
fun upto1 m n = if m >= n then [] else m :: upto1 (m+1) n;
```

### Naív változat, egy segédfüggvénnyel

```
(* upto2 : int -> int -> int list
 upto2 m n = 1 különbségű növekvő számtani sorozat
 az alulról zárt [m,n) tartományban
*)
fun upto2 m n =
 let fun up m = if m >= n then [] else m :: up (m+1)
 in
 up m
 end;
```

## Példák listák használatára: számtani sorozatok előállításá (folyt.)

### Akkumulátoros változat d különbséggel, egy segédfüggvénnyel

```
(* fnto3 : int -> int -> int -> int list
 fnto3 d n m = d>0 különbségű, m<n esetén növekvő, m>n esetén
 csökkenő számtani sorozat az alulról, ill. felülről zárt
 [m,n) tartományban
*)
fun fnto3 d m n =
 let fun to cmp inc m zs = if cmp m
 then rev zs
 else to cmp inc (inc m) (m::zs)
 in
 if d <= 0
 then []
 else if n >= m
 then to (fn i => i >= n) (fn j => j + d) m []
 else (* if n < m then *)
 to (fn i => i <= n) (fn j => j - d) m []
 end;
```

## Példák listák használatára: számtani sorozatok előállításá (folyt.)

### Naív változatok d különbséggel

```
(* fnto? : int -> int -> int -> int list
 fnto? d m n = d>0 különbségű, m<n esetén növekvő, m>n esetén csökkenő
 számtani sorozat az alulról, ill. felülről zárt [m,n) tartományban
*)
```

### ...két segédfüggvénnyel

```
fun fnto1 d m n =
 let fun up m = if n <= m then [] else m :: up (m+d)
 fun dn m = if m <= n then [] else m :: dn (m-d)
 in
 if d <= 0 then []
 else if n >= m then up m
 else (* if n < m then *) dn m
 end;
```

### ...egy segédfüggvénnyel

```
fun fnto2 d m n =
 let fun to cmp inc m = if cmp m then [] else m :: to cmp inc (inc m)
 in
 if d <= 0 then []
 else if n >= m then to (fn i => i >= n) (fn j => j + d) m
 else (* if n < m then *) to (fn i => i <= n) (fn j => j - d) m
 end;
```

## Példák listák használatára: számtani sorozatok előállításá (folyt.)

### Generikus akkumulátoros változat komparáló és különbségi függvénnyel

```
(* fnto4 : ('a * 'a -> bool) -> ('a -> 'a) -> 'a -> 'a -> 'a list
 fnto4 le df m n = az le kisebb-vagy-egyenlő reláció és a df
 különbségi függvény szerint növekvő sorozat az alulról,
 ill. felülről zárt [m,n) tartományban
*)
fun fnto4 le df m n =
 let fun to cmp m zs = if cmp m
 then rev zs
 else to cmp (df m) (m::zs);
 infix le
 in
 if m le df m andalso m le n (*df növel, [m,n)<>üres*)
 then to (fn i => n le i) m []
 else if df m le m andalso n le m (*df csökkent, [m,n)<>üres*)
 then to (fn i => i le n) m []
 else []
 end;
```

## Példák listák használatára: számtani sorozatok előállítása (folyt.)

### Egyszerű akkumulátoros változat, egy segédfüggvénnyel

Az egyesével növekvő sorozat utolsó eleme ismert (n-1), ezért a listát hátulról visszafelé építjük fel.

```
(* upto3 : int -> int -> int list
 upto3 m n = 1 különbségű növekvő számtani sorozat
 az alulról zárt [m,n) tartományban
*)
fun upto3 m n =
 let
 (* hátulról visszafelé haladva építjük a listát! *)
 fun up n zs = if m >= n then zs else up (n-1) (n-1::zs)
 in
 up n []
 end;

infix --;
fun fm -- to = upto3 fm to;
```

Gyakorló feladat: a d = +1 vagy -1 különbségű számtani sorozatot előállító upto4 d m n függvény megírása.

## FUTÁSI IDŐ MÉRÉSE

## Példák listák használatára: számtani sorozatok előállítás (folyt.)

```
use "upto.sml";

(* egyesével növekvő egészlisták *)

val ns50k = 1 -- 50000;
val ns150k = 50001 -- 200000;
val ns200k = 200001 -- 400000;
val ns400k = 200001 -- 600000;
val ns600k = 600001 -- 1200000;
val ns1M = 100001 -- 1100000;
val ns2M = 100001 -- 2100000;

(* más sorozatok *)

(* 1.5-del csökkenő, 666667 elemű valóslista *)
val rs666667 = fmto4 op>= (fn i => i-1.5) 1000000.0 0.0;
(* #"A"-tól kezdve minden 2. betűt tartalmazó 13 elemű lista *)
val cs13 = fmto4 op<= (fn i => chr(ord i + 2)) #"A", #"Z";
```

## Futási idő mérése PolyML alatt

### ● Akkumulátor nélküli, naív változatok (PolyML: "Increasing stack")

```
PolyML.timing(true);
```

```
val _ = upto1 1 1000000; (* +1 különbségű növekvő sorozat *)
val _ = upto2 1 1000000; (* +1 különbségű növekvő sorozat, 1 segédfv. *)
val _ = fmto1 1 1 1000000; (* +d különbségű növekvő/csökkenő, 1 segédfv. *)
val _ = fmto2 1 1 1000000; (* +d különbségű növekvő/csökkenő, 2 segédfv. *)
```

```
PolyML.timing(false);
```

### Részletek a PolyML válaszából:

```
Warning - Increasing stack from 147456 to 294912 bytes
...
Timing - parse:0.0, semantics:0.0, translate:0.0, generate:0.0, run:1.9
Timing - parse:0.0, semantics:0.0, translate:0.0, generate:0.0, run:1.4
Warning - Increasing stack from 9437184 to 18874368 bytes
Timing - parse:0.0, semantics:0.0, translate:0.0, generate:0.0, run:2.2
Timing - parse:0.0, semantics:0.0, translate:0.0, generate:0.0, run:1.5
```



## Futási idő mérése PolyML alatt (folyt.)

- Akkumulátoros változatok

```
PolyML.timing(true);
```

```
val _ = upto3 1 1000000; (* +1 különbségű növekvő sorozat, végétől kezd *)
val _ = finto3 1 1 1000000; (* +d különbségű növekvő/csökkenő, 1 segédfv. *)
val _ = finto4 op<= (fn i => i+1) 1 1000000; (* +d kül., gen. *)
val _ = finto4 op<= (fn i => i+1.0) 1.0 1000000.0; (* +d kül., gen. *)
```

```
PolyML.timing(false);
```

Részletek a PolyML válaszából:

```
Timing - parse:0.0, semantics:0.0, translate:0.0, generate:0.0, run:0.8
Timing - parse:0.0, semantics:0.0, translate:0.0, generate:0.0, run:2.4
Timing - parse:0.0, semantics:0.0, translate:0.0, generate:0.0, run:2.2
Timing - parse:0.0, semantics:0.0, translate:0.0, generate:0.0, run:4.6
```

## FONTOS APRÓSÁGOK

## Fontos apróságok az SML-ről

- A nullas és a unit típus

A () vagy {} jelet *nullasnak* nevezzük, típusa: unit. A nullas a unit típus egyetlen eleme. A unit típusműveletek *egységeleme*.

- A print függvény

Ha a string -> unit típusú print függvényt egy füzérre alkalmazzuk, eredménye a *nullas*, *mellékhatásként* pedig kiírja a a füzér értékét.

- Az (e1; e2; e3) szekvenciális kifejezés *eredménye* azonos az e3 kifejezés eredményével.

Ha az e1 és e2 kifejezéseknek van mellékhatásuk, az érvényesül. (e1; e2; e3) egyenértékű a következő let-kifejezéssel:

```
let val _ = e1 val _ = e2 in e3 end
```

- Az e1 before e2 before e3 kifejezés *eredménye* azonos az e1 kifejezés eredményével.

Ha az e2 és e3 kifejezésnek van mellékhatása, az érvényesül. e1 before e2 before e3 egyenértékű a következő let-kifejezéssel:

```
let val e = e1 val _ = e2 val _ = e3 in e end
```

## FUTÁSI IDŐ MÉRÉSE

## Futási idő mérése: Timer – SML Basis Library

---

```

type cpu_timer
type real_timer
val startCPUTimer : unit -> cpu_timer
val checkCPUTimer : cpu_timer ->
 {usr: Time.time, sys: Time.time, gc: Time.time}

[cpu_timer] is the type of timers for measuring CPU time
consumption (user time, garbage collection time, and system time).
[real_timer] is the type of timers for measuring the passing of
real time (wall-clock time).
[startCPUTimer ()] returns a cpu_timer started at the moment of
the call.
[checkCPUTimer tmr] returns {usr, sys, gc} where usr is the amount
of user CPU time consumed since tmr was started, gc is the amount
of user CPU time spent on garbage collection, and sys is the
amount of system CPU time consumed since tmr was started. Note
that gc time is included in the usr time. Under MS DOS, usr time
and gc time are measured in real time.

```

Deklaratív programozás. BME VIK, 2003. tavaszi félév

(Funkcionális programozás)

## Futási idő mérése: idomero

---

```

(* app load ["Timer", "Time"]; *)

(* idomero = fn : ('a -> 'b) -> 'a -> 'b * time * time
 idomero f a = az a argumentumra alkalmazott f függvény
 eredményéből, továbbá a kiértékeléséhez felhasznált CPU-időt
 és valós időt másodpercben tartalmazó füzérekből álló hármas
 *)
fun idomero f a =
 let
 val RealStart = Timer.startRealTimer()
 val CPUstart = Timer.startCPUTimer()
 val res = f a
 val {usr, ...} = Timer.checkCPUTimer CPUstart
 val rtm = Timer.checkRealTimer RealStart
 in
 (res, usr, rtm)
 end;

```

Deklaratív programozás. BME VIK, 2003. tavaszi félév

(Funkcionális programozás)

## Futási idő mérése: Timer és Time – SML Basis Library (folyt.)

---

```

val startRealTimer : unit -> real_timer
val checkRealTimer : real_timer -> Time.time

[startRealTimer ()] returns a real_timer started at the moment of
the call.
[checkRealTimer tmr] returns the amount of real time that has passed
since tmr was started.

eqtype time
exception Time
val fmt : int -> time -> string

[time] is a type for representing durations as well as absolute
points in time (which can be thought of as durations since some
fixed time zero).
[zeroTime] represents the 0-second duration, and the origin of time,
so zeroTime + t = t + zeroTime = t for all t.
[fmt n t] returns as a string the number of seconds represented by
t, rounded to n decimal digits. If n <= 0, then no decimal digits
are reported.

```

Deklaratív programozás. BME VIK, 2003. tavaszi félév

(Funkcionális programozás)

## Futási idő mérése: idoiro

---

```

(* idoiro = fn : string -> ('a -> 'b) -> 'a -> 'b
 idoiro t f a = f a
 mellékhatásként kiírja a t füzért, az a argumentumra
 alkalmazott f függvény kiértékeléséhez felhasznált CPU-időt
 és valós időt (mp-ben)
 *)
fun idoiro t f a =
 let val (res, usr, rtm) = idomero f a
 in
 (print (t ^ " CPU time: " ^ Time.fmt 2 usr ^
 ", real time: " ^ Time.fmt 2 rtm ^ " (sec)\n");
 res
)
 end;

```

Deklaratív programozás. BME VIK, 2003. tavaszi félév

(Funkcionális programozás)

## LISTÁK HASZNÁLATA

### Példák listák használatára: max1 még több változatban

```
(* app load ["Int"]; *)
use "idomero.sml";
use "szamlistak.sml";

(* max1? : int list -> int
 max1? ns = az ns egészlista legnagyobb eleme
*)

fun max1l [n] = n
 | max1l (n::ns) =
 Int.max(n, max1l ns)
 | max1l [] = raise Empty;

fun max12 [n] = n
 | max12 (n::m::ns) =
 max12(Int.max(n, m)::ns)
 | max12 [] = raise Empty;

fun max13 [] = raise Empty
 | max13 [n] = n
 | max13 (n::m::ns) =
 max13(Int.max(n, m)::ns);

fun max14 [] = raise Empty
 | max14 ns =
 let (* [] esete lefedetlen! *)
 fun mxl [n] = n
 | mxl (n::m::ns) =
 mxl(Int.max(n, m)::ns)
 in mxl ns
 end;
 let fun mxl5 [] = raise Empty
 | mxl5 (n::ns) =
 let fun mxl (m, []) = m
 | mxl (m, n::ns) =
 mxl(Int.max(m, n), ns)
 in mxl(n, ns)
 end;
 end;
```

Deklaratív programozás. BME VIK, 2003. tavaszi félév

(Funkcionális programozás)

Listák használata: max1 több változatban FP-75

### Példák listák használatára: max1 még több változatban (folyt.)

```
(* max1? : ('a*'a->'a) * 'a list -> 'a
 max1? (max, ns) = az ns lista max
 szerinti legnagyobb eleme
*)
fun max16 (max, []) = raise Empty
 | max16 (max, [n]) = n
 | max16 (max, n::m::ns) =
 max16(max, max(n, m)::ns);

fun max17 (max, []) = raise Empty
 | max17 (max, ns) =
 let (* [] esete lefedetlen! *)
 fun mxl [n] = n
 | mxl (n::m::ns) =
 mxl(max(n, m)::ns)
 in mxl ns
 end;
 let fun max18 (max, []) = raise Empty
 | max18 (max, n::ns) =
 let fun mxl (m, []) = m
 | mxl (m, n::ns) =
 mxl(max(m, n), ns)
 in mxl(n, ns)
 end;
 end;
```

```
datatype 'a esetleg = Semmi
 | Valami of 'a;
(* max19 : ('a*'a->'a) * 'a list
 -> 'a esetleg
 max19 (max, ns) = az ns lista max
 szerinti legnagyobb eleme
*)
fun max19 (max, []) = Semmi
 | max19 (max, n::ns) =
 let fun mxl (m, []) = Valami m
 | mxl (m, n::ns) =
 mxl(max(m, n), ns)
 in mxl(n, ns)
 end;
```

- **Példák max1 alkalmazására:**

```
max17(Int.max, [1,3,9,7,4,6,2]) = 9;
max18(op+, [1,2,3,4,5,6,7,8,9]) = 45;
max19(op*, [1,2,3,4,5,6,7,8,9]) =
 Valami 36288
```
- **Furcsa „maximum” max18(...) és max19(...) eredménye!**

Deklaratív programozás. BME VIK, 2003. tavaszi félév

(Funkcionális programozás)

Listák használata: max1 több változatban FP-76

### max1 futási idejének mérése idoiro-val

```
idoiro "max11, ns2M: " max11 ns2M;
idoiro "max12, ns2M: " max12 ns2M;
idoiro "max13, ns2M: " max13 ns2M;
idoiro "max14, ns2M: " max14 ns2M;
idoiro "max15, ns2M: " max15 ns2M;
idoiro "max16, ns2M: " max16(Int.max, ns2M);
idoiro "max17, ns2M: " max17(Int.max, ns2M);
idoiro "max18, ns2M: " max18(Int.max, ns2M);
idoiro "max19, ns2M: " max19(Int.max, ns2M);
```

#### Részletek a PolyML válaszból:

```
Warning - Increasing stack from 9437184 to 18874368 bytes
max11, ns2M: CPU time: 4.57, real time: 7.99 (sec)
max12, ns2M: CPU time: 0.73, real time: 1.00 (sec)
max13, ns2M: CPU time: 0.80, real time: 0.80 (sec)
max14, ns2M: CPU time: 0.83, real time: 0.83 (sec)
max15, ns2M: CPU time: 0.14, real time: 0.16 (sec) !!!
max16, ns2M: CPU time: 1.41, real time: 1.41 (sec)
max17, ns2M: CPU time: 1.54, real time: 1.55 (sec)
max18, ns2M: CPU time: 0.85, real time: 0.85 (sec)
val it = 2099999 : Int.int
max19, ns2M: CPU time: 0.86, real time: 0.85 (sec)
val it = Valami 2099999 : Int.int esetleg
```

Deklaratív programozás. BME VIK, 2003. tavaszi félév

(Funkcionális programozás)

## LISTÁK HASZNÁLATA

Listák: futamok előállítása FP-79

### Példák listák használatára: futamok előállítása (folyt.)

- Példák:

```
futam1 op<= (1, [9,19,3,4,24,34,4,11,45,66,13,45,66,99]);
maradek op<= (1, [9,19,3,4,24,34,4,11,45,66,13,45,66,99]);
futamok1 op<= [1,9,19,3,4,24,34,4,11,45,66,13,45,66,99];
futamok1 op<= [99,1];
futamok1 op<= [99];
futamok1 op<= [];
```

- Hatékonyságot rontó tényezők

1. futamok1 kétszer megy végig a listán: először futam1, azután maradek,
2. p-t paraméterként adjuk át futam1-nek és maradek-nak,
3. egyik függvény sem használ akkumulátort.

- Javítási lehetőség

1. futam2 egy párt adjon eredményül, ennek első tagja legyen a futam, második tagja pedig a maradek; a futam elemeinek gyűjtésére használjunk akkumulátort,
2. futam2 legyen lokális futamok2-n belül,

### Példák listák használatára: futamok előállítása

A futam olyan elemekből álló lista, amelynek szomszédos elemei kielégítenek egy predikátumot. Írjon olyan SML függvényt futamok néven, amelynek egy lista futamaiból álló lista az eredménye.

- Első változat: futam és maradek előállítása két függvénnyel

```
(* futam1 : ('a * 'a -> bool) -> ('a * 'a list) -> 'a list
 futam1 p (x, ys) = az x::ys p-t kielégítő első futama (prefixuma *)
fun futam1 p (x, []) = [x]
 | futam1 p (x, y::ys) = if p(x, y) then x :: futam1 p (y, ys) else [x]

(* maradek : ('a * 'a -> bool) -> ('a * 'a list) -> 'a list
 maradek p (x, ys) = az x::ys p-t kielégítő futama utáni maradéka *)
fun maradek p (x, []) = []
 | maradek p (x, y::ys) = if p(x, y) then maradek p (y, ys) else y::ys

(* futamok1 : ('a * 'a -> bool) -> 'a list -> 'a list list
 futamok1 p xs = az xs p-t kielégítő futamaiból álló lista *)
fun futamok1 p [] = []
 | futamok1 p (x::xs) =
 let val fs = futam1 p (x, xs)
 val ms = maradek p (x, xs)
 in
 if null ms then [fs] else fs :: futamok1 p ms
 end;
```

Deklaratív programozás. BME VIK, 2003. tavaszi félév

(Funkcionális programozás)

Listák: futamok előállítása FP-80

### Példák listák használatára: futamok előállítása (folyt.)

- Második változat: futam és maradek előállítása egy függvénnyel

```
(* futam2 : ('a * 'a -> bool) -> ('a * 'a list) -> 'a list * 'a list

 futam2 p (x, ys) zs = olyan pár, amelynek első tagja az x::ys p-t
 kielégítő első futama (prefixuma) a zs elé fűzve,
 második tagja pedig az x::ys maradéka
*)
fun futam2 p (x, []) zs = (rev(x::zs), [])
 | futam2 p (x, y::ys) zs = if p(x, y)
 then futam2 p (y, ys) (x::zs)
 else (rev(x::zs), y::ys);

(* futamok2 : ('a * 'a -> bool) -> 'a list -> 'a list list
 futamok2 p xs = az xs p-t kielégítő futamaiból álló lista
*)
fun futamok2 p [] = []
 | futamok2 p (x::xs) =
 let val (fs, ms) = futam2 p (x, xs) []
 in
 if null ms then [fs] else fs :: futamok2 p ms
 end;
```

## Példák listák használatára: futamok előállítása (folyt.)

- Harmadik változat: futam és maradék előállítása egy lokális függvénnyel

```
(* futamok3 : ('a * 'a -> bool) -> 'a list -> 'a list list
 futamok3 p xs = az xs p-t kielégítő futamaiból álló lista
*)
fun futamok3 p [] = []
 | futamok3 p (x::xs) =
 let (* futam : ('a * 'a list) -> 'a list * 'a list
 futam (x, ys) zs = olyan pár, amelynek első tagja az x::ys p-t
 kielégítő első futama (prefixuma) a zs elé
 fűzve, második tagja pedig az x::ys maradéka
 *)
 fun futam (x, []) zs = (rev(x::zs), [])
 | futam (x, yys as y::ys) zs = if p(x, y)
 then futam (y, ys) (x::zs)
 else (rev(x::zs), yys);
 val (fs, ms) = futam (x, xs) []
 in
 if null ms then [fs] else fs :: futamok3 p ms
 end;
```

## LISTÁK

## Példák listák használatára: futamok előállítása (folyt.)

- Negyedik változat: az egyes futamokat és a futamok listáját is gyűjtjük

```
(* futamok4 : ('a * 'a -> bool) -> 'a list -> 'a list list
 futamok4 p xs = az xs p-t kielégítő futamaiból álló lista
*)
fun futamok4 p [] = []
 | futamok4 p (x::xs) =
 let (* futamok : ('a * 'a list) -> 'a list -> 'a list * 'a list
 futamok (x, ys) zss = az x::ys p-t kielégítő futamaiból álló
 lista zss elé fűzve
 *)
 fun futamok (x, []) zss = rev(rev(x::zs)::zss)
 | futamok (x, yys as y::ys) zss =
 if p(x, y)
 then futamok (y, ys) (x::zs) zss
 else futamok (y, ys) [] (rev(x::zs)::zss)
 in
 futamok (x, xs) [] []
 end;
```

## Adott számú elem egy lista elejéről és végéről (take, drop)

- Legyen  $xs = [x_0, x_1, \dots, x_{i-1}, x_i, x_{i+1}, \dots, x_{n-1}]$ , akkor

$take(xs, i) = [x_0, x_1, \dots, x_{i-1}]$  és  $drop(xs, i) = [x_i, x_{i+1}, \dots, x_{n-1}]$ .

```
(* take : 'a list * int -> 'a list
 take (xs, i) = xs, ha i < 0; az xs első i db eleméből
 álló lista, ha i >= 0
*)
fun take (_, 0) = []
 | take ([], _) = []
 | take (x::xs, i) = x :: take(xs, i-1);

(* drop : 'a list * int -> 'a list
 drop(xs, i) = xs, ha i < 0; az xs első i db elemének
 eldobásával előálló lista, ha i >= 0
*)
fun drop ([], _) = []
 | drop (x::xs, i) = if i > 0 then drop (xs, i-1) else x::xs;
```

- Könyvtári változatok, `List.take`, ill. `List.drop`, ha az `xs` listára alkalmazzuk,  $i < 0$  vagy  $i > \text{length } xs$  esetén `Subscript` néven kivételt jelez.

## Lista redukciója kétoperandusú művelettel (foldr, foldl)

- Vissza-visszatérő feladat egy lista redukciója kétoperandusú művelettel. Közös, hogy  $n$  db értékből egyetlen értéket kell előállítani (vö. *redukció*).
- `foldr` jobbról balra, `foldl` balról jobbra haladva egy kétoperandusú műveletet (pontosabban egy *párra alkalmazható, prefix* pozíciójú függvényt) alkalmaz egy listára. Példák szorzat és összeg kiszámítására:

```
foldr op* 1.0 [] = 1.0; foldl op+ 0 [] = 0;
foldr op* 1.0 [4.0] = 4.0; foldl op+ 0 [4] = 4;
foldr op* 1.0 [1.0, 2.0, 3.0, 4.0] = 24.0; foldl op+ 0 [1, 2, 3, 4] = 10;
```

- Jelöljön  $\oplus$  tetszőleges kétoperandusú infix operátort. Akkor
 
$$\text{foldr } \text{op}\oplus \text{ e } [x_1, x_2, \dots, x_n] = (x_1 \oplus (x_2 \oplus \dots \oplus (x_n \oplus \text{e}) \dots))$$

$$\text{foldr } \text{op}\oplus \text{ e } [] = \text{e}$$

$$\text{foldl } \text{op}\oplus \text{ e } [x_1, x_2, \dots, x_n] = (x_n \oplus \dots \oplus (x_2 \oplus (x_1 \oplus \text{e}) \dots))$$

$$\text{foldl } \text{op}\oplus \text{ e } [] = \text{e}$$
- Asszociatív műveleteknél `foldr` és `foldl` eredménye azonos.

## LISTA REDUKCIÓJA

### Példák foldr és foldl alkalmazására

- $A \oplus$  művelet  $e$  operandusa néhány gyakori műveletben – összeadás, szorzás, konjunkció (logikai „és”), alternáció (logikai „vagy”) – a (jobb oldali) *egységelem* szerepét tölti be.
- `isum` egy egészlista elemeinek összegét, `rprod` egy valószínűségi lista elemeinek szorzatát adja eredményül.
- A `length` függvény is definiálható `foldl`-lel vagy `foldr`-rel. Kétoperandusú műveletként olyan segédfüggvényt (`inc`) alkalmazunk, amelyik *nem használja* az első paraméterét.

```
val isum = foldr op+ 0; val rprod = foldr op* 1.0;
val isum = foldl op+ 0; val rprod = foldl op* 1.0;

(* inc : 'a * int -> int
 inc (_, n) = n + 1 *)
fun inc (_, n) = n + 1;
(* lengthl, lengthr : 'a list -> int *)
val lengthl = fn ls => foldl inc 0 ls;
fun lengthr ls = foldr inc 0 ls;

lengthl (explode "tengertanc");
lengthr (explode "hajdu sogor");
```

### Lista: foldr és foldl definíciója

- $\text{foldr } \text{op}\oplus \text{ e } [x_1, x_2, \dots, x_n] = (x_1 \oplus (x_2 \oplus \dots \oplus (x_n \oplus \text{e}) \dots))$   
 $\text{foldr } \text{op}\oplus \text{ e } [] = \text{e}$ 

```
(* foldr f e xs = az xs elemeire jobbról balra haladva
 alkalmazott, kétoperandusú, e egységelemű
 f művelet eredménye
 foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b *)
fun foldr f e (x::xs) = f(x, foldr f e xs)
 | foldr f e [] = e;
```
- $\text{foldl } \text{op}\oplus \text{ e } [x_1, x_2, \dots, x_n] = (x_n \oplus \dots \oplus (x_2 \oplus (x_1 \oplus \text{e}) \dots))$   
 $\text{foldl } \text{op}\oplus \text{ e } [] = \text{e}$ 

```
(* foldl f e xs = az xs elemeire balról jobbra haladva
 alkalmazott, kétoperandusú, e egységelemű
 f művelet eredménye
 foldl : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b *)
fun foldl f e (x::xs) = foldl f (f(x, e)) xs
 | foldl f e [] = e;
```

## További példák foldr és foldl alkalmazására

- Egy lista elemeit egy másik lista elé fűzi foldr és foldl, ha kétoperandusú műveletként a cons konstruktorfüggvényt – azaz az op :: -ot – alkalmazzuk.

```
foldr op :: ys [x1, x2, x3] = (x1 :: (x2 :: (x3 :: ys)))
```

```
foldl op :: ys [x1, x2, x3] = (x3 :: (x2 :: (x1 :: ys)))
```

- A :: nem asszociatív, ezért foldl és foldr eredménye különböző!

```
(* append : 'a list -> 'a list -> 'a list
 append xs ys = az xs ys elé fűzésével előálló lista *)
fun append xs ys = foldr op :: ys xs;
```

```
(* revApp : 'a list -> 'a list -> 'a list
 revApp xs ys = a megfordított xs ys elé fűzésével
 előálló lista *)
fun revApp xs ys = foldl op :: ys xs;
```

```
append [1, 2, 3] [4, 5, 6] = [1, 2, 3, 4, 5, 6]; (vö. Prolog: append)
```

```
revApp [1, 2, 3] [4, 5, 6] = [3, 2, 1, 4, 5, 6]; (vö. Prolog: revapp)
```

## Lista redukciója bal oldali egységelemű függvénnyel (foldL)

- A kivonás művelete balra köt:  $x_1 - x_2 - x_3 - x_4 = ((x_1 - x_2) - x_3) - x_4$ .
  - Nem feleltethető meg sem foldr-nek, sem foldl-nek.
- ```
foldr op⊕ e [x1, x2, ..., xn] = (x1 ⊕ (x2 ⊕ ... ⊕ (xn ⊕ e) ...))
foldl op⊕ e [x1, x2, ..., xn] = (xn ⊕ ... ⊕ (x2 ⊕ (x1 ⊕ e)) ...)
```
- Nevezzük foldL-nek a listában balról jobbra haladó, alábbi specifikációjú függvényt. Vegyük észre, hogy ⊕ bal oldali egységelemet vár.

```
foldL op⊕ e [x1, x2, ..., xn] =
  (... ((e ⊕ x1) ⊕ x2) ⊕ ... ⊕ xn)
```

- foldL olyan kétargumentumú függvényt vár, amelynek az „egységelem” (valójában: a részeredmény) az első argumentuma: $f : 'a * 'b \rightarrow 'a$.

```
(* foldL : ('a * 'b -> 'a) -> 'a -> 'b list -> 'a
   foldL f e xs = az xs elemeire balról jobbra haladva
                 alkalmazott, kétoperandusú, e egységelemű
                 f művelet eredménye *)
```

```
fun foldL f e (x::xs) = foldL f (f(e, x)) xs
  | foldL f e [] = e;
```

További példák foldr és foldl alkalmazására

- maxl két további megvalósítása

```
datatype 'a esetleg = Semmi | Valami of 'a
```

```
(* maxl10 : ('a * 'a -> 'a) -> 'a list -> 'a esetleg
   maxl10 max ns = az ns lista max szerinti legnagyobb eleme *)
```

```
fun maxl10 max [] = Semmi
  | maxl10 max (n::ns) = Valami(foldl max n ns)
```

```
(* maxl11 : ('a * 'a -> 'a) -> 'a list -> 'a esetleg
   maxl11 max ns = az ns lista max szerinti legnagyobb eleme *)
```

```
fun maxl11 max [] = Semmi
  | maxl11 max (n::ns) = Valami(foldr max n ns)
```

```
idoiro "maxl10, ns200k, foldl-rel: " maxl10 Int.max ns200k;
```

```
idoiro "maxl11, ns200k, foldr-rel: " maxl11 Int.max ns200k;
```

```
idoiro "maxl10, ns2M, foldl-rel: " maxl10 Int.max ns2M;
```

Példák listaelemek különbségének és hányadosának képzésére

- Az e argumentum aktuális értéke a sorozat első eleme – a kisebbítendő, ill. az osztandó.

```
foldL op- 20 [] = 20;
```

```
foldL op- 20 [5, 6, 7] = (((20 - 5) - 6) - 7);
```

```
foldL (op div) 180 [] = 180;
```

```
foldL (op div) 180 [2, 3, 5] = (((180 div 2) div 3) div 5);
```

- Ha többször használjuk e műveleteket, érdemes nekik nevet adni. A kisebbítendő, ill. az osztandó speciális kezelését elrejtjük.

```
fun subtract ns = foldL op- (hd ns) (tl ns);
subtract [20, 5, 6, 7] = (((20 - 5) - 6) - 7);
```

```
fun divide ns = foldL op div (hd ns) (tl ns);
divide [180, 2, 3, 5] = (((180 div 2) div 3) div 5);
```

Listaelemek különbsége és hányadosa foldl-lel és foldr-rel

- Igazság szerint foldL felesleges: a feladat jól megoldható foldl-lel vagy foldr-rel is.

```
fun subtract1 ns = hd ns - foldl op+ 0 (tl ns);
subtract1 [20, 5, 6, 7] = ((20 - 5) - 6) - 7;
```

```
fun divide1 ns = hd ns div foldl op* 1 (tl ns);
divide1 [180, 2, 3, 5] = (((180 div 2) div 3) div 5);
```

- foldr és foldl típusa, ha egyparaméteres függvénynek tekintjük őket (a -> jobbra köt!):
 foldr, foldl : ('a * 'b -> 'b) -> ('b -> 'a list -> 'b)
 Azaz ha foldr-t vagy foldl-t egy 'a -> * 'b -> 'b típusú függvényre alkalmazzuk, akkor olyan függvényt ad eredményül, amelyet egy 'b típusú egységelemre és egy 'a list típusú listára alkalmazva 'b típusú (redukált) értéket kapunk.

ÖSSZETETT ADATTÍPUSOK

Felhasználói adattípusok: ismét a datatype deklarációról

- person néven új összetett típust hozunk létre:

```
datatype person = King
  | Peer of string * string * int
  | Knight of string
  | Peasant of string
```

- Az új típusnak négy *adatkonstruktor* (röviden: *konstruktor*) van: King, Peer, Knight és Peasant.
- King ún. *adatkonstruktorállandó*, a többi ún. *adatkonstruktorfüggvény*.
- Az adatkonstruktoroknak is van típusuk:

```
King :    person
Peer :    string * string * int -> person
Knight :  string -> person
Peasant : string -> person
```

A datatype deklaráció (folyt.)

```
King :    person
Peer :    string * string * int -> person
Knight :  string -> person
Peasant : string -> person
```

- King (király) csak egy van, ezért definiálhattuk konstruktorállandóként.
- A Peer-t (főnemes) nemesi címe (string), birtokának neve (string) és sorszáma (int) azonosítja.
- A Knight-ot (lovagot) és a Peasant-ot (parasztot) csupán a neve (string) azonosítja.
- Példa a person adattípus alkalmazására:


```
- val persons = [King, Peasant "Jack Cade", Knight "Gawain",
                  Peer("Duke", "Norfolk", 9)];
> val persons = [King, Peasant "Jack Cade", ...] : person list
```
- Az egyes esetek mintaillesztéssel választhatók szét.
- Minden esetet le kell fedni mintával; ha nem, figyelmeztetést kapunk.
- A minták tetszőlegesen összetettek lehetnek.

A datatype deklaráció (folyt.)

- Az alábbi példában a négy közül az egyik a `Peasant name` *minta*, és benne `name` a *mintaazonosító*.

```
(* title : person -> string
   title p = p megszólítása *)
fun title King = "His Majesty the King "
  | title (Peer (deg, ter, _)) = "The " ^ deg ^ " of " ^ ter
  | title (Knight name) = "Sir " ^ name
  | title (Peasant name) = name
```

- A `sirs` függvény az összes `Knight` nevét összegyűjti a `person` típusú személyek egy listájából (a változatok sorrendje *fontos* az `_` miatt!):

```
(* sirs : person list -> string list
   sirs ps = az összes Knight nevének listája *)
fun sirs [] = []
  | sirs ((Knight s)::ps) = s::sirs ps
  | sirs (_::ps) = sirs ps
```

A datatype deklaráció (folyt.)

- A sorrend még fontosabb a következő példában, amelyben személyek hierarchiáját vizsgáljuk. Itt 16 helyett csak 7 esetet kell megkülönböztetnünk: azokat, amelyek *igaz* eredményt adnak.

```
(* superior : person * person -> bool
   superior (p, r) = igaz, ha p magasabb rangú r-nél *)
fun superior (King, Peer _) = true
  | superior (King, Knight _) = true
  | superior (King, Peasant _) = true
  | superior (Peer _, Knight _) = true
  | superior (Peer _, Peasant _) = true
  | superior (Knight _, Peasant _) = true
  | superior _ = false
```

A datatype deklaráció (folyt.)

- Ha más lenne a változatok sorrendje, a `_::ps` minta nemcsak a `King`-re, a `Peer`-re és a `Peasant`-ra illeszkedne (ti. ezek helyett áll a példában), hanem a `Knight`-ra is.
- Az összes diszjunkt eset felsorolása segíti az algoritmus helyességének belátását, bizonyítását.
- Azért vontunk össze három esetet egyetlen változatban, mert a részletezésük hosszabbá tenné a program szövegét is, végrehajtását is.
- A bizonyítás nem okoz gondot, ha a függvény harmadik sorát (`sirs (_::ps) = sirs ps`) *feltételes egyenletek* tekintjük:

$$\text{sirs}(p::ps) = \text{sirs } ps \text{ if } \forall s.p \neq \text{Knight } s.$$

Felsorolásos típus datatype deklarációval

- Gyakori, hogy egy név csak néhány különböző értéket vehet fel (azaz a név által felvehető értékek halmaza kis számosságú), ilyen esetben érdemes *felsorolásos típust* létrehozni a `datatype` deklarációval. Pl.

```
datatype degree = Duke | Marquis | Earl | Viscount | Baron
```

- A felsorolásos típusnak csak *konstruktorállandói* vannak. Az új típus alkalmazásához a `person` típust újra deklarálnunk kell:

```
datatype person = King
  | Pear of degree * string * int
  | Knight of string
  | Peasant of string
```

Felsorolásos típus datatype deklarációval (folyt.)

- A degree típusú adatok feldolgozásakor külön-külön elemezzük az előforduló eseteket, pl.

```
(* lady : degree -> string
   lady p = p főnemes hitvesének rangja *)
fun lady Duke    = "Duchess "
  | lady Marquis = "Marchioness"
  | lady Earl    = "Countess"
  | lady Viscount = "Viscountess"
  | lady Baron   = "Baroness"
```

- A belső bool típushoz hasonló Bool típust és hozzá a Not függvényt például így is deklarálhatnánk, ill. definiálhatnánk:

```
datatype Bool = True | False
(* Not : Bool -> Bool
   Not b = b negáltja *)
fun Not True = False | Not False = True
```

Polimorf adattípusok: megkülönböztetett egyesítés

- Következő példánk két típus *megkülönböztetett egyesítése*, más néven diszjunkt uniója:

```
datatype ('a, 'b) disun = In1 of 'a | In2 of 'b
```

- Itt három dolgot definiáltunk:

1. a kétargumentumú disun típusoperátort,
2. az In1 : 'a -> ('a, 'b) disun és
3. az In2 : 'b -> ('a, 'b) disun adatkonstruktorfüggvényeket.

- ('a, 'b) disun az 'a és 'b típusok megkülönböztetett egyesítése. *Megkülönböztetettnek* nevezzük az egyesítést, mert később is bármikor meg tudjuk mondani, hogy egy ('a, 'b) disun típusú pár egyik vagy másik eleme melyik alaptípusból származik. Az új típusba tartozó értékek In1 x alakúak, ha x 'a típusú, és In2 y alakúak, ha y 'b típusú.
- Az In1 és In2 konstruktorfüggvények olyan *címkének* tekinthetők, amelyek az 'a típust megkülönböztetik a 'b típustól.

Polimorf adattípusok

- Láttuk, hogy a list postfix pozíciójú *típusoperátor*, nem típus: a datatype deklaráció az adatkonstruktorok mellett *típuskonstruktor* is létrehoz.

- A belső 'a list típushoz hasonló 'a List listát és vele együtt a Nil és a Cons *adatkonstruktorokat* például így definiálhatjuk:

```
datatype 'a List = Nil | Cons of 'a * 'a List
```

- A Cons *adatkonstruktorfüggvény* alkalmazásával elég körülményes a listák létrehozása. Az 1, 2, 3, 4 sorozatot például így kell megadni:

```
Cons(1, Cons(2, Cons(3, Cons(4, Nil))))
```

- Bevezethetjük az infix pozíciójú :: *adatkonstruktoroperátort*:

```
infix 5 :: ; val op :: = Cons
```

- A *hatospontot* közvetlenül a típusdeklarációban is definiálhatjuk:

```
infix 5 :: ; datatype 'a List = Nil | :: of 'a * 'a List
```

Megkülönböztetett egyesítés (folyt.)

- A megkülönböztetett egyesítés lehetővé teszi, hogy különböző típusokat használjunk ott, ahol egyébként csak egyetlen típust használhatnánk (vö. objektum-orientált programozás, ahol pl. egy *alakzat* osztálynak *téglalap*, *háromszög* vagy *kör* nevű leszármazottai lehetnek).
- Az SML-ben megkülönböztetett egyesítéssel tudunk létrehozni *különböző típusú elemekből* álló listát:

```
[In2 King, In1 "Skócia"] : ((string, person) disun) list;
[In1 "zsarnok", In2 1040] : ((string, int) disun) list
```

- A lehetséges eseteket most is *mintaillesztéssel* elemezhetjük, pl.

```
(* concat : (string, 'a) disun list -> string
   concat d = a d diszjunkt unió In1 címkejű
               elemeinek konkatenációja *)
fun concat [] = ""
  | concat (In1 s :: ls) = s ^ concat ls
  | concat (In2 _ :: ls) = concat ls
```

Megkülönböztetett egyesítés (folyt.)

- Egy példa concat alkalmazására:

```
- concat [In1 "Ó! ", In2 King, In1 "Skócia"];
> val it = "Ó! Skócia : string
```

- Az In1 konstruktorfüggvény típusa 'a -> ('a, 'b) disun, ezért a string típusú "Ó!" argumentumra alkalmazva (string, 'b) disun típusú érték az eredmény.
- Az In2 konstruktorfüggvény típusa 'b -> ('a, 'b) disun, ezért a person típusú King kifejezésre alkalmazva ('a, person) disun típusú érték az eredmény.
- Az [In1 "Ó!", In2 King, In1 "Skócia"] kifejezésben mindkét alaptípust lekötjük, ezért ennek a listának a típusa: ((string, person) disun) list.
- Az [In2 "Ó", In2 King, In1 "Skócia"] kifejezés kiértékelése hibajelzést eredményez, mert a 'b típusváltozót nem lehet ugyanabban a kifejezésben egyszer így, másszor úgy lekötöni.

Esetsztérválasztás (case)

```
case E of P1 => E1 | P2 => E2 | ... | Pn => En
```

Az SML-értelmező – balról jobbra és fölülről lefelé haladva – megpróbálja E-t P1-re illeszteni, ha nem sikerül, P2-re s.í.t. A case-kifejezés eredménye az E kifejezésre illeszkedő első Pi mintához tartozó Ei kifejezés lesz.

A case is csak szintaktikus édesítőszert, ui. helyettesíthető fn-jelöléssel:

```
(fn P1 => E1 | P2 => E2 | ... | Pn => En) E
```

Például a lady függvényt így is definiálhattuk volna:

```
datatype degree = Duke | Marquis | Earl | Viscount | Baron
(* lady : degree -> string
   lady p = p főnemes
           hitvesének rangja *)
fun lady p =
  case p of
    Duke    => "Duchess "
  | Marquis => "Marchioness"
  | Earl    => "Countess"
  | Viscount => "Viscountess"
  | Baron   => "Baroness"

(* lady : degree -> string
   lady p = p főnemes
           hitvesének rangja *)
fun lady p =
  (fn
    Duke    => "Duchess "
  | Marquis => "Marchioness"
  | Earl    => "Countess"
  | Viscount => "Viscountess"
  | Baron   => "Baroness"
  ) p
```

ESETSZÉTVÁLASZTÁS, OPCIONÁLIS ÉRTÉK

Opcionális érték kezelése ('a option)

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

Függvények az Option könyvtárból:

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

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

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

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

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

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

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

Példák opcionális értékek kezelésére

- Egészlista legnagyobb elemének kiválasztása

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

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

- Füzér elején álló karaktersorozat átalakítása egész számmá

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

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

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

LOKÁLIS ÉRVÉNYŰ DEKLARÁCIÓK

Deklaráció lokális érvényű deklarációval: local-deklaráció

- Ún. local-deklarációt használunk, ha egyes deklarációkat fel akarunk használni más deklarációkban, miközben *el akarjuk rejteni* őket a program többi része elől.

- Szintaxisa:


```
local d1      ahol d1 egy nemüres deklarációsorozat,
in d2        ahol d2 egy másik nemüres deklarációsorozat.
end
```

- Példa:

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

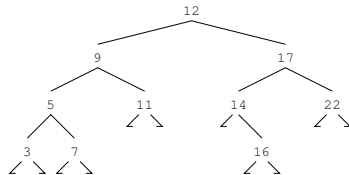
BINÁRIS FÁK

Bináris fák datatype deklarációval

- A listához hasonlóan rekurzív adatszerkezetet a *fa*.
- Először olyan bináris fát deklarálunk, amelynek a levelei üresek, a csomópontjaiban pedig előbb a bal részfat, majd az 'a' típusú értéket, és végül a jobb részfat adjuk meg:

```
datatype 'a tree = L | B of 'a tree * 'a * 'a tree
```

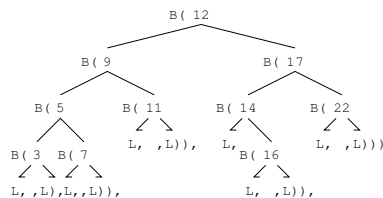
- Tekintsük például az alábbi fát:



- Az 'a tree' adattípus L és B adatkonstruktoraival ez a fa pl. a következő lapon látható módon írható le.

Bináris fák datatype deklarációval (folyt.)

- A fastruktúra szöveges leírása átláthatóbb, ha az egyes részfáknak nevet adunk, és a részfákból építjük fel a teljes fát:

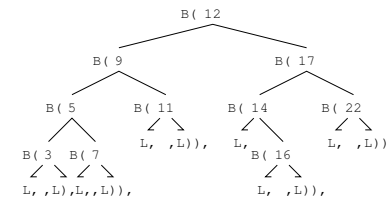


```
val tr3 = B(L,3,L);          val tr7 = B(L,7,L);
val tr5 = B(tr3,5,tr7);     val tr11 = B(L,11,L);
val tr9 = B(tr5,9,tr11);   val tr16 = B(L,16,L);
val tr14 = B(L,14,tr16);   val tr22 = B(L,22,L);
val tr17 = B(tr14,17,tr22); val tr12 = B(tr9,12,tr17)
```

Bináris fák datatype deklarációval (folyt.)

```
B(B(B(B(L,3,L),
      5,
      B(L,7,L)
    ),
    9,
    B(L,11,L)
  ),
  12,
  B(B(L,
    14,
    B(L,16,L)
  ),
  17,
  B(L,22,L)
)
)
```

A bal oldali kifejezést elég nehéz átlátni. A fastruktúra szöveges leírását megkönnyíti, ha az ábrába beírjuk a megfelelő adatkonstruktorkat.



Bináris fák datatype deklarációval (folyt.)

- Másféle fastruktúrákat is deklarálhatunk, pl.
 - kezdhetjük az 'a' típusú értékkel, majd folytathatjuk előbb a bal, azután a jobb részfa megadásával,
 - felhasználhatjuk a levelet is értékek tárolására,
 - az értéket nem tároló üres csomókat pedig E-vel jelölhetjük.
- A leírtak szerinti bináris fát hoz létre a következő deklaráció:

```
datatype 'a tree = E | L of 'a | B of 'a * 'a tree * 'a tree
```

- A rekurzív függvényekhez hasonlóan a rekurzív adattípusok deklarációjában is kell lennie nemrekurzív ágak (ún. triviális esetnek).
- A nemrekurzív ág hiánya miatt az alábbi, szintaktikailag helyes deklarációk használhatatlanok:

```
datatype 'a badtree = B of 'a badtree * 'a * 'a badtree
datatype 'a badtree = L of 'a badtree
                    | B of 'a badtree * 'a * 'a badtree
```

Egyszerű műveletek bináris fákon

- nodes egy fa csomópontjait számlálja meg. Legyen

```
datatype 'a tree = L | N of 'a * 'a tree * 'a tree

(* nodes : 'a tree -> int
   nodes f = az f fa csomópontjainak a száma *)
fun nodes (N(_, t1, t2)) = 1 + nodes t2 + nodes t1
  | nodes L = 0
```

- nodes akkumulátort használó változata (nodesa):

```
fun nodesa f =
  let (* nodes0(f, n) = n + a csomópontok száma f-ben
       nodes0 : 'a tree * int -> int *)
      fun nodes0 (N(_, t1, t2), n) =
          nodes0(t1, nodes0(t2, n+1))
        | nodes0 (L, n) = n
      in nodes0(f, 0)
  end
```

Egyszerű műveletek bináris fákon (folyt.)

- fulltree n mélységű teljes bináris fát épít, és a fa csomópontjait 1-től $2^n - 1$ -ig beszámozza. Egy teljes bináris fában minden csomópontból pontosan két él indul ki, és minden levelének ugyanaz a szintje.

```
(* fulltree : int -> 'a tree
   fulltree n = n mélységű teljes fa *)
fun fulltree n =
  let fun ftree (_, 0) = L
       | ftree (k, n) = N(k, ftree(2*k, n-1), ftree(2*k+1, n-1))
      in
        ftree(1, n)
      end
```

- reflect a fát a függőleges tengelye mentén tükrözi.

```
(* reflect : 'a tree -> 'a tree
   reflect t = a függőleges tengelye mentén tükrözött t fa *)
fun reflect L = L
  | reflect (N(v,t1,t2)) = N(v, reflect t2, reflect t1)
```

Egyszerű műveletek bináris fákon (folyt.)

- A fa gyökeréből a leveléhez vezető úton az élek számát (az út hosszát) az adott levél szintjének is nevezzük. A szintek közül a legnagyobbat a fa *mélységének* hívjuk.

- depth egy fa mélységét határozza meg.

```
(* depth : 'a tree -> int
   depth f = az f fa mélysége *)
fun depth (N(_, t1, t2)) = 1 + Int.max(depth t2, depth t1)
  | depth L = 0
```

- depth akkumulátort használó változata (deptha):

```
fun deptha f = let fun depth0 (N(_, t1, t2), d) =
                    Int.max(depth0(t1, d+1), depth0(t2, d+1))
                  | depth0 (L, d) = d
                  in
                    depth0(f, 0)
                  end
```

Lista előállítás bináris fa elemeiből

- Mindhárom függvény *bináris fából listát* állít elő. Abban különböznek egymástól, hogy a csomópontokban tárolt értékeket mikor veszik ki, és milyen sorrendben járják be a részfákat:

- preorder először az értéket veszi ki, majd bejárja a bal, és azután a jobb részfát;
- inorder először bejárja a bal részfát, majd kiveszi az értéket, végül bejárja a jobb részfát;
- postorder először bejárja a bal, majd a jobb részfát, és utoljára veszi ki az értéket.

- Az akkumulátort nem használó változatok egyszerűek, érthetőek, de nem elég hatékonyak a @ operátor használata miatt.

```
(* preorder : 'a tree -> 'a list
   preorder f = az f fa elemeinek preorder sorrendű listája *)
fun preorder L = []
  | preorder (N(v,t1,t2)) = v :: preorder t1 @ preorder t2
(* inorder : 'a tree -> 'a list
   inorder f = az f fa elemeinek inorder sorrendű listája *)
fun inorder L = []
  | inorder (N(v,t1,t2)) = inorder t1 @ (v :: inorder t2)
(* postorder : 'a tree -> 'a list
   postorder f = az f fa elemeinek postorder sorrendű listája *)
fun postorder L = []
  | postorder (N(v,t1,t2)) = postorder t1 @ (postorder t2 @ [v])
```

Lista előállítás bináris fa elemeiből (folyt.)

- Ha inorder előző változatában az `inorder t1 @ (v :: inorder t2)` kifejezésben a `v :: inorder t2` részkifejezést nem tesszük zárójelbe, a fordító hibát jelez, mivel `::` és `@` azonos precedenciájú, és ezért zárójelek nélkül a nyilvánvalóan hibás `inorder t1 @ v` részkifejezést akarná kiértékelni.
- `inorder` előző megvalósításával kb. egyenértékű a következő változata, amelyben a `v` elem helyett az egyelemű `[v]` listát fűzzük `inorder t2` elé:

```
fun inorder L = []
  | inorder (N(v,t1,t2)) = inorder t1 @ ([v] @ inorder t2)
```

Ez a változat azonban *roppant sérülékeny*, ugyanis a hatékonysága függ a zárójelek kirakásától.

Ha a `[v] @ inorder t2` részkifejezést nem tesszük zárójelbe, akkor a fordító először a `inorder t1 @ [v]` részkifejezést fogja kiértékelni, azaz egy egyelemű listához fűz egy (általában) jóval hosszabbat!

- Az elmondottakhoz hasonló okból `postorder` bemutatott változata is *rendkívül sérülékeny!* Ha ugyanis a `postorder t1 @ (postorder t2 @ [v])` kifejezésben az amúgyis rossz hatékonyságú `postorder t2 @ [v]` részkifejezést nem tesszük zárójelbe, akkor a fordító először a `postorder t1 @ postorder t2` részkifejezést értékeli ki, azaz a két, feltehetően hosszú listát fűzi egybe, majd a létrehozott eredménylistát fűzi az egyelemű listához!

Bináris fa előállítás lista elemeiből: `balPreorder`

- Listát *kiegyensúlyozott (balanced) bináris fává* alakítanak a következő függvények: `balPreorder`, `balInorder` és `balPostorder`; a különbség közöttük most is a bejárási sorrendben van.
- `(* balPreorder: 'a list -> 'a tree`
`balPreorder xs = az xs lista elemeiből álló, preorder`
`bejárású, kiegyensúlyozott fa`
`*)`

```
fun balPreorder [] = L
  | balPreorder (x::xs) =
    let val k = length xs div 2
    in
      N(x, balPreorder(List.take(xs, k)),
        balPreorder(List.drop(xs, k)))
    end
```
- A hatékonyságot kisebb mértékben rontja, hogy `List.take` és `List.drop` egymástól függetlenül *kétszer* mennek végig a lista első felén.

Lista előállítás bináris fa elemeiből (folyt.)

Az akkumulátort használó változatok nehezebben érthetőek meg, de *hatékonyabbak*, elsősorban a veremhasználat szempontjából.

```
(* preord : 'a tree * 'a list -> 'a list
   preord(f, vs) = az f fa elemeinek a vs lista elé fűzött,
                  preorder sorrendű listája *)
fun preord (L, vs) = vs
  | preord (N(v,t1,t2), vs) = v::preord(t1, preord(t2,vs))

(* inord : 'a tree * 'a list -> 'a list
   inord(f, vs) = az f fa elemeinek a vs lista elé fűzött,
                 inorder sorrendű listája *)
fun inord (N(v,t1,t2), vs) = inord(t1, v::inord(t2,vs))
  | inord (L, vs) = vs

(* postord : 'a tree * 'a list -> 'a list
   postord(f, vs) = az f fa elemeinek a vs lista elé fűzött,
                  postorder sorrendű listája *)
fun postord (N(v,t1,t2), vs) = postord(t1, postord(t2, v::vs))
  | postord (L, vs) = vs
```

`take` és `drop` egyetlen függvénnyel: `take'ndrop`

- Írjunk `take'ndrop` néven olyan függvényt, amelynek egy `xs` listából és egy `k` egészszől álló pár az argumentuma, és egy olyan pár az eredménye, amelynek első tagja a lista első `k` db eleme, második tagja pedig a lista többi eleme.
- `(* take'ndrop : 'a list * int -> 'a list * 'a list`
`take'ndrop(xs, k) = olyan pár, amelynek`
`első tagja xs első k db eleme,`
`második tagja pedig xs maradéka`
`*)`

```
fun take'ndrop (xs, k) =
  let fun td (xs, 0, ts) = (rev ts, xs)
      | td ([], _, ts) = (rev ts, [])
      | td (x::xs, k, ts) = td(xs, k-1, x::ts)
  in
    td(xs, k, [])
  end
```
- `take'ndrop` felhasználása, nevezetesen az eredményül átadott pár miatt módosítani kell `balpreorder` felépítésén.

Bináris fa előállítás lista elemeiből: balPreorder, újra

- Ez volt:

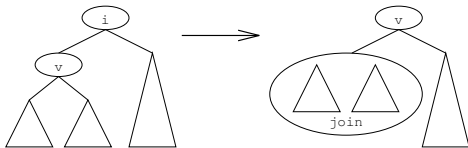
```
fun balPreorder [] = L
  | balPreorder (x::xs) =
    let val k = length xs div 2
    in N(x, balPreorder(List.take(xs, k)),
        balPreorder(List.drop(xs, k)))
    end
```

- Ez lett:

```
(* balPreorder: 'a list -> 'a tree
   balPreorder xs = az xs lista elemeiből álló, preorder ... *)
fun balPreorder [] = L
  | balPreorder (x::xs) =
    let val k = length xs div 2
        val (ts, ds) = take'ndrop(xs, k)
    in N(x, balPreorder ts, balPreorder ds)
    end
```

Elem törlése bináris fából

- Adott értékű *elemet* rekurzív módszerrel *megkeresni* egyszerű feladat.
- Új elemet beszúrni* sem nehéz: rekurzív módszerrel keresünk egy levelet, és ennek a helyére berakjuk az új értéket. Ha a fa rendezve van, ügyelnünk kell arra, hogy a rendezettség megmaradjon.
- Adott értékű *elemet* vagy *elemeket* rekurzív módszerrel *kitörölni* valamivel nehezebb: ha a törlendő érték az éppen vizsgált részfa gyökerében van, a két részre széteső fa részfáit *egyesíteni* kell, miután a törlést a két részfán már végrehajtottuk.



- Megtehetjük, hogy előbb egyesítjük a két részfát, majd az eredményül kapott fából töröljük az adott értékű elemet.

Bináris fa előállítás lista elemeiből

- (* balInorder: 'a list -> 'a tree

balInorder xs = az xs lista elemeiből álló, inorder bejárású, kiegyensúlyozott fa

*)

fun balInorder [] = L

 | balInorder (x::xs) =

 let val k = length xxs div 2

 val ys = List.drop(xxs, k)

 in

 N(hd ys, balInorder(List.take(xxs, k)),

 balInorder(tl ys))

 end
- (* balPostorder: 'a list -> 'a tree

balPostorder xs = az xs lista elemeiből álló, postorder bejárású, kiegyensúlyozott fa

*)

fun balPostorder xs = balPreorder(rev xs)
- balInorder take'ndrop-pal való definiálását meg hagyjuk gyakorló feladatnak.

Elem rekurzív törlése bináris fából (folyt.)

- A join-nal egyesítjük a törlés hatására létrejövő két részfát: a bal részfát lebontja, és közben az elemét egyesével berakja a jobb részfába.

```
(* join : 'a tree * 'a tree -> 'a tree
   join(b, j) = a b és a j fák egyesítésével létrehozott fa *)
fun join (L, tr) = tr
  | join (N(v, lt, rt), tr) = N(v, join(lt, rt), tr)
```

- A remove rendezetlen bináris fából törli az *i* értékű elem *összes* előfordulását.

```
(* remove : 'a * 'a tree -> 'a tree
   remove(i, f) = i összes előfordulását törli f-ből *)
fun remove (i, L) = L
  | remove (i, N(v, lt, rt)) =
    if i < v
    then N(v, remove(i, lt), remove(i, rt))
    else join(remove(i, lt), remove(i, rt))
```


Bináris keresőfák: blookup, binsert

- Rendszerint adott kulcsú elemet keresünk egy rendezett bináris fában, ehhez értékeket kell összehasonlítanunk egymással, ehhez a keresett kulcsnak *egyenlőségi típusúnak* kell lennie (a példában a string típust használjuk).
- A függvények *kivételt* jeleznek, ha a keresett kulcsú elem nincs a keresőfában: `exception Bsearch of string`.
- A `blookup` függvény adott kulcshoz tartozó értéket ad vissza:

```
(* blookup : (string * 'a) tree * string -> 'a
   blookup(f, b) = az f fában a b kulcshoz tartozó érték
*)
fun blookup (L, b) = raise Bsearch("LOOKUP: " ^ b)
| blookup (N((a,x), t1, t2), b) =
  if b < a      then blookup(t1,b)
  else if a < b then blookup(t2, b)
  else x
```

FUNKCIONÁLIS PROGRAMOZÁS, II. RÉSZ

Bináris keresőfák: bupdate

- A `binsert` függvény egy új kulcsú elemet rak be egy rendezett bináris fába, ha még nincs benne:

```
(* binsert : (string * 'a) tree * (string * 'a) -> (string * 'a) tree
   binsert(f, (b,y)) = az új (b,y) kulcs-érték párral bővített f fa *)
fun binsert (L, (b,y)) = N((b,y), L, L)
| binsert (N((a, x), t1, t2), (b,y)) =
  if b < a      then N((a, x), binsert(t1, (b,y)), t2)
  else if a < b then N((a, x), t1, binsert(t2, (b,y)))
  else (* a=b *) raise Bsearch("INSERT: " ^ b)
```

- A `bupdate` függvény meglévő kulcsú elembe új értéket ír be egy rendezett bináris fában:

```
(* bupdate : (string * 'a) tree * (string * 'a) -> (string * 'a) tree
   bupdate(f, (b,y)) = az f fa, a b kulcshoz tartozó érték helyén
                       az y értékkel *)
fun bupdate (L, (b,y)) = raise Bsearch("UPDATE: " ^ b)
| bupdate (N((a,x), t1, t2), (b,y)) =
  if b < a      then N((a,x), bupdate(t1, (b,y)), t2)
  else if a < b then N((a,x), t1, bupdate(t2, (b,y)))
  else (* a=b *) N((b,y), t1, t2)
```

- A függvények *generikussá* tételét meg hagyjuk gyakorló feladatnak.

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

- A funkcionális programozás alapjai, λ -kalkulus
- Típus, típusváltozó, egyenlőségi típus, típuskifejezés, egyszerű és összetett típus (az SML *erősen típusos* nyelv)
- Név, érték, változó (nem frissíthető!), kifejezés (az SML-ben minden érték *teljes jogú érték*)
- Függvény (egyetlen paraméter!), részlegesen alkalmazható és magasabbrendű függvény; névtelen függvény (fn-jelölés vagy λ -kifejezés), mintaillesztés
- Egyszerű típus (int, real, char, bool, word, string, unit)
- Összetett típus: lista, rekord, ennes, nullas, datatype-deklaráció
- Listakezelő függvények (hd, length, map, filter, max1, @, rev, revAppend, take, drop, foldr, foldl stb.)
- Lokális érvényű deklaráció: let-kifejezés, local-deklaráció
- Könyvtárak (Char, Int, Real, String, List, Bool, Option, Time, Timer stb.)
- Logikai műveletek (andalso, orelse, not), if-then-else kifejezés); lusta kiértékelés
- Polimorfizmus: paraméteres és többszörös terheléses
- Listák használata, futási idő mérése

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

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

Kiírás

- `{TextIO.}print : string -> unit`
print s = kiírja az s értékét a standard kimenetre, és azonnal kiüríti a puffert.
- `{General.}makestring : numtxt-> string`
makestring v = eredménye a v érték ábrázolása.
- `{Meta.}printVal : 'a -> 'a`
printVal e = kiírja az e kifejezés értékét a standard kimenetre pontosan úgy, ahogyan az SML értelmező írja ki a „legfelső szinten”, és azonnal kiüríti a puffert. Eredményül visszaadja az e kifejezés értékét. *Csak interaktív módban használható.*

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

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

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

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

KIÍRÁS

Kiírás (folyt.)

- Példák:

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

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

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

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

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

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

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

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

Kiírás (folyt.)

- `printVal`-al `datatype`-deklarációval létrehozott típusú érték is kiírható ki, pl.


```
- datatype t = L | B of t * t;
> New type names: =t
  datatype t = (t, con B : t * t -> t, con L : t)
  con B = fn : t * t -> t
  con L = L : t
- val fa = B(B(B(L, B(L, B(L, B(B(L, L), L))), L), B(L, L)));
> val fa = B(B(B(L, B(L, B(L, B(B(L, L), L))), L), B(L, L)) : t
- printVal fa;
B(B(B(L, B(L, B(L, B(B(L, L), L))), L), B(L, L))> val it = B(...
```
- A kiírt sor túl hosszú, a folytatását ...-tal helyettesítettük.
- Törjük el a sort a `> válaszcím` előtt *szekvenciális* kifejezés alkalmazásával:


```
- (printVal fa; print "\n");
B(B(B(L, B(L, B(L, B(B(L, L), L))), L), B(L, L))
> val it = () : unit
```
- Sajnos, az eredményül kapott érték nem `fa` értéke!

Kiírás (folyt.)

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


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


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


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

Kiírás (folyt.)

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


```
- let val res = printVal fa; val _ = print "\n"
  in
    res
  end;
B(B(B(L, B(L, B(L, B(B(L, L), L))), L), B(L, L))
> val it = B(B(B(L, B(L, B(L, B(B(L, L), L))), L), B(L, L)) : t
```
- Ez elég körülményes. A `before` operátort az ilyen esetek kezelésére vezették be:


```
- printVal fa before print "\n";
B(B(B(L, B(L, B(L, B(B(L, L), L))), L), B(L, L))
> val it = B(B(B(L, B(L, B(L, B(B(L, L), L))), L), B(L, L)) : t
```
- Szekvenciális kifejezés alkalmazásával további magyarázó szöveget írhatunk ki:


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

NYOMKÖVETÉS KIÍRÁSSAL: LISTÁK

Nyomkövetés kiírással: length (nem iteratív)

- Az MOSML-ben nyomkövetés csak a program szövegébe beírt kiíró függvényekkel lehetséges.

- Példa: a length függvény két változatának kiértékelése

- A length „naív” változata

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

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

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

Nyomkövetés kiírással: length egy alkalmazása

- length egy alkalmazása

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

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

Nyomkövetés kiírással: lengthi (iteratív)

- A length iteratív változata

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

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

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

Nyomkövetés kiírással: lengthi egy alkalmazása

- lengthi egy alkalmazása

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

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

Nyomkövetés kiírással: length és lengthi összehasonlítása

- length és lengthi kiértékelésének összehasonlítása

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

Deklaratív programozás. BME VIK, 2003. tavaszi félév

(Funkcionális programozás)

Nyomkövetés kiírással FP-147

Hibakeresés Poly/ML-ben (folyt.)

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

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

Deklaratív programozás. BME VIK, 2003. tavaszi félév

(Funkcionális programozás)

Hibakeresés Poly/ML-ben

- Példafüggvény a Poly/ML debugger használatához

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

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

Deklaratív programozás. BME VIK, 2003. tavaszi félév

(Funkcionális programozás)

Nyomkövetés kiírással FP-148

Hibakeresés Poly/ML-ben (folyt.)

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

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

- length egy hibás változata a hibakeresés kipróbálásához

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

Deklaratív programozás. BME VIK, 2003. tavaszi félév

(Funkcionális programozás)

Hibakeresés Poly/ML-ben (folyt.)

- length egy másik hibás változata a hibakeresés kipróbálásához

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

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

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

Nyomkövetés kiírással: bináris fák

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

NYOMKÖVETÉS KIÍRÁSSAL: BINÁRIS FÁK

Nyomkövetés kiírással: nodes (akkumulátort nem használ)

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

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

Nyomkövetés kiírással: nodesa (akkumulátort használ)

```

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

```

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

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

```

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

(nodes f7) | (nodesa f7)
|
|
<2> N(5, L, L) * | <2> N(4, L, L) %
<5> L * | N(5, L, L) *
L % | 5 $
$ 1 # | <5> L %
N(4, L, L) % | L *
<4> L * | 6 $ #
L % | <4> L %
$ 1 # | L *
$ 3 # | 7 $ # # #
$ 7 # |
|
> val it = 7 : int | > val it = 7 : int

```

Nyomkövetés kiírással: nodes és nodesa egy alkalmazása

```

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

- nodes f7; | - nodesa f7;
|
|
<1> N(3, N(6, L, L), N(7, L, L)) * | <1> N(2, N(4, L, L), N(5, L, L)) %
<3> N(7, L, L) * | N(3, N(6, L, L), N(7, L, L)) *
<7> L * | 1 $
L % | <3> N(6, L, L) %
$ 1 # | N(7, L, L) *
N(6, L, L) % | 2 $
<6> L * | <7> L %
L % | L *
$ 1 # | 3 $ #
$ 3 # | <6> L %
N(2, N(4, L, L), N(5, L, L)) % | L *
| 4 $ # #

```

Folytatása a következő lapon.

Nyomkövetés kiírással: depth (akkumulátort nem használ)

```

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

```

Nyomkövetés kiírással: deptha (akkumulátort használ)

```

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

```

- A következő föliákon depth-t és deptha-t hét csomópontból álló teljes fára alkalmazzuk.

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

Folytatás az előző lapról.

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

```

(depth f7) | (deptha f7)
|
|
| <2> N(5, L, L) * | <2> N(5, L, L) *
| <5> L * | 2 $
| L % | <5> L *
| 1 # | 3 $
| N(4, L, L) % | L %
| <4> L * | 3 &
| L % | 3 #
| 1 # | N(4, L, L) %
| 2 # | 2 &
| 3 # | <4> L *
| | 3 $
| | L %
| | 3 &
| | 3 #
| | 3 #
| | 3 #
| > val it = 3 : int | > val it = 3 : int

```

Nyomkövetés kiírással: depth és deptha egy alkalmazása

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

```

- depth f7; | - deptha f7;
|
|
| <1> N(3, N(6, L, L), N(7, L, L)) * | <1> N(3, N(6, L, L), N(7, L, L)) *
| <3> N(7, L, L) * | 1 $
| <7> L * | <3> N(7, L, L) *
| L % | 2 $
| 1 # | <7> L *
| N(6, L, L) % | 3 $
| <6> L * | L %
| L % | 3 &
| 1 # | 3 #
| 2 # | N(6, L, L) %
| N(2, N(4, L, L), N(5, L, L)) % | 2 &
| <6> L * | 3 $
| 3 $ | L %
| 3 & | 3 &
| 3 # | 3 #
| 3 # |
| N(2, N(4, L, L), N(5, L, L)) % |
| 1 & |

```

KIVÉTELKEZELÉS

Kivételkezelés

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

Kivételkezelés (folyt.)

- A következő programrészlet példa kivétel deklarálására, jelzésére és kezelésére

```
exception Hiba of char * int;

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

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

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

Kivételkezelés (folyt.)

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

Kivételkezelés (folyt.)

- Példa visszalépés programozására kivételkezeléssel

```
exception Valt;

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

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

Kivételkezelés (folyt.)

- A leggyakoribb belső kivételek

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

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

HALMAZMŰVELETEK

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

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

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

infix isMem

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

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

newMem, ha a sorrendtől eltekintünk, halmazt hoz létre.

Halmazműveletek: „listából halmaz” (setof)

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

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

- Öt halmazműveletet definiálunk:

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

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

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

- Két halmaz uniója

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

- Két halmaz metszete

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

Halmazműveletek: „halmaz hatványhalmaza” (powerSet)

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

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

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

- Részalmaz-e egy halmaz egy másiknak?

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

infix isSubset
```

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

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

Halmazműveletek: „halmaz hatványhalmaza” (folyt.)

- Ezzel a `pws` függvény:

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

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

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

Halmazműveletek: „halmaz hatványhalmaza”, hatékonyabban

- pws rossz hatékonyságú, mert kétfelé ágazó rekurziót használ. Pl. egy 19 egész számból álló lista hatványhalmazának előállítását nem lehet kivárni. Írjunk hatékonyabb változatot.

- Az insAll segédfüggvény egy elemet szűr be egy listából álló lista minden eleme elé.

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

- powerSet insAll-t használó rekurzív változata

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

- powerSet insAll-t használó iteratív változata

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

Egyidejű deklaráció

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

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

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

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

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

EGYIDEJŰ DEKLARÁCIÓ

Egyidejű deklaráció (folyt.)

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

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

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

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

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

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

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

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

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

AZ ORDER TÍPUS

Az order típus

Az order típus definíciója (ld. General.sig)

```
datatype order = LESS | EQUAL | GREATER
```

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

Példák az SML-alapkönyvtárból (SML Basis Library)

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

Listák rendezése

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

LISTÁK RENDEZÉSE

Beszúró rendezés

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

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

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

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

- Példa inssort alkalmazására:

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

Beszúró rendezés, generikus változat (folyt.)

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

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

Beszúró rendezés, generikus változat

- Az ins függvényt generikussá tesszük:

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

- Ezzel inssort egy újabb változata:

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

Beszúró rendezés foldr-rel és foldl-lel

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

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

- Példák inssort-tal és inssort2-vel:

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

- Példák foldr és foldl felhasználásával:

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

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

A futási idők mérése, összehasonlítása

- 2000 elemet tartalmazó, véletlenszerűen előállított, illetve eredetileg éppen fordított sorrendű listák rendezéséhez szükséges futási időt mérünk.

- Véletlen eloszlású egészlistát állít elő a Random könyvtárbeli rangelist függvény:


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

- Növekvő sorrendű egészlistát állít elő a -- operátor:

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

val xs2000N = 1 -- 2000;
```

Deklaratív programozás. BME VIK, 2003. tavaszi félév

(Funkcionális programozás)

Listák rendezése FP-187

A futási idők mérése, összehasonlítása (folyt.)

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

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

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

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

Deklaratív programozás. BME VIK, 2003. tavaszi félév

(Funkcionális programozás)

A futási idők mérése, összehasonlítása (folyt.)

- A futási időt az alábbi függvénnyel mérhetjük:

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

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

Deklaratív programozás. BME VIK, 2003. tavaszi félév

(Funkcionális programozás)

Listák rendezése FP-188

Kiválasztó rendezés

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

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

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

Deklaratív programozás. BME VIK, 2003. tavaszi félév

(Funkcionális programozás)

Kiválasztó rendezés (folyt.)

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

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

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

Deklaratív programozás. BME VIK, 2003. tavaszi félév

(Funkcionális programozás)

Listák rendezése FP-191

Gyorsrendezés akkumulátorral

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

Deklaratív programozás. BME VIK, 2003. tavaszi félév

(Funkcionális programozás)

Gyorsrendezés akkumulátor nélkül

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

Deklaratív programozás. BME VIK, 2003. tavaszi félév

(Funkcionális programozás)

Listák rendezése FP-192

A futási idők mérése, összehasonlítása

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

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

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

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

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

Deklaratív programozás. BME VIK, 2003. tavaszi félév

(Funkcionális programozás)

Összefésülő rendezések

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

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

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

TÖBB MEGOLDÁS ELŐÁLLÍTÁSA VISSZALÉPÉSSEL

Fölről lefelé haladó összefésülő rendezés

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

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

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

n vezér a sakktáblán

- Hányféleképpen rakható n vezér a sakktáblára úgy, hogy ne üssék egymást?
- A vezéreket tartalmazó mezők sorának j sorszámát az egyes oszlopokon belül egy n hosszú sorvektor adott oszlophoz rendelt mezőjébe írt szám adja meg, ahol $j \leq s < n$. Példa $n=4$ esetén:
- A sorvektort (egy egyre bővülő) listával valósítjuk meg. Egy listához balról könnyű új elemeket fűzni, ezért a táblát és a vezérek helyzetét leíró listát hossz tengelye mentén tükrözzük.

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

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

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

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

n vezér a sakktáblán (folyt.)

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

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

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

```

...+-----+
  s |   |   |
...+-----+

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

```

n vezér a sakktáblán: egy megoldás előállítása

```

exception Zsakutca

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

```

n vezér a sakktáblán: „ütésben van”-vizsgálat

```

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

```

n vezér a sakktáblán: több megoldás előállítása visszalépéssel

```

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

```

n vezér a sakktáblán: több megoldás előállítása listák listájával

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

FÜGGVÉNYEK KOMPOZÍCIÓJA

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

Akkumulátor alkalmazásával:

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

Függvények kompozíciója

- Az $f \circ g$ függvénykompozíció az SML-ben

```
(* f o g = az f és g függvények kompozíciója
*)
```

```
infix 2 o;
fun (f o g) = fn x => f(g x); vagy
fun (f o g) x = f(g x);
```

- Az o típusa $? \rightarrow ?$ szerkezetű. Mít írjunk a $?$ -ek helyébe? Vezessük le!

- A függvénydefiníció jobb oldalán álló kifejezés elemzésével kezdjük.

```
x : 'a      g : 'a -> 'b      (g x) : 'b      f : 'b -> 'c
```

- A $\text{fun } (f \circ g) \ x = f(g \ x)$ függvénydefinícióban az egyenlőségjel (=) bal és jobb oldalán álló kifejezéseknek azonos értéket kell eredményül adniuk, ezért $f \circ g$ és f eredményének azonos a típusa (azaz 'c).

```
(f o g) : 'a -> 'c      o : ('b -> 'c) * ('a -> 'b) -> ('a -> 'c)
```

- Példa: $\text{round} : \text{real} \rightarrow \text{int}$, $\text{chr} : \text{int} \rightarrow \text{char}$
 $\text{chr} \circ \text{round} : \text{real} \rightarrow \text{char}$

ÖSSZEFOGLALÓ: TÍPUSOK, ÁLLANDÓK, NEVEK, KÖTÉSEK

Különleges állandók az SML-ben (összefoglaló)

- Előjeles egész állandó

Példák: 0 ~0 4 ~04 999999 0xFFFF ~0x1ff
Ellenpéldák: 0.0 ~0.0 4.0 1E0 -317 0xFFFF -0x1ff

- Valós állandó

Példák: 0.7 ~0.7 3.32E5 3E~7 ~3E~7 3e~7 ~3e~7
Ellenpéldák: 23 .3 4.E5 1E2.0 1E+7 1E-7

- Előjel nélküli egész állandó

Példák: 0w0 0w4 0w999999 0wxFFFF 0wx1ff
Ellenpéldák: 0w0.0 ~0w4 -0w4 0w1E0 0wXXXX 0wXXXX

- Füzérállandó: "-ek között álló nulla vagy több nyomtatható karakter, szóköz vagy \ jellel kezdődő *escape-szekvencia* (l. a táblázatot a következő lapon).

- Karakterállandó: # jelet közvetlenül követő, egykarakteres füzérállandó.

Példák: # "a" # "\n" # "\^Z" # "\255" # "\""
Ellenpéldák: # "a" #c # ""

Típusok az SML-ben (összefoglaló)

- Típusok és programozási nyelvek

- Típus nélküli nyelvek, pl. assembly, LISP, Prolog
- Gyengén típusos nyelvek, pl. Fortran, Algol, BASIC, C, C++, Pascal
- Erősen típusos nyelvek, pl. Ada, **SML**, clean, haskell
- Erős típus: a típusok (~ halmazok) diszjunktak, azaz nincs közös elemük

- Egyszerű SML-típusok

- int – előjeles egész szám, a Z egy részhalmaza
- word, word8 – előjel nélküli nemnegatív egész szám, az N_0 egy részhalmaza
- real – előjeles racionális (valós?!) szám, a Q egy részhalmaza
- bool, char, order, unit, string, substring, Time.time, Date.date, ...
- felsorolásos típus, pl. datatype color = Red | White | Green

- Polimorf összetett SML-típusok

- rekord, ennes, pl. type ('a, 'b) par = 'a * 'b
- lista, vektor (nem frissíthető), tömb (frissíthető), pl. 'a list, 'b vector, 'c array
- fák és más rekurzív típusok, pl. datatype 'a t = L | N of 'a t * 'a t * 'a

Escape-szekvenciák az SML-ben (összefoglaló)

- Escape-szekvenciák

\a Csengőjel (BEL, ASCII 7).
 \b Visszalépés (BS, ASCII 8).
 \t Vízszintes tabulátor (HT, ASCII 9).
 \n Újsor, soremelés (LF, ASCII 10).
 \v Függőleges tabulátor (VT, ASCII 11).
 \f Lapdobás (FF, ASCII 12).
 \r Kocsi-vissza (CR, ASCII 13).
 \^c Vezérlő karakter, ahol $64 \leq c \leq 95$ (@ ... _), és ^c ASCII-kódja 64-gyel kevesebb c ASCII-kódjánál.
 \ddd A ddd kódú karakter (d decimális számjegy).
 \uxxxx A xxxx kódú karakter (x hexadecimális számjegy).
 \" Idézőjel (").
 \\ Hátrátört-vonal (\).
 \f...f\ Figyelmen kívül hagyott sorozat. f...f nulla vagy több formázókaraktert (szóköz, HT, LF, VT, FF, CR) jelent.

Név képzése az SML-ben (összefoglaló)

- Alfanumerikus (*alphanumeric*) név: kis- és nagybetűk, számjegyek, perccjelek (') és aláhúzás-jelek (_) olyan sorozata, amely betűvel vagy perccjellel kezdődik

- Példák: `tothGyorgy Toth_3_Gyorgy toth'gyorgy`

- Írásjelekből álló (*symbolic*) név: az alábbi írásjelek tetszőleges, nem üres sorozata

! % & \$ # + - / : < = > ? @ \ ~ ' ^ | *

- Példák: `++ <-> ||| ## |=|`

- Speciális a szerepe az alábbi *fenntartott írásjeleknek*

() [] { } , ;

- Más jelentés nem rendelhető az alábbi *fenntartott nevekhez*

```
abstype and andalso as case do datatype else end eqtype exception
fn fun functor handle if in include infix infixr let local nonfix
of op open orelse raise rec sharing sig signature struct structure
then type val where with withtype while : :: :> _ | = => -> #
```

Függvénynév helyzete és kötése

- Függvénynév (másnéven függvényjel) helyzete és kötése általában
 - Egy függvénynév *prefix*, *infix* vagy *postfix* helyzetű lehet.
 - Az infix helyzetű függvénynevet gyakran *operátornak* nevezik.
 - Egy (infix helyzetű) operátor lehet *asszociatív* vagy *nem-asszociatív*; köthet balra vagy jobbra, vagy semerre. Asszociatív operátor esetén a kötési iránynak nincs jelentősége.
- Infix Prolog-operátor kötése
 - $xfx = f$ mindkét oldalán f csak zárójelben ismétlődhet (f „nem köt”),
 - $yfx = f$ bal oldalán f zárójelezés nélkül ismétlődhet (f „balra köt”),
 - $xfy = f$ jobb oldalán f zárójelezés nélkül ismétlődhet (f „jobbra köt”).

Értékdeklaráció – név kötése értékhez – az SML-ben (összefoglaló)

- Tetszőleges típusú érték köthető tetszőleges névhez:

```
val harom = 2 + 1           : int
val MHz = 94.5             : real
val veege = true           : bool
val kisA = #"a"            : char
val palindrom = "ABBA"     : string
val kisebb = LESS          : order
val ezNemSemmi = ()        : unit
val rat = {num = 3, den = 4} : {den : int, num : int}
val bLista = [2,3,4] @ [3,2] : int list
val telenek = [0w123, 0wxcd] : word list
```

true, false
LESS, EQUAL, GREATER
Egyetlen értéke a (), azaz nulla
Mezőnevek ábécéjében.

- Függvényérték is köthető tetszőleges névhez:

```
val incMod = fn i => fn n => (i + 1) mod n : int -> int -> int
fun incMod i n = (i + 1) mod n           : int -> int -> int
```

λ -jelöléssel
A szokásos alakba

- Típusmegkötés:

```
val id = fn (n : int) => n           : int -> int
val telenek = [0w65, 0wx41 : word8] : word8 list
```

Pl. `id 3`, `de id 4.5` nem!

Függvénynév helyzete és kötése az SML-ben

- Kifejezések és típuskifejezések az SML-ben
 - Az SML-ben a szokásos kifejezések mellett vannak *típuskifejezések* is.
 - A függvények *értékekre*, a típusfüggvények *típusokra* alkalmazhatók.
- Függvénynév és típusfüggvénynév helyzete és kötése az SML-ben
 - Függvénynév: *prefix* vagy *infix*.
 - Típusfüggvénynév: *infix* vagy *postfix*.
 - Az *infix* helyzetű függvénynév és típusfüggvénynév (szokásos nevén operátor, ill. típusoperátor) balra vagy jobbra köt, vagy semerre nem köt.
- Típusoperátorok
 - A két infix helyzetű beépített típusoperátor közül a `->` jobbra, a `*` semerre nem köt.
 - A `*` operátornak magasabb a precedenciája, mint a `->` operátornak.
 - A típusoperátoroknak magasabb a precedenciájuk a többi operátorénál.
 - Példák:

<code>int * real * string</code>	Egy hármas
<code>int * (real * string)</code>	Egy pár, amelynek a második tagja is egy pár
<code>(int * real) * string</code>	Egy pár, amelynek az első tagja is egy pár
<code>int * real -> string</code>	Függvény, amelynek egy pár az argumentuma
<code>int * (real -> string)</code>	Egy pár, amelynek a második tagja egy függvény

Függvénynév helyzete és kötése az SML-ben

- Tetszőleges kétargumentumú függvénynévet lehet meghatározott preferenciájú (infix helyzetű) operátorként deklarálni az `infix` vagy az `infixr` deklaratívával.
- Az `infix` balra, az `infixr` jobbra kötő operátort deklarál.
- Az `op` deklarátiva egy (esetleg `infix` helyzetű) operátort átmenetileg `prefix` helyzetűvé alakít.
- A `nonfix` deklarátiva egy (esetleg `infix` helyzetű) operátort tartósan `prefix` helyzetűvé alakítja.
- Egy minősített nevet, vagy egy olyan nevet, amelyet az `op` deklarátiva előz meg, csak `prefix` helyzetben lehet alkalmazni.
- Táblázatban összefoglalva:

<code>infix <d> id₁...id_n</code>	balra köt	binds to the left
<code>infixr <d> id₁...id_n</code>	jobbra köt	binds to the right
<code>nonfix id₁...id_n</code>	prefix	prefix
- A táblázatban `idi` tetszőleges nevet jelöl ($n \geq 1$). A `d` 0 és 9 közötti számjegy, az operátor precedenciája (opcionális, alapértelmezés szerinti értéke 0). Nagyobb szám nagyobb precedenciát jelent (éppen fordítva, mint a Prologban!).

EGYSZERŰSÍTETT SML-SZINTAXIS

A beépített operátorok és precedenciájuk az SML-ben

Az alábbi táblázatban `wordint`, `num` és `numtxt` az alábbi típusnevek helyett állnak.

`wordint` = `int`, `word`, `word8`. `num` = `int`, `real`, `word`, `word8`.
`numtxt` = `int`, `real`, `word`, `word8`, `char`, `string`.

Prec.	Operátor	Típus	Eredmény	Kivétel
7	*	<code>num * num -> num</code>	szorzat	Overflow
	/	<code>real * real -> real</code>	hányados	Div, Overflow
	<code>div, mod</code>	<code>wordint * wordint -> wordint</code>	hányados, maradék	Div, Overflow
	<code>quot, rem</code>	<code>int * int -> int</code>	hányados, maradék	Div, Overflow
6	<code>+, -</code>	<code>num * num -> num</code>	összeg, különbség	Overflow
	<code>^</code>	<code>string * string -> string</code>	egybeírt szöveg	Size
5	<code>::</code>	<code>'a * 'a list -> 'a list</code>	elemmel bővített lista (jobbra köt)	
	<code>@</code>	<code>'a list * 'a list -> 'a list</code>	összefűzött lista (jobbra köt)	
4	<code>=, <></code>	<code>'a * 'a -> bool</code>	egyenlő, nem egyenlő	
	<code><, <=</code>	<code>numtxt * numtxt -> bool</code>	kisebb, kisebb-egyenlő	
	<code>>, >=</code>	<code>numtxt * numtxt -> bool</code>	nagyobb, nagyobb-egyenlő	
3	<code>:=</code>	<code>'a ref * 'a -> unit</code>	értékkadás	
	<code>o</code>	<code>('b -> 'c) * ('a -> 'b) > ('a -> 'c)</code>	a két függvény kompozíciója	
0	<code>before</code>	<code>'a * 'b -> 'a</code>	a bal oldali argumentum	

SML-szintaxis: szintaktikai kategóriák (egyszerűsítve)

- A nevek *szintaktikai kategóriákba* sorolhatók

val	értéknév	value identifier	long
tyvar	típusváltozó	type variable	
tycon	típuskonstruktor	type constructor	long
lab	mezőnév	record label	
strid	struktúranév	structure identifier	long
sigid	szignatúranév	signature identifier	
unitid	állománynév	unit identifier	
- Az *értéknév* tetszőleges név; jelölhet állandó értéket, függvényértéket, adatkonstruktor, kivételkonstruktor. Példák: `pi`, `+`, `sin`, `nil`, `true`, `Match`
- A *típusváltozó* perccel kezdődő alfanumerikus név. Példa: `'a`.
- A *típuskonstruktor* tetszőleges név; jelölhet típusállandót vagy típusfüggvényt. Példák: `int`, `order`, `*`, `->`, `list`
- A *mezőnév* tetszőleges név vagy (nem 0-val kezdődő) pozitív egész szám. Példák: `num`, `2`
- A *strid* tetszőleges név: SML-kódot tartalmazó „implementációs” modult (ún. SML-struktúrát) azonosít. A fájlnev-kiterjesztés célszerűen `.sml`, de ez nem kötelező. Példák: `Int.sml`, `Rat.ml`

SML-szintaxis: szintaktikai kategóriák (folyt.)

- A *sigid* tetszőleges név: SML-kódot tartalmazó „interfész”- modult (ún. SML-szignatúrát) azonosít. A fájlnev-kiterjesztés kötelezően *.sig*. Példák: `Int.sig`, `Rat.sig`.
- A *unitid* egy struktúra, ill. szignatúra lefordításával létrejövő, tárgykódot tartalmazó fájl neve. A fájlnev-kiterjesztés szignatúra esetén *.ui*, struktúra esetén *.uo*. Példák: `Int.ui`, `Rat.uo`.
- Minden, az előző felsorolásban „long”-gal megjelölt *X* szintaktikai kategóriának van egy *longX* párja. A *longX* szintaktikai kategóriába tartozó nevek rövid és hosszú (ún. minősített) alakban is felírhatók. A *rövid alak* csak egy névből, a *hosszú alak* egy hosszú struktúranévből, egy pontból és egy névből áll:

$$\text{long}x ::= x \quad \left| \begin{array}{l} \text{név} \\ \text{longstrid}.x \end{array} \right| \begin{array}{l} \text{identifier} \\ \text{teljes név} \\ \text{qualified identifier} \end{array}$$

Példák:

- `explode, Misc.explode`
- `Real.toString`
- `Int.+`
- `List.filter`

SML-szintaxis: kifejezések és klózsorozatok (egyszerűsítve)

- Kifejezés (*exp*: expression)

(1) <code>exp ::= infexp</code>		
(2) <code>exp : ty</code>	típusmegkötés	type constraint
(3) <code>raise exp</code>	kivételjelzés	raise exception
(4) <code>case exp of match</code>	esetszétválasztás	case analysis
(5) <code>fn match</code>	függvénykifejezés	function expression

- Példák:

```
fn (n : int) => n;                vö. (2), (5)
case c of 00 => 01 | 01 => 11 | 11 => 10 | 10 => 00;  vö. (4), (19)
fn 00 => 01 | 01 => 11 | 11 => 10 | 10 => 00;      vö. (5), (19)
fn 00 => 01 | 01 => 11 | 11 => 10 | 10 => 00
  | _ => raise Domain; vö. (3), (5), (19)
```

SML-szintaxis: nemterminális szimbólumok, nyelvtani jelölések

- Minden nemterminális szimbólumot *változatok* sorozataként definiálunk, soronként egy változattal. Üres sor üres változatot jelent.
- A `<` és a `>` csúcsos zárójelpárok opcionális kifejezést fognak közre.
- Bármely *X* nemterminális szimbólumra az alábbiak szerint definiáljuk az *Xseq* nemterminális szimbólumot:

$Xseq ::= X$	egyelemű sorozat	singleton sequence
	üres sorozat	empty sequence
X_1, \dots, X_n	sorozat, $n \geq 1$	sequence, $n \geq 1$
- A változatokat csökkenő prioritási sorrendben soroljuk föl.
- A változatokat számozzuk, a példákban utalunk az alkalmazott változatra.
- A függvénynevek és operátorok általában balra kötnek, az eltérést jelezzük.
- Minden ismétlődő konstrukció (pl. a *klózsorozat*) a lehető legmesszebb terjeszkedik jobbra. Ezért pl. egy *case*-kifejezést egy másik *case*- vagy *fn*-kifejezésen, valamint egy *fun*-definíción belül zárójelbe kell tenni.

SML-szintaxis: kifejezések és klózsorozatok (folyt.)

- Infix kifejezés (*infexp*: infix expression)

(6) <code>infexp ::= appexp</code>		
(7) <code>infexp₁ id infexp₂</code>	infix alkalmazás	infix application

- Applikatív kifejezés (*appexp*: applicative expression)

(8) <code>appexp ::= atexp</code>		
(9) <code>appexp atexp</code>	(prefix) alkalmazás	(prefixed) application

- Példák:

```
3 + 4;                vö. (7)
Real.toString 3.56;   vö. (9)
Int.toString(round 3.56); vö. (9), (17)
```


LISTÁK RENDEZÉSE

Listák rendezése FP-227

Összefésülő rendezések (folyt.)

- Az összefésülő rendezéshez kell egy olyan függvény, amely két listát növekvő sorrendben egyesít. Korábban már definiáltuk a `merge` függvényt.

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

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

Listák rendezése (folyt.)

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

Deklaratív programozás. BME VIK, 2003. tavaszi félév

(Funkcionális programozás)

Listák rendezése FP-228

Alulról felfelé haladó összefésülő rendezés

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

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

```

Alulról fölfelé haladó összefésülő rendezés (folyt.)

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

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

Alulról fölfelé haladó összefésülő rendezés (folyt.)

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

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

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

Alulról fölfelé haladó összefésülő rendezés (folyt.)

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

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

Alulról fölfelé haladó összefésülő rendezés (folyt.)

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


```
bmsort [1,2,3,4,5,6,7,8,9]
----> sorting ([1,2,3,4,5,6,7,8,9], [], 0)
```
- Amíg `sorting` első argumentuma a nem üres (`x::xs`) lista, `sorting` saját magát hívja meg. A rekurzív hívás
 - első argumentuma a lépésenként egyre rövidülő `xs` lista,
 - második argumentuma a `mergepairs([x]:lss, k+1)` függvényalkalmazás eredménye, ahol kezdetben `lss = []`,
 - harmadik argumentuma $(k+1)$ a már feldolgozott listaelemek száma.

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

Alulról fölfelé haladó összefésülő rendezés (folyt.)

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

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

Simarendezés

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


```
(* nextrun (run, xs) = olyan pár, amelynek első tagja xs egy
    növekvő sorrendű futama, második tagja
    pedig xs maradéka
    nextrun : int list * int list -> int list * int list
*)
fun nextrun (run, x::xs) =
    if x < hd run
    then (rev run, x::xs)
    else nextrun(x::run, xs)
  | nextrun (run, []) = (rev run, [])
```
- nextrun eredménye egy pár, amelynek első tagja a futam (egy növekvő számsorozat), a második tagja pedig a rendezendő lista maradéka.
- A futam csökkenő sorrendben bővül, kilépéskor a futam meg kell fordítani.

Alulról fölfelé haladó összefésülő rendezés (folyt.)

lss	n	j	k		fun sorting (x::xs, lss, k) =
[[1]]	1	1	0	m1	sorting(
[[2], [1]]	2	2	1	m2	xs,
[[1, 2]]	1			m3	mergepairs([x]::lss, k+1),
[[3], [1, 2]]	3	3	10	m3	k+1
[[4], [3], [1, 2]]	4	4	11	m2)
[[3, 4], [1, 2]]	2			m2	sorting ([], lss, k) =
[[1, 2, 3, 4]]	1			m3	hd(mergepairs(lss, 0))
[[5], [1, 2, 3, 4]]	5	5	100	m3	m1: Az argumentumként átadott listának egyetlen eleme van (maga is
[[6], [5], [1, 2, 3, 4]]	6	6	101	m2	lista), ezért az argumentumot mergepairs második klóza
[[5, 6], [1, 2, 3, 4]]	3			m3	változtatás nélkül visszaadja az őt hívó sorting-nak.
[[7], [5, 6], [1, 2, 3, 4]]	7	7	110	m3	m2: n páros, ez azt jelzi, hogy az argumentumként átadott lista első
[[8], [7], [5, 6], [1, 2, 3, 4]]	8	8	111	m2	két eleme egyforma hosszú lista, amelyet merge egyetlen
[[7, 8], [5, 6], [1, 2, 3, 4]]	4			m2	rendezett listává futtat össze, majd az eredménnyel
[[5, 6, 7, 8], [1, 2, 3, 4]]	2			m2	mergepairs első klóza meghívja saját magát.
[[1, 2, 3, 4, 5, 6, 7, 8]]	1			m3	m3: n páratlan, ez azt jelzi, hogy az argumentumként átadott lista első
[[9], [1, 2, 3, 4, 5, 6, 7, 8]]	9	9	1000	m3	két eleme nem egyforma hosszú lista, ezért az argumentumot
[[9], [1, 2, 3, 4, 5, 6, 7, 8]]	0	0		m4	mergepairs első klóza változtatás nélkül visszaadja az őt hívó
[[1, 2, 3, 4, 5, 6, 7, 8, 9]]					sorting-nak.
					m4: n=0, az összes listák listáját olyan listává kell összefuttatni,
					amelynek egyetlen lista az eleme.

Simarendezés (folyt.)

- smsorting a futamokat ismételtelen előállítja és összefuttatja:


```
(* smsorting(xs, lss, k) = a még rendezetlen xs lista elemeit
    berakja a rendezett részlisták összesen
    már k elemet tartalmazó lss listájába
    smsorting : int list * int list list * int -> int list
    PRE: k >= 0
*)
fun smsorting (x::xs, lss, k) =
    let val (run, tail) = nextrun([x], xs)
    in smsorting(tail, mergepairs(run::lss, k+1), k+1)
    end
  | smsorting ([], lss, k) = hd(mergepairs(lss, 0))
```
- (* smsort xs = az xs elemeinek <= szerint rendezett listája


```
 smsort : int list -> int list
*)
fun smsort xs = smsorting(xs, [], 0)
```
- A simarendezés egy változata sort néven megtalálható a Listsort könyvtárban.

A futási idők összehasonlítása

```

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

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

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

```

Deklaratív programozás. BME VIK, 2003. tavaszi félév

(Funkcionális programozás)

Egyszerűsített SML-szintaxis FP-239

SML-szintaxis: típuskifejezések

- Típus (*ty*: type)

(31)	<i>ty</i> ::= <i>tyvar</i>	típusváltozó	type variable
(32)	<i>tycon</i>	típuskonstruktor	type constructor
(33)	{ < <i>tyrow</i> > }	rekordtípus-kifejezés	record type expression
(34)	<i>ty</i> ₁ * <i>ty</i> ₂	pár-típus	pair type
(35)	<i>ty</i> ₁ -> <i>ty</i> ₂	függvénytípus-kifejezés	function type expression
(36)	(<i>ty</i>)	típus zárójelben	parenthesized type

- Típuskifejezés-sor (*tyrow*: type-expression row)

(37) *tyrow* ::= *lab* : *ty* <, *tyrow*>

- Példák:

'a, 'c, 'gamma vö. (31)
 int, real, word, word8, char, bool, string, order vö. (32)
 int * int -> int, unit -> unit vö. (34), (35)
 ('a -> 'b) -> ('a list -> 'b list) vö. (35), (36)
 {num : int, den : int}, num : int, den : int vö. (33), (37)

Deklaratív programozás. BME VIK, 2003. tavaszi félév

(Funkcionális programozás)

EGYSZERŰSÍTETT SML-SZINTAXIS

SML-szintaxis: minták

- Atomi minta (*atpat*: atomic pattern)

(38)	<i>atpat</i> ::= <i>_</i>	mindenesjel	wildcard
(39)	<i>scon</i>	különleges állandó	special constant
(40)	<op> <i>longvid</i>	értéknév	value identifier
(41)	{ < <i>patrow</i> > }	rekord	record
(42)	(<i>pat</i> ₁ * <i>pat</i> ₂)	pár	pair
(43)	(), { }	nullas	0-tuple
(44)	[<i>pat</i> ₁ , ..., <i>pat</i> _{<i>n</i>}]	lista, <i>n</i> ≥ 0	list, <i>n</i> ≥ 0
(45)	(<i>pat</i>)	minta zárójelben	parenthesized pattern

- Példák:

fun le GREATER = false | le EQUAL = true | le LESS = true; vö. (40)
 fun le GREATER = false | le _ = true; vö. (38), (40)
 fun neg Bool.false = true | neg (true) = Bool.false; vö. (40), (45)
 fun prod [a, b] = a*b | prod [a, b, c] = a*b*c
 | prod [a] = a | prod () = 1; vö. (43), (44)

Deklaratív programozás. BME VIK, 2003. tavaszi félév

(Funkcionális programozás)

SML-szintaxis: minták (folyt.)

● Mintasor (*patrow*: pattern row)

(46)	<code>patrow ::= ...</code>	mindenesjel	wildcard
(47)	<code>lab = pat <, patrow></code>	mintasor	pattern row
(48)	<code>lab <: ty> <, patrow></code>	mezőnév mint változó	label as variable

● Példák:

```
fun // {den = 0, ...} = raise Domain
  | // {num = n, den = d} = (real n) / (real d); vö. (46), (47)
fun // {den = 0, ...} = raise Domain
  | // {num, den} = (real num) / (real den); vö. (46), (48)
```

SML-szintaxis: szintaktikai korlátozások

- Nem illeszthető minta kétszer ugyanarra a névre (*vid*). Nem illeszthető kifejezősor, mintasor vagy típuskifejezés-sor kétszer ugyanarra a mezőnévre (*lab*).
- Ugyanaz a név nem köthető le kétféleképpen egy *valbind*, *typbind*, *datbind* vagy *exbind* deklarációban. A *datbind* deklarációban ugyanez érvényes az adatkonstruktorokra is.
- Ugyanaz a típusváltozó (*tyvar*) nem szerepelhet kétszer egy *tyvarseq* sorozatban valamely *typbind* vagy *datbind* deklaráció bal oldali *tyvarseq tycon* részében. Minden olyan típusváltozónak (*tyvar*), amelyik előfordul a jobb oldalon, szerepelnie kell *tyvarseq*-ben.
- A *rec*-et követő minden *pat = exp* értékkötésben az *exp*-nek, szükség esetén zárójelben, *fn match* alakúnak kell lennie, ahol egy vagy több névhez típusmegkötés is társítható.
- *true*, *false*, *nil*, *::* és *ref* nem kaphat értéket *valbind*, *datbind* vagy *exbind*, *it* pedig *datbind* vagy *exbind* deklarációban.

SML-szintaxis: minták (folyt.)

● Minta (*pat*: pattern)

(49)	<code>pat ::= atpat</code>	atomi minta	atomic pattern
(50)	<code><op> longvid atpat</code>	értékkonstrució	value construction
(51)	<code>pat₁ vid pat₂</code>	infix értékkonstrució	infix value constr.
(52)	<code>pat : ty</code>	minta típusmegkötéssel	typed pattern
(53)	<code><op> var <: ty> as pat</code>	réteges minta	layered pattern

● Példa:

```
fun sum [] = 0 vö. (50)
  | sum [a : real] = a vö. (52)
  | sum (x::z::(yxs as y::xs)) = x + z + sum yxs vö. (51), (53)
  | sum (x::y::xs) = x + y + sum xs vö. (51)
  | sum (op::(x, xs)) = x + sum xs vö. (50)
```

AZ SML MODULNYELVE

Modulok

Mi a modul?

- fordítási egység
- a *csopontosítás* és az *elrejtés*, azaz az *absztrakció* eszköze

Modulok az SML-ben

- Szignatúra (*signature*): specifikációs modul; a struktúra *specifikációja*, „*típusa*”.
- Struktúra (*structure*): implementációs modul; a szignatúra *megvalósítása*.
- Funktor (*functor*): generikus, azaz *struktúrával* paraméterezhető modul; eredménye is egy *struktúra*.

Egy struktúra akkor és csak akkor valósít meg egy szignatúrát, ha a struktúra kielégíti a szignatúra által támasztott összes követelményt. (Később pontosítjuk.)

Szignatúra és struktúra (folyt.)

A szignatúra-deklaráció

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

A struktúra-deklaráció egyszerű változata

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

A struktúra-deklaráció bonyolultabb változata

- struktúra szignatúrához kötése; amely kétféle lehet:
 - áttetsző (opál, opaque): `structure strid := sigid = strexp`
 - átlátszó (transparens, transparent): `structure strid : sigid = strexp`

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.

Példa: a KCsiga és a Csigá szignatúra

- A KCsiga struktúra (a keretprogram) szignatúrája

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

- A Csigá struktúra (a főmodul) szignatúrája

```
signature Csigá =
sig
  val buvosCsigá :
    TCsiga.feladvanyleiro -> TCsiga.csigatabla list
end
```

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

Példa: a TCsiga struktúra és törzsszignatúrája

- A TCsiga struktúra (a típusleíró modul) és törzsszignatúrája

```

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

  type feladvanyleiro =
    meret * ciklus * adottElem list

  type ertekVagyUres = int
  type sor           =
    ertekVagyUres list

  type csigatabla =
    sor list
end
  (* TCsiga.sml
  törzsszignatúrája *)
  type meret      = int
  type ciklus    = int
  type sorszam   = int
  type oszlopszam = int
  type ertek     = int
  type adottElem =
    int * int * int

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

  type ertekVagyUres = int
  type sor           =
    int list

  type csigatabla =
    int list list

```

Példa: változatok a TCsiga struktúrára és szignatúrára

- Struktúra *átlátszó* szignatúrával

```

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* szignatúrával

```

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

```

- Szignatúra (a részleteket *elrejtő*) absztrakt adattípus megvalósításához

```

signature TCsiga =
sig
  type feladvanyleiro
  type csigatabla
end

```

Példa: a TCsiga struktúra és törzsszignatúrája (folyt.)

- *Törzsszignatúra* (principal signature): egy struktúra összetevőinek legspecifikusabb leírása.
- TCsiga csupa típusspecifikációt tartalmaz


```

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

```
- Egy struktúra szignatúra-megkötés nélkül nem rejt el a részleteket, azaz *gyenge absztrakciót* valósít meg.

AZ SML MODULNYELVE

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
```

típuspecifikáció
kivételspecifikáció
értékpecifikációk

Minden típus és érték polimorf!

típuskonstruktor deklarálása
kivételkonstruktor deklarálása
névkonstruktorok deklarálása

Deklaratív programozás. BME VIK, 2003. tavaszi félév

(Funkcionális programozás)

Az SML modulnyelve FP-256

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.

Deklaratív programozás. BME VIK, 2003. tavaszi félév

(Funkcionális programozás)

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
```

Deklaratív programozás. BME VIK, 2003. tavaszi félév

(Funkcionális programozás)

Az SML modulnyelve FP-257

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
```

Deklaratív programozás. BME VIK, 2003. tavaszi félév

(Funkcionális programozás)

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
```

Deklaratív programozás. BME VIK, 2003. tavaszi félév

(Funkcionális programozás)

Az SML modulnyelve FP-260

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).

Deklaratív programozás. BME VIK, 2003. tavaszi félév

(Funkcionális programozás)

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
```

Deklaratív programozás. BME VIK, 2003. tavaszi félév

(Funkcionális programozás)

Az SML modulnyelve FP-261

Szignatúra-illeszkedés

- Mikor tekinthető egy struktúra egy szignatúra megvalósításának?

Akkor, ha a struktúra *az összes olyan komponenszt definiálja és az összes olyan típusdefiniációt 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 komponenszt* 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.

Deklaratív programozás. BME VIK, 2003. tavaszi félév

(Funkcionális programozás)

Törzstípus, törzsszignatúra

- Egy érték *törzstípusa* (principal type) az adott értékhez rendelhető *legáltalánosabb* típus.
- Minden jóldefiniált értéknek van törzstípusa.
Pl. ha `fun I x = x`, akkor `I : 'a -> '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
 - `type (tyvar1, ..., tyvarn) tycon = typ` alakú, akkor a törzsszignatúra az ezzel azonos specifikációt tartalmazza;
 - `datatype (tyvar1, ..., tyvarn) tycon = con1 of typ1 | ... | conk of typk` alakú, akkor a törzsszignatúra az ezzel azonos specifikációt tartalmazza;
 - `exception id of typ` alakú, akkor a törzsszignatúra az ezzel azonos specifikációt tartalmazza;
 - `val id = exp` alakú, akkor a törzsszignatúra a `val id : typ` specifikációt tartalmazza, ahol `typ` az `exp` kifejezés törzstípusa.

Deklaratív programozás. BME VIK, 2003. tavaszi félév

(Funkcionális programozás)

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.

Deklaratív programozás. BME VIK, 2003. tavaszi félév

(Funkcionális programozás)

## Szignatúra-illeszkedés (folyt.)

- Egy *szignatúra-jelölt* (candidate signature) akkor és csak akkor illeszkedik egy *célszignatúrára* (target signature), ha az összes olyan komponenset 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.

Deklaratív programozás. BME VIK, 2003. tavaszi félév

(Funkcionális programozás)

## 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.

Deklaratív programozás. BME VIK, 2003. tavaszi félév

(Funkcionális programozás)

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 (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-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-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 = 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 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 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 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 átlátszó szignatúra-kötéssel: ORDERED, MyString

- Átlátszó szignatúra-kötéssel csökkenthető az explicit típus-specifikációk 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 szövegkörnyezetben 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 áttetsző szignatúra-kötéssel: prioritási sor (folyt.)

- A megvalósítástól független absztrakt típus *áttetsző szignatúráját* 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 (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 elrejtí. `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 elrejtteni, 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.

MODULOK HIERARCHIÁJA

Átlátszóság, áttetszőség, függőség

- Az átlátszó szignatúra-kötés a típusvezetéshez 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 függnek egymástól a modulok, annál könnyebben rakhatók össze a végén egyetlen rendszerré.

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 *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 *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 *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!

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
```

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
```
- Füzek 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: 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

```

FUNKTOROK

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.

Paraméterezhető, más néven generikus modulok

- Az újrafelhasználhatóságot segíti elő a *paraméterezhető*, 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.

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 megadott áttetsző, a bővített átlátszó szignatúrákötés 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 típusdefiniciónak 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 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.

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 LISTÁK

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

- 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.)

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 futatórendszer mindig feldolgozza.**

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.

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
```

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 sortozatot 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)
```

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
```

Prímszámok előállítását *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 (folyt.)

- Ha a `nextrandom`-ot 1.0 és 2147483647.0 közötti `seed`-re alkalmazzuk, ugyanebbe a tartományba eső más értéket állít elő az `a * seed mod 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 egészre:


```
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 $x_{k+1} = \frac{a/x_k + x_k}{2}$ 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.)

- 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 groot egy „tisztább” változata:

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

Négyzetgyökvonás Newton-Raphson módszerrel (folyt.)

- Ezzel

```
(* groot : real -> real
*)
fun groot 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 ($|x - y| < \epsilon$) teszteljük, de vizsgálhatnánk pl. a relatív különbséget ($|\frac{x}{y} - 1| < \epsilon$) vagy az $\frac{|x-y|}{\frac{|x|+|y|}{2}} < \epsilon$ feltételt.
- A feladat többi része független attól, hogy milyen leállásvizsgálatot alkalmazunk, és így is kell megfogalmazni a megoldást.

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 (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 simítani 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

- 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 kiírható `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.)

- `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
```

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.

MODULOK SZIMMETRIKUS ÖSSZEKAPCSOLÁSA

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 egységeselemét (`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 értékeket 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 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 szignatúrákötést használunk, a látható szignatúráik mind különbözőnek!
- *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 különböző volta

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

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

- Az *áttetsző* szignatúrákötésnek 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`-tól, hanem pl. `Geom2D.Sphere.Vector` is különbözik `Geom2D.Point.Vector`-tól!
- 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 (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 típusegyenletek mindig teljesüljenek, amikor a GEOMETRY szignatúrákat 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 (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

- 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.)

- 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 EXT_D_GEOMETRY-ben, GEOMETRY kiegészített, de a Point alstruktúra nélküli változatában:

```
signature EXT_D_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 programelemeinek 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 teremt meg az egész program koherenciáját. Ez a megközelítés az SML – más nyelvekben ismeretlen – sajátossága.

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

- 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.)

- 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 EXT_D_GEOMETRY =
sig
  structure Sphere : SPHERE
  structure SemiSpace : SEMI_SPACE
  sharing Sphere.Point = SemiSpace.Point
end
```

A típusmegosztás elkerülése funktor használatakor (folyt.)

- Az első lehetőség az, hogy ExtGeomFun 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 EXT_D_GEOMETRY szignatúrát ezzel a funktorral valósítjuk meg:

```
functor ExtGeomFun
  (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 EXT_D_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 ExtGeomFun funktor első változatával, ExtGeomFun_1-gyel több gond van:

```
functor ExtGeomFun_1
  (structure P : POINT) : GEOMETRY =
struct
  structure Sphere = SphereFun (structure P = Point)
  structure SemiSpace = SemiSpaceFun (structure P = Point)
end
```

- ExtGeomFun_1-ben alstruktúra-definícióban fordul elő SphereFun és SemiSpaceFun, és ez olyan paraméterekre korlátozza ExtGeomFun_1-et, amelyek e két funktorral állíthatók elő. – Ez erős korlátozás ExtGeomFun-hoz képest, amely SPHERE, ill. SEMI_SPACE bármely megvalósításának alkalmazását lehetővé teszi.
- ExtGeomFun_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 ExtGeomFun_1 miért éppen POINT egy megvalósítását kapja paraméterként. (Nem elég nyomós) oka az, hogy ExtGeomFun_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 kötik SPHERE, ill. SEMI_SPACE (ugyancsak paraméterként átveendő) megvalósítását, és így érjük el a típusgeyenletek 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 vezetünk be, amelynek az egyetlen szerepe az, hogy elhagyhassuk a megosztási előírást.

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 típusmegosztás elkerülése funktor használatakor (folyt.)

- ExtdGeomFun_2 rafináltabb változata ExtdGeomFun_3 és ExtdGeomFun_4. Mindkettőnek csak két paraméterre van szüksége azáltal, 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
```