

FUNKCIONÁLIS PROGRAMOZÁS

Bevezető FP-1..12-2

Tartalom

Az első rész tartalma

- Bevezető
- Absztrakció függvényekkel (eljárásokkal)
 - Programelemek
 - Függvények (eljárások) és az általuk létrehozott folyamatok
 - Magasabb rendű függvények (eljárások)
- Absztrakció adatokkal
 - Az adatabsztrakció fogalma
 - Hierarchikus adatok
 - Absztrakt adatok többszörös ábrázolása
 - Polimorfizmus, generikus műveletek

Irodalom: [SICP] Abelson, Sussman & Sussman: *Structure and Interpretation of Computer Programs*, The MIT Press, 1996

Történeti áttekintés

Funkcionális programozási nyelvek

- 1930-40-es évek: *Alonzo Church: λ -kalkulus*
- LISP (LISt Processing), 1950-es évek vége, MIT, US, John McCarthy; típus nélküli
 - bizonyos logikai kifejezések (ún. rekurzív egyenletek) igazolására
 - szimbolikus kifejezések kezelésére
- ML (Meta Language), Edinborough, GB, 1970-es évek közepe; típusos
- Scheme, 1975, MIT, US; típus nélküli
- SML (Standard Meta Language), 1980-as évek vége; típusos
- Miranda, 1980-as évek; típusos
- Haskell, 1990-es évek, US; típusos
- Common LISP, 1994, ANSI standard; típus nélküli
- Clean, 1990-es évek közepe, Nijmegen, NL; típusos
- Alice, 2003, Saarbrücken, DE; típusos

Funkcionális programozás

Ami közös a funkcionális nyelvekben

- Rekurzív eljárások (függvények)
- Rekurzív adatstruktúrák
- Eljárások (függvények) kezelése adatként

A következő hetekben

- *számítási folyamatokkal* és az általuk kezelt *adatokkal* foglalkozunk,
- programjainkat – a folyamatokat vezérlő szabályrendszert – az SML funkcionális nyelven írjuk,
- *eszközül* az MOSML vagy a PolyML értelmezőt/fordítót használjuk,
- az absztrakcióról, modellezésről, programstruktúráról tanulandók más programozási nyelvek használatakor is hasznosak lesznek.

ABSZTRAKCIÓ FÜGGVÉNYEKKEL (ELJÁRÁSOKKAL)

Absztrakció függvényekkel (eljárásokkal) FP-1..12-6

Programelemek

Minden használható programozási nyelvben háromféle mechanizmus van a számítási folyamatok és az adatok leírására:

- kifejezések
- összetételi eszközök
- absztrakciós eszközök (pl. névadás)

Kifejezések az SML-ben

- elemiek: nevek, továbbá állandók (jelölések), pl. 486 , 2.0 , "text", # "A"
- összetett kifejezések, pl. $482+4$, $2.3-0.3$, "te" ^ "xt", $op+(482, 4)$, # "A" < # "a"

Megjegyzések: < és # ún. tapadó írásjelek, ezért közéjük szóközt kell rakni a példában. Az op kulcsszó egy infix helyzetű operátort átmenetileg prefix helyzetűvé tesz.

Összetételi eszközök

- operátor (műveleti jel, függvényjel)
- operandus (formális paraméter)
- argumentum (aktuális paraméter)
- rekurzió

Példák az MOSML használatára

Az SML értelmező is ún. *read-eval-print* ciklusban dolgozik. A kiértékelés (*eval*) a `;`, majd az *enter* leütésére kezdődik el.

```
Moscow ML version 2.00 (June 2000)
Enter 'quit();' to quit.
- 486;
> val it = 486 : int
- 2.3-0.3;
> val it = 2.0 : real
- "te" ^ "xt";
> val it = "text" : string
- op+(482,4);
> val it = 486 : int
- #"A" < #"a";
> val it = true : bool
- val it = 486;
> val it = 486 : int
```

Minden kifejezés kiértékelése valójában *értékdeklaráció*: ha nem adunk meg más nevet, az SML az `it` nevet köti az adott kifejezés értékéhez.

Névadás, (globális) környezet

Egy *értékdeklarációval* egy nevet kötünk egy értékhez:

```
- val size = 2;
> val size = 2 : int
- 5*size;
> val it = 10 : int
- val ||| = 3;
> val ||| = 3 : int
- ||| * size;
> val it = 6 : int
```

Megjegyzés: `|` és `*` ún. tapadó írásjelek, ezért közéjük szóközt kell rakni a példában.

Egy név lehet:

- alfanumerikus (az angol ábécé kis- és nagybetűiből, az `_` és a `'` jelekből állhat, betűvel kell kezdődnie),
- írásjelekből álló (húszféle írásjel használható).

A *névadás* a legegyszerűbb absztrakciós eszköz (a programozási nyelvekben is).

A *név-érték* párt az SML a „memóriájában”, az ún. globális *környezetben* tárolja. Később látni fogjuk, hogy vannak ún. lokális környezetek is.

Nevek képzési szabályai

- Alfanumerikus név: kis- és nagybetűk, számjegyek, percjelek (') és aláhúzás-jelek (_) olyan sorozata, amely betűvel vagy percjellel kezdődik

- Példák: `tothGyorgy` `Toth_3_Gyorgy` `toth'gyorgy` `'gyurika`
- Percjellel kezdődő név csak típusváltozót jelölhet.

- Írásjelekből álló név: az alábbi 20 *tapadó* írásjel 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 : :: :> _ | = => -> #
```

Egyszerű adattípusok

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

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
	/	real * real -> real	hányados	Div, Overflow
	div, mod	<i>wordint</i> * <i>wordint</i> -> <i>wordint</i>	hányados, maradék	Div, Overflow
	quot, rem	int * int -> int	hányados, maradék	Div, Overflow
6	+, -	<i>num</i> * <i>num</i> -> <i>num</i>	összeg, különbség	Overflow
	^	string * string -> string	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 $-\infty$, 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).

Különleges állandók

- Előjeles egész állandó

Példák: 0 ~0 4 ~04 999999 0xFFFF ~0x1fff

Ellenpéldák: 0.0 ~0.0 4.0 1E0 -317 0XFFFF -0x1fff

- 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 0w1fff

Ellenpéldák: 0w0.0 ~0w4 -0w4 0w1E0 0wXFFFF 0WxFFFF

- 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

Különleges állandók, escape-szekvenciák

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

Összetett kifejezés kiértékelése

Egy összetett kifejezést az SML két lépésben értékeli ki (ez az ún. mohó kiértékelés):

1. először kiértékeli az operátort (műveleti jelet, függvényjelet) és argumentumait (aktuális paramétereit),
2. majd alkalmazza az operátort az argumentumokra.

Vegyük észre, hogy ez a kiértékelési szabály azért ilyen egyszerű, mert rekurzív.

Az egyszerű kifejezések kiértékelési szabályai:

1. az állandók (jelölések) értéke az, amit jelölnek,
2. a belső (beépített) műveletek a megfelelő gépi utasításokat aktivizálják,
3. a nevek értéke az, amihez a környezet köti az adott nevet.

Megjegyzés: a 2. pont a 3. pont speciális esetének is tekinthető.

Példa:

$$(2+4*6) * (3+5+7) = \text{op}^*(\text{op}+(2, \text{op}^*(4, 6)), \text{op}+(\text{op}+(3, 5), 7))$$

A kifejezéseket ún. kifejezésfával ábrázolhatjuk (lásd SICP-ben). A levelek operátorok vagy primitív kifejezések (állandók, nevek). A kiértékelés során az operandusok alulról fölfelé „terjednek”.

Névtelen függvény, függvény definiálása

Névtelen függvény λ -jelöléssel: pl. $(fn\ x \Rightarrow x*x)$

Névtelen függvény alkalmazása: pl. $(fn\ x \Rightarrow x*x)\ 2$

- Az fn -t *lambdának* olvassuk.
- Az x a függvény formális paramétere (lokális [érvényű] név!).
- Az $x*x$ a függvény törzse.
- A 2 a függvény aktuális paramétere.

Névtelen függvényértéknek (azaz függvénynév deklarációja):

```
val square = fn x => x * x
```

```
val sumOfSquares = fn (x, y) => square x + square y
```

```
val f = fn a => sumOfSquares(a+1, a*2)
```

A felhasználó által definiált függvények ugyanúgy használhatók, mint a belső (beépített) függvények.

További példák SML-függvények definiálására

Egyszeres Hamming-távolságú ciklikus kód előállítás

- A függvényt pl. *táblázattal* adhatjuk meg:

00	01	fn	00 =>	01
01	11			01 =>
11	10			11 =>
10	00			10 =>

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

További példák SML-függvények definiálására (folyt.)

Modulo n alapú inkrementálás (pl. $n = 5$)

- A függvényt általában *algoritmussal* adjuk meg (nem táblázattal), egyébként
 - az argumentum nem lehetne változó,
 - túl sok változatot kellene felírni stb.
- `fn i => (i + 1) mod 5`
 - az `i` ún. *kötött változó*, a névtelen függvény argumentuma
- A függvény néhány alkalmazása:
 - `(fn i => (i + 1) mod 5) 2`
 - `(fn i => (i + 1) mod 5) 4`
 - `(fn i => (i + 1) mod 5) 3.0` – Hiba!
- Ez a függvény definiálható két klózzal is (a sorrend fontos!):
`fn 4 => 0 | i => i + 1`
- Az SML (a Prologgal ellentétben) mindig csak az *első* illeszthető klózt használja!
- A függvény egyik változata sem robusztus. Melyik a szerencsésebb?

Függvényérték névhez kötése (függvényérték deklarálása)

- Láttuk, hogy nevet függvényértékhez ugyanúgy köthetünk, mint bármely más értékhez.
 - `val kovKod = fn 00 => 01 | 01 => 11 | 11 => 10 | 10 => 00`
 - `val incMod = fn 4 => 0 | i => i + 1`
- Szintaktikus édesítőszerrel (`fun`)
 - `fun kovKod 00 = 01`
 | `kovKod 01 = 11`
 | `kovKod 11 = 10`
 | `kovKod 10 = 00`
 - `fun incMod 4 = 0`
 | `incMod i = i + 1`
- Alkalmazásuk argumentumra
 - `kovKod 01`
 - `incMod 4`

Fejkomment

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

- (* kovKod cc = az egyszeres Hamming-távolságú, kétbites, ciklikus kódkészlet cc-t követő eleme
PRE: $cc \in \{00, 01, 11, 10\}$
*)
fun kovKod 00 = 01
| kovKod 01 = 11
| kovKod 11 = 10
| kovKod 10 = 00
- PRE = *precondition*, előfeltétel
- PRE: $cc \in \{00, 01, 11, 10\}$ jelentése: a kovKod függvény cc argumentumának a $\{00, 01, 11, 10\}$ halmazbeli értéknek kell lennie, ellenkező esetben a függvény eredménye nincs definiálva.
- (* incMod i = (i+1) modulo 5 szerint
PRE: $5 > i \geq 0$
*)
fun incMod i = (i+1) mod 5

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
 - round 5.4 = 5, azaz egy *egész* típusú érték az eredménye ennek a függvényalkalmazásnak
 - round \circ sin (a típusa: *valós* \rightarrow *egész*)
 - (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. Az infix operátor balra vagy jobbra köt.

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:
 $\text{Math.sin } 1.00$ (Math.cos) Math.pi $\text{round}(3.17)$
 $2 + 3$ (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. square , $/\backslash$, head

A függvényalkalmazás kiértékelése

Egy felhasználói függvényeket tartalmazó kifejezést az összetett kifejezéshez hasonlóan értékel ki az SML. Ott hallgatólagosan feltettük, hogy az SML tudja, hogyan alkalmazza a belső függvényeket az aktuális paramétereikre.

Ezt most azzal egészítjük ki, hogy az SML egy függvény alkalmazásakor

- a függvény törzsében a formális paraméterek összes előfordulását lecseréli a megfelelő aktuális paraméterre, majd kiértékeli a függvény törzsét.

Nézzük az $f\ 5$ kifejezés kiértékelését! Minden lépésben egy részkifejezést egy vele egyenértékű kifejezéssel helyettesítünk.

$$\begin{aligned} f\ 5 &\rightarrow \text{sumOfSquares}(5+1, 5*2) \rightarrow \text{sumOfSquares}(6, 5*2) \rightarrow \\ &\text{sumOfSquares}(6, 10) \rightarrow \text{square } 6 + \text{square } 10 \rightarrow 6*6 + \text{square } 10 \rightarrow \\ &36 + \text{square } 10 \rightarrow 36 + 10*10 \rightarrow 36 + 100 \rightarrow 136 \end{aligned}$$

($\text{val sumOfSquares} = \text{fn } (x, y) => \text{square } x + \text{square } y; \text{val square} = \text{fn } x => x * x$)
 A függvényalkalmazás itt bemutatott *helyettesítési modellje* – egyenlők helyettesítése egyenlőkkel, *equals replaced by equals* – segíti a függvényalkalmazás *jelentésének* megértését. Olyan esetekben alkalmazható, amikor egy függvény jelentése független a környezetétől.

Az értelmezők/fordítók rendszerint más, bonyolultabb modell szerint működnek.

Applikatív sorrend (mohó kiértékelés), normál sorrend (lusta kiértékelés)

- Az összetett kifejezés kiértékelésénél leírtak szerint az SML először kiértékeli az operátort és argumentumait, majd alkalmazza az operátort az argumentumokra. Ezt a kiértékelési sorrendet *applikatív sorrendű* (applicative order) vagy *mohó* (eager) kiértékelésnek nevezzük.
- Van más lehetőség is: a kiértékelést addig halogatjuk, ameddig csak lehetséges. Ezt *normál sorrendű* (normal order), *szükség szerinti* (by need) vagy *lusta* (lazy) kiértékelésnek nevezzük.

Nézzük $f\ 5$ kiértékelését, ha az SML lusta kiértékelést alkalmazna:

```
f 5 → sumOfSquares(5+1, 5*2) → square(5+1) + square(5*2) →
(5+1)*(5+1) + (5*2)*(5*2) → 6*(5+1) + (5*2)*(5*2) → 6*6 + (5*2)*(5*2)
→ 36 + (5*2)*(5*2) → 36 + 10*(5*2) → 36 + 10*10 → 36 + 100 → 136
```

- Igazolható, hogy olyan függvények esetén, amelyek jelentésének megértésére a helyettesítési modell alkalmas, a kétféle kiértékelési sorrend azonos eredményt ad.
- Vegyük észre, hogy szükség szerinti kiértékelés mellett a példában egyes részkifejezéseket akár kétszer is ki kell értékelni.
- Ezen jobb értelmezők/fordítók (pl. Alice, Haskell) úgy segítenek, hogy az azonos részkifejezéseket megjelölik, és amikor egy részkifejezést először kiértékelnek, *az eredményét megjegyzik*, a többi előfordulásakor pedig ezt az eredményt veszik elő. E módszer hátránya a nyilvántartás szükségessége. Ma általában ezt nevezik *lusta* kiértékelésnek.

Feltételes kifejezések, logikai műveletek, predikátumok

- Típusnév: `bool`, adatkonstruktorok: `false`, `true`, beépített függvény: `not`.
- *Lusta kiértékelésű* beépített műveletek (speciális nyelvi elemek!)
 - 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.
- Mind a három logikai művelet 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!!!`

Feltételes kifejezések, logikai műveletek, predikátumok (folyt.)

Lássunk néhány példát!

```

val absolute = fn x => if      x < 0 then ~x
                    else if x > 0 then x
                    else      0

val absolute = fn x => if      x < 0 then ~x
                    else      x

use "sumOfSquares.sml";

val sumOfSquaresOfTwoLarger =
  fn (x,y,z) =>
    if      x < y andalso x < z then sumOfSquares(y, z)
    else if y < x andalso y < z then sumOfSquares(x, z)
    else      sumOfSquares(x, y);

```

Predikátumnak az olyan függvényt nevezzük, amelynek logikai (bool típusú) érték az eredménye, pl.

```

val isAlphaNum = fn c =>
  #"A" <= c andalso c <= #"Z" orelse
  #"a" <= c andalso c <= #"z" orelse
  #"0" <= c andalso c <= #"9"

```

Feltételes kifejezések, logikai műveletek, predikátumok (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> (*) && (a, b) = a /\ b && : bool * bool -> bool *) fun op&& (a, b) = a andalso b; infix 2 && </pre>	<pre> (*) (a, b) = a \/ b : bool * bool -> bool *) fun op (a, b) = a orelse b; infix 1 </pre>
--	---
- `infix prec név1 név2 ... : a név1 név2 ...` függvényeket *prec* precedenciaszintű, *infix* helyzetű, balra kötő *operátor*rá alakítja.

Négyzetgyökvonás Newton-módszerrel

- A funkcionális nyelvi függvények sokban hasonlítanak a matematikai függvényekhez: egy vagy több argumentumtól függő értéket adnak eredményül. Egy dologban azonban mindenképpen különböznek: a funkcionális nyelvi függvényeknek *hatékonyaknak* is kell lenniük.
- Nézzük pl. a négyzetgyök következő definícióját: $\sqrt{x} = y$, ahol $y \geq 0$ és $y^2 = x$.
- Ez az egyenletrendszer alkalmas pl. annak ellenőrzésére, hogy egy szám egy másiknak a négyzetgyöke-e, de nem alkalmas a négyzetgyök előállítására.
- A matematikai függvénnyel egy bizonyos tulajdonságot *deklarálunk*, a funkcionális nyelvi függvénnyel (eljárással) azt is megmondjuk, *hogyan kell kiszámítani* az adott értéket. (A deklaratív programozás tehát csak az imperatív programozáshoz *képest* tekinthető deklaratívnak. Vö. MIT és HOGYAN.)
- A négyzetgyökszámítás legismertebb módszere a *szukcesszív approximáció*: ha az y az x négyzetgyökének egy közelítése, akkor az y és az x/y átlaga a négyzetgyök egy jobb közelítése lesz. A lépéssorozat akkor fejeződik be, amikor a közelítő értéket már elég jónak tartjuk.
- Írjuk le ezt az algoritmust SML-nyelven (l. következő dia)!

Négyzetgyökvonás Newton-módszerrel (folyt.)

```
val rec sqrtIter =
  fn (guess, x) => if goodEnough(guess, x)
                  then guess
                  else sqrtIter(improved(guess, x), x)
```

- Itt a `rec` kulcsszó arra utal, hogy az értékdeklaráció *rekurzív*: a most deklarált nevet *használjuk* a deklaráció jobb oldalán, a függvény definíciójában.
- A megoldási stratégiánkat jól tükrözi a fenti programrészlet. Ezt a stílust *fölülről lefelé haladó* (top down) módszernek nevezik. Kezdetben nem foglalkozunk a részletekkel, feltesszük, hogy minden megvan, amire szükségünk van, legfeljebb később megírjuk.

Most tehát definiálnunk kell még néhány részletet.

```
val improved = fn (guess, x) => average(guess, x/guess)
val average = fn (x, y) => (x+y)/2.0
val goodEnough = fn (guess, x) => abs(square_r guess - x) < 0.001
val square_r = fn (x : real) => x * x
```

Végül meg kell hívnunk az `sqrtIter` függvényt a négyzetgyök első közelítő értékével.

```
val sqrt = fn x => sqrtIter(1.0, x);
```

Négyzetgyökvonás Newton-módszerrel (folyt.)

- Sajnos, gond van a deklarációk sorrendjével, az SML ugyanis azt igényli, hogy egy deklaráció jobb oldalán minden kifejezésnek legyen értéke.
- Ha a deklarációk sorrendjét megfordítjuk, a programszöveg kevésbé hűen tükrözi a követett módszert.
- Megoldást az ún. *egyidejű deklaráció* jelent, amely előbb beolvassa, majd egyidejűleg dolgozza fel az összes deklarációt. Az egyidejűleg deklarálni kívánt értékeket az `and` kulcsszóval kell elválasztani egymástól.

```
val rec sqrtIter =
  fn (guess, x) => if goodEnough(guess, x)
                  then guess
                  else sqrtIter(improved(guess, x), x)
and improved = fn (guess, x) => average(guess, x/guess)
and average = fn (x, y) => (x+y)/2.0
and goodEnough = fn (guess, x) => abs(square_r guess - x) < 0.001
and square_r = fn (x : real) => x * x
val sqrt = fn x => sqrtIter(1.0, x)
```

Négyzetgyökvonás Newton-módszerrel (folyt.)

- Eddigi absztrakciós eszközeink (a névadás, a függvény, ill. eljárás jók a dolgok *megnevezésére*, de alkalmatlanok bizonyos *részletek elrejtésére*.
- Elrejtésre az SML-ben több eszközünk is van, a legalapvetőbb maga a függvény, és ilyen a „kifejezés lokális érvényű deklarációval” (rövidebben „kifejezés lokális deklarációval”), másnéven `let`-kifejezés speciális nyelvi elem is.
- `let`-kifejezést használunk akkor is, ha ismétlődő részkifejezéseket *csak egyszer* akarunk kiszámítani.
- Szintaxisa:

<code>let</code>	<code>d</code>	<code>ahol</code>	<code>d</code> egy nemüres deklarációsorozat,
	<code>in</code>		<code>e</code> egy nemüres kifejezés.
	<code>end</code>		

A továbbiakban a függvénydefiníciókat a `fun` „szintaktikai édesítőszerrel” találjuk.

Négyzetgyökvonás Newton-módszerrel (folyt.)

```

fun sqrt x =
  let fun sqrtIter (guess, x) = if goodEnough(guess, x)
                                then guess
                                else sqrtIter(improved(guess, x), x)
      and improved (guess, x) = average(guess, x/guess)
      and average (x, y) = (x+y)/2.0
      and goodEnough (guess, x) = abs(square_r guess - x) < 0.001
      and square_r (x : real) = x * x
  in
    sqrtIter(1.0, x)
  end

```

- Az SML-ben a nevek *szövegekörnyezettől függő* érvényességi és láthatósági szabályai hasonlóak a más programozási nyelvekben megszokottakhoz.
- Az `sqrt` függvény `x` formális paramétere például *látható* a függvény törzsében definiált függvényekben is, hacsak egy azonos nevű formális paraméter el nem fedí. Az `sqrt`-ben *lokális* `x` *globális [érvényű] névként* használható a lokális függvénydefiníciókban.
- A `:real` *típusmegkötés* elhagyható: az SML a szövegekörnyezetből kitalálja a paraméter típusát.

Négyzetgyökvonás Newton-módszerrel (folyt.)

A könnyített változat:

```

fun sqrt x =
  let fun sqrtIter guess = if goodEnough guess
                          then guess
                          else sqrtIter(improved guess)
      and improved guess = average(guess, x/guess)
      and average (x, y) = (x+y)/2.0
      and goodEnough guess = abs(square_r guess - x) < 0.001
      and square_r x = x * x
  in
    sqrtIter 1.0
  end;

```

Azzal, hogy értelmes nevet adtunk az egyes programelemeknek, könnyebbé, egyszerűbbé tettük

- „az ügyek szétválasztásával” (*separation of concerns*) a program kidolgozását,
- a jövőbeli olvasóknak a megértését,
- a javítását (ha a segédfüggvényeknek nincsen *mellékhatásuk*, a specifikáció megtartása mellett bármikor lecserélhetők).

Eljárások (függvények) és folyamatok

- Az eljárások (függvények) olyan *minták*, amelyek megszabják a számítási folyamatok (processzek) menetét, *lokális* viselkedését.
- Egy számítási folyamat *globális* viselkedését (pl. a szükséges lépések számát, a végrehajtási időt) általában nehéz megbecsülni, de törekednünk kell rá.

Lineáris rekurzió és iteráció

- A faktoriális matematikai definíciójának hű tükörképe az alábbi SML-program:

```

(* PRE : n >= 0 *)
0! = 1                fun factorial 0 = 1
n! = n(n - 1)!      | factorial n = n * factorial(n-1)

```

- Ha a helyettesítési modellünket alkalmazzuk, láthatjuk, hogy a program által létrehozott folyamat az összes tényezőt n -től 1-ig eltárolja, mielőtt az első szorzást végrehajtaná („késlelteti” a szorzásokat) – a folyamat *lineáris-rekurzív folyamat*.
- Ha ehelyett 1-et szoroznánk 2-vel, majd a részszorzatokat 3-mal, 4-gyel s.í.t., akkor az n érték meghaladásakor az utolsó részszorzat éppen n faktoriálisa lenne! Ehhez a programban szükségünk van egy olyan *formális paraméterre* (tkp. lokális változóra), amely tárolja a részszorzat aktuális értékét, és egy másikra, amely 1-től n -ig számlál. A létrehozott folyamat *lineáris-iteratív folyamat*.

Lineáris rekurzió és iteráció (folyt.)

```

fun factorial n =
  let fun factIter (product, counter) =
        if counter > n
        then product
        else factIter(product*counter, counter+1)
      in
        factIter(1, 0)
      end

```

- `factIter` világosabb szerkezetű változatát kapjuk, ha a számlálót *lefelé* számláltatjuk.

```

(* PRE : n >= 0 *)
fun factorial n =
  let fun factIter (product, 0) =
        product
        | factIter (product, counter) =
            factIter(product*counter, counter-1)
      in
        factIter(1, n)
      end

```

Eljárások (függvények) és folyamatok

- Ne tévesszük össze egymással a rekurzív (számítási) folyamatot és a rekurzív függvényt (eljárást)!
- Egy rekurzív függvény esetén csupán a szintaxisról van szó, arról, hogy hivatkozik-e a függvény (eljárás) önmagára.
- A folyamat esetében viszont a folyamat menetéről, lefolyásáról beszélünk.
- Ha egy függvény *jobbrekurzív (tail-recursive)*, a megfelelő folyamat – az értelmező/fordító jószágától függően – még lehet iteratív.

Még visszatérünk az „absztrakció függvényekkel” témakörhöz, de most témát váltunk: megismerkedünk a *paraméteres polimorfizmus* fogalmával, majd egy nagyon funkcionális adatszerkezettel, a listával foglalkozunk.

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. pl. objektum-orientált programozás). Az ún. *genericitás* is tekinthető az öröklődéses polimorfizmus egy változatának.

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*, röviden *állandó*.
- A *nil* helyett általában a `[]` jelet használjuk (szintaktikus édesítőszer).
- 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 *adatkonstruktoroperátor*nak 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	
<code>[]</code>	<code>[]</code>	azonos	<code>(x :: xs)</code>	<code>[X Xs]</code>	különböző
<code>[1, 2, 3]</code>	<code>[1, 2, 3]</code>	azonos	<code>(x :: y :: ys)</code>	<code>[X, Y Ys]</code>	különböző

Minták

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

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!

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;
```

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

map: adott függvény alkalmazása egy lista minden elemére

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

```
load "Math";
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  $\equiv$  ('a -> 'b) -> ('a list -> 'b list)
```

- A map egy *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.

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 (visszavezethető 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 *alap 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 #" \t" = true; Char.isAlphaNum #"!" = false
```



## filter: adott predikátumot kielégítő elemek kiválogatása

- Példa: gyűjtsük 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 (a -> 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) típusú függvényt ad eredményül. A kapott függvényt egy 'a list típusú listára alkalmazva egy 'a list típusú listát kapunk.

## Lista legnagyobb elemének megkeresése

- Üres listának nincs legnagyobb eleme,
- egyelemű lista egyetlen eleme a „legnagyobb”,
- legalább kételemű lista legnagyobb eleme

- az első elem és a maradéklista elemeinek legnagyobbika közül a nagyobb:

|                                                                                                                                                                                                      |                                                                                                                                                                                                     |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>load "Int"; (* 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>max egy változata egészekre: (* 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> |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

- az első két elem legnagyobbika és a maradéklista elemei közül a legnagyobb:

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

- maxl-lel szemben itt a klózik sorrendje közömbös (a minták diszjunktak).
- maxl' *jobbrekurzív*, tárigénye konstans.

## Lista legnagyobb elemének megkeresése (folyt.)

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

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

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

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

## 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 lexikografikusan 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`;

## 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)

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

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 megfordítása: példa nrev alkalmazására

- Egy példa nrev egyszerűsítésére

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

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

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

```
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 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.

## ÖSSZETETT ADATTÍPUSOK

---

## Rekord és ennes

---

- Két különböző típusú értékből rekordot vagy párt képezhetünk. Pl.

$\{x = 2, y = 1.0\} : \{x : \text{int}, y : \text{real}\}$  és  $(2, 1.0) : (\text{int} * \text{real})$

- A pár is csak szintaktikus édesítőszer. 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.

$\{\text{nev} = \text{"Bea"}, \text{tel} = 3192144, \text{kor} = 19\} : \{\text{kor} : \text{int}, \text{nev} : \text{string}, \text{tel} : \text{int}\}$

Egy hasonló rekord egészszám-mezőnevekkel:

$\{1 = \text{"Bea"}, 3 = 3192144, 2 = 19\} : \{1 : \text{string}, 2 : \text{int}, 3 : \text{int}\}$

Az *utóbbi* azonos az alábbi *ennessel* (n-tuple):

$(\text{"Bea"}, 19, 3192144) : (\text{string} * \text{int} * \text{int})$

azaz

$(\text{string} * \text{int} * \text{int}) \equiv \{1 = \text{string}, 2 = \text{int}, 3 = \text{int}\}$

- Egy rekordban a tagok sorrendje közömbös, a tagokat a mezőnév azonosítja. Egy ennesben a tagok sorrendje nem közömbös, a tagokat a *pozícionális* mezőnév azonosítja.

## Adattípusok: gyenge és erős absztrakció

- Gyenge absztrakció: a név szinonima, az adatszerkezet részei továbbra is hozzáférhetőek.
- Erős absztrakció: a név új dolgot (entitást, objektumot) 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.

## Adattípusok: felsorolásos és polimorf típusok `datatype` deklarációval

- `datatype logi = Igen | Nem` | Felsorolásos típus.  
`datatype logi3 = igen | nem | talan` | Felsorolásos típus.  
`datatype 'a esetleg = Semmi | Valami of 'a` | 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 értéknevek.
- Természetesen az adatkonstruktoroknak is van típusuk, pl.
 

|                   |                     |                     |                                    |
|-------------------|---------------------|---------------------|------------------------------------|
| <code>Igen</code> | <code>: logi</code> | <code>Semmi</code>  | <code>: 'a esetleg</code>          |
| <code>Nem</code>  | <code>: logi</code> | <code>Valami</code> | <code>: 'a -&gt; 'a esetleg</code> |
- Példa `datatype` deklarációval létrehozott adattípust kezelő függvényre
   
`fun inverz Nem = Igen | inverz Igen = Nem`

# FONTOS APRÓSÁGOK

Nullas, unit, print, szekvenciális kifejezés, before operátor FP-1..12-62

## 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
```

# ABSZTRAKCIÓ FÜGGVÉNYEKKEL (ELJÁRÁSOKKAL)

Absztrakció függvényekkel (eljárásokkal) FP-1..12-64

## Elágazó rekurzió

- Korábban lineáris-rekurzív, ill. lineáris-iteratív folyamatokra láttunk példákat (faktoriális kiszámítása kétféleképpen).
- Most *elágazó rekurzióra* nézzünk példát: állítsuk elő a Fibonacci-számok sorozatát.
- Egy Fibonacci-számot az előző két Fibonacci-szám összege adja:

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

- A Fibonacci-számok matematikai definíciója könnyen átírható SML-függvénnyé:

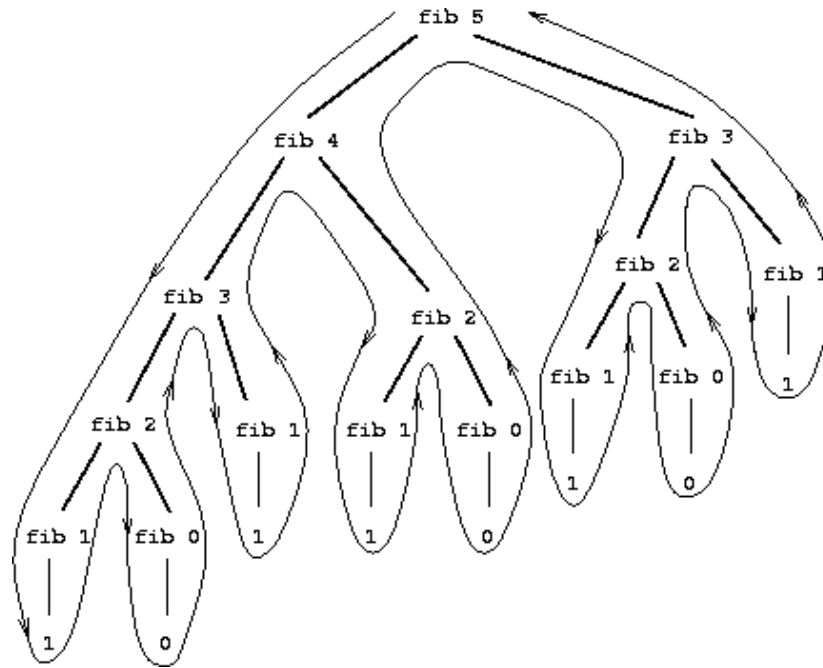
|                                       |  |                             |
|---------------------------------------|--|-----------------------------|
| $F(0) = 0$                            |  | fun fib 0 = 0               |
| $F(1) = 1$                            |  | fib 1 = 1                   |
| $F(n) = F(n-1) + F(n-2)$ , ha $n > 1$ |  | fib n = fib(n-1) + fib(n-2) |

Emlékeztetőül: a `fib` függvény definíciójában a 3. klóznak az utolsónak kell lennie, mert az `n` minta minden argumentumra illeszkedik.

- A következő lapon látható ábra illusztrálja az elágazóan rekurzív folyamatot `fib 5` kiszámítása esetén.



## Elágazó rekurzió (folyt.)



- fib 5-öt fib 4 és fib 3, fib 4-et fib 3 és fib 2 kiszámításával stb. kapjuk.

## Elágazó rekurzió (folyt.)

- Az előző program alkalmas az elágazó rekurzió lényegének bemutatására, de szinte alkalmatlan a Fibonacci-számok előállítására!
- Vegyük észre, hogy pl. fib 3-at kétszer is kiszámítjuk, azaz a munkának ezt a részét kb. a harmadát feleslegesen végezzük el.
- Belátható, hogy  $F(n)$  meghatározásához pontosan  $F(n+1)$  levélből álló fát kell bejárni, azaz ennyiszor kell meghatározni  $F(0)$ -t vagy  $F(1)$ -et.
- $F(n)$  exponenciálisan nő  $n$ -nel.  
Pontosabban,  $F(n)$  a  $\Phi^n / \sqrt{5}$ -höz közel eső egész, ahol  $\Phi = (1 + \sqrt{5})/2 \approx 1.61803$ , az ún. aranymetszés arányszáma.  $\Phi$  kielégíti a  $\Phi^2 = \Phi + 1$  egyenletet.
- A megteendő lépések száma tehát  $F(n)$ -nel együtt exponenciálisan nő  $n$ -nel. Ugyanakkor a tárigény csak lineárisan nő  $n$ -nel, mert csak azt kell nyilvántartani, hogy hányadik szinten járunk a fában.
- Általában is igaz, hogy elágazó rekurzió esetén a lépések száma a fa csomópontjainak a számával, a tárigény viszont a fa maximális mélységével arányos.

## Elágazó rekurzió (folyt.)

---

- A Fibonacci-számok lineáris-iteratív folyamattal is előállíthatók.

Ha az  $a$  és  $b$  változók kezdőértéke rendre  $F(1) = 1$  és  $F(0) = 0$ , és ismétlődően alkalmazzuk az  $a \leftarrow a + b, b \leftarrow a$  transzformációkat, akkor  $n$  lépés után  $a = F(n + 1)$  és  $b = F(n)$  lesz. Az iteratív folyamatot létrehozó SML-függvény egy változata:

```
fun fib n = let fun fibIter (i, b, a) =
 if i = n then b
 else fibIter(i+1, a, a+b)
 in
 fibIter(0, 0, 1)
 end
```

- *Mintaillesztést* használhatunk, ha  $i$ -t nem növeljük, hanem  $n$ -től 0-ig csökkentjük.

Figyelem: a klózik sorrendje, mivel nem egymást kizáróak a minták, lényeges!

```
fun fib n = let fun fibIter (0, b, a) = b
 | fibIter (i, b, a) = fibIter(i-1, a, a+b)
 in
 fibIter(n, 0, 1)
 end
```

## Elágazó rekurzió (folyt.)

---

- A Fibonacci-példában a lépések száma elágazó rekurzióval tehát  $n$ -nel exponenciálisan, lineáris rekurzióval  $n$ -nel arányosan nőtt, kis  $n$ -ekre is hatalmas a nyereség!
- Téves lenne azonban azt a következtetést levonni, hogy az elágazó rekurzió használhatatlan. Amikor hierarchikusan strukturált adatokon kell műveleteket végezni, pl. egy fát kell bejárni, akkor az elágazó rekurzió (angolul: *tree recursion*) nagyon is természetes és hasznos eszköz.
- Az elágazó rekurzió numerikus számításoknál az algoritmus első megfogalmazásakor is hasznos lehet: gondoljunk csak arra, hogy milyen könnyű volt átírni a Fibonacci-számok matematikai definícióját programmá.
- Ha már értjük a feladatot, az első, rossz hatékonyságú változatot könnyebb átírni jó, hatékony programmá. Az elágazó rekurzió segíthet a feladat megértésében.

Az iteratív Fibonacci-algoritmushoz csak egy aprócska ötlet kellett. A következő feladatra azonban nem lenne könnyű iteratív algoritmust írni.

- Hányféleképpen lehet felváltani egy dollárt 50, 25, 10, 5 és 1 centesekre?
- Általánosabban: adott összeget adott érmékkel hányféleképpen lehet felváltani?

## Elágazó rekurzió (folyt.): pénzváltás

Tegyük föl, hogy  $n$  darab érme áll a rendelkezésünkre valamilyen (pl. nagyság szerint csökkenő) sorrendben. Ekkor az  $a$  összeg lehetséges felváltásainak számát úgy kapjuk meg, hogy

- kiszámoljuk, hogy az  $a$  összeg hányféleképpen váltható fel az első ( $d$  értékű) érmét kivéve a többi érmével, és ehhez
- hozzáadjuk, hogy az  $a - d$  összeg hányféleképpen váltható fel az összes érmével, az elsőt is beleértve – más szóval azt, hogy az  $a$  összeget hányféleképpen tudjuk úgy felváltani, hogy a  $d$  érmét legalább egyszer felhasználjuk.

A feladat tehát rekurzióval megoldható, hiszen redukálható úgy, hogy kisebb összegeket kevesebb érmével kell felváltanunk. A következő alapeseteket különböztessük meg:

- Ha  $a = 0$ , a felváltások száma 1.  
(Ha az összeg 0, csak egyféleképpen, 0 db érmével lehet „felváltani”.)
- Ha  $a < 0$ , a felváltások száma 0.
- Ha  $n = 0$ , a felváltások száma 0.

A példában a `firstDenomination` (magyarul *első címlet*) függvényt felsorolással valósítottuk meg. Tömörebb és rugalmasabb lenne a megvalósítása lista alkalmazásával.

## Elágazó rekurzió (folyt.): pénzváltás

```
fun countChange amount =
 let (* cC amount kindsOfCoins = az amount összes felváltásainak száma
 kindsOfCoins db érmével *)
 fun cC (amount, kindsOfCoins) =
 if amount < 0 orelse kindsOfCoins = 0 then 0
 else if amount = 0 then 1
 else cC (amount, kindsOfCoins - 1) +
 cC (amount - firstDenomination kindsOfCoins, kindsOfCoins)
 and firstDenomination 1 = 1
 | firstDenomination 2 = 5
 | firstDenomination 3 = 10
 | firstDenomination 4 = 25
 | firstDenomination 5 = 50
 in
 cC(amount, 5)
 end;
```

```
countChange 10 = 4; countChange 100 = 292;
```

*Gyakorló feladatok.*

- Írja át a `firstDenomination` függvényt úgy, hogy a címleteket egy lista tartalmazza.
- Írja meg a `cC` függvény mintaillesztést használó változatát!

## Hatványozás

- Az eddig látott folyamatokban a kiértékelési (végrehajtsági) lépések száma az adatok  $n$  számával lineárisan, ill. exponenciálisan nőtt. Most olyan példa következik, amelyben a lépések száma az  $n$  logaritmusával arányos.

- A  $b$  szám  $n$ -edik hatványának definícióját ugyancsak könnyű átrakni SML-be.

$$b^0 = 1 \quad \left| \begin{array}{l} \text{fun expt (b, 0) = 1} \\ b^n = b \cdot b^{n-1} \quad \left| \begin{array}{l} \text{expt (b, n) = b * expt(b, n-1)} \end{array} \right. \end{array} \right.$$

- A létrejövő folyamat lineáris-rekurzív.  $O(n)$  lépés és  $O(n)$  méretű tár kell a végrehajtásához.
- A faktoriálisszámításhoz hasonlóan könnyű felírni lineáris-iteratív változatát.

```
fun expt (b, n) =
 let fun exptIter (0, product) = product
 | exptIter (counter, product) =
 exptIter (counter-1, b * product)
 in
 exptIter(n, 1)
 end
```

- $O(n)$  lépés és  $O(1)$  – azaz konstans – méretű tár kell a végrehajtásához.

## Hatványozás (folyt.)

- Kevesebb lépés is elég, ha kihasználjuk az alábbi egyenlőségeket:

$$b^0 = 1$$

$$b^n = (b^{n/2})^2, \text{ ha } n \text{ páros}$$

$$b^n = b \cdot b^{n-1}, \text{ ha } n \text{ páratlan}$$

```
fun expt (b, n) =
 let fun exptFast 0 = 1
 | exptFast n =
 if even n
 then square(exptFast(n div 2))
 else b * exptFast(n-1)
 and even i = i mod 2 = 0
 and square x = x * x
 in exptFast n end
```

- A lépések száma és a tár mérete  $O(\lg n)$ -nel arányos. Konstans tárigényű iteratív változata:

```
fun expt (b, 0) = 1 (* Nem hagyható el! Miért nem? *)
 | expt (b, n) = let fun exptFast (1, r) = r
 | exptFast (n, r) =
 if even n then exptFast(n div 2, r*r)
 else exptFast(n-1, r*b)
 and even i = i mod 2 = 0
 in exptFast(n, b) end
```

# LISTÁK

További listakezelő függvények FP-1..12-74

## 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  
 $\text{take}(xs, i) = [x_0, x_1, \dots, x_{i-1}]$  és  $\text{drop}(xs, i) = [x_i, x_{i+1}, \dots, x_{n-1}]$ .
- take egy megvalósítása (jobbrekurzív-e? jobbrekurzívvá tehető-e? robusztus-e?)

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

- drop egy megvalósítása (jobbrekurzív-e? jobbrekurzívvá tehető-e? robusztus-e?)

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

- Könyvtári változatuk – `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

Idézzük föl az egészlista maximális értékét megkereső `maxl` függvény két változatát:

- `maxl` jobbról balra egyszerűsítő (nem jobbrekurzív) változata

```
(* maxl : int list -> int
 maxl ns = az ns egészlista legnagyobb eleme
*)
fun maxl [] = raise Empty
 | maxl [n] = n
 | maxl (n::ns) = Int.max(n, maxl ns)
```

- `maxl` balról jobbra egyszerűsítő (jobbrekurzív) változata:

```
(* maxl' : int list -> int
 maxl' ns = az ns egészlista legnagyobb eleme
*)
fun maxl' [] = raise Empty
 | maxl' [n] = n
 | maxl' (n::m::ns) = maxl' (Int.max(n,m)::ns)
```

- Amint ez a példa is mutatja, vissza-visszatérő feladat egy lista redukciója kétoperandusú művelettel.
- Közös bennük, hogy  $n$  db értékből egyetlen értéket kell előállítani, ezért is beszélünk *redukcióról*.

## Lista redukciója kétoperandusú művelettel (`foldr`, `foldl`)

- `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

```
foldr op \oplus e [x1, x2, ..., xn] = (x1 \oplus (x2 \oplus ... \oplus (xn \oplus e) ...))
foldr op \oplus e [] = e
foldl op \oplus e [x1, x2, ..., xn] = (xn \oplus ... \oplus (x2 \oplus (x1 \oplus e)) ...)
foldl op \oplus e [] = e
```

- A  $\oplus$  művelet  $e$  operandusa néhány gyakori műveletben – összeadás, szorzás, konjunkció (logikai „és”), alternáció (logikai „vagy”) – a (jobb oldali) *egységelem* szerepét tölti be.
- Asszociatív műveleteknél `foldr` és `foldl` eredménye azonos.

## Példák foldr és foldl alkalmazására

- `isum` egy egészlista elemeinek összegét, `rprod` egy valóslista elemeinek szorzatát adja eredményül.

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

- A `length` függvény is definiálható `foldl`-lel vagy `foldr`-rel. Kétooperandusú műveletként olyan segédfüggvényt (`inc`) alkalmazunk, amelyik *nem használja* az első paraméterét.

```
(* 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;
val lengthl = foldl inc 0;
```

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

## Lista: foldr és foldl definíciója

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

```
(* 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;
```

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

```
(* 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)
```

## További példák `foldr` és `foldl` alkalmazására

---

- `maxl` két megvalósítása

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

```
(* nem jobbrekurzív *)
```

```
fun maxl max [] = raise Empty
 | maxl max (n::ns) = foldr max n ns
```

```
(* jobbrekurzív *)
```

```
fun maxl' max [] = raise Empty
 | maxl' max (n::ns) = foldl max n ns
```



## Példa listák használatára: futamok előállítása

- A futam egy olyan lista, amelynek az elemei egy adott feltételnek megfelelnek.
- Az adott feltételt az előző és az aktuális elemre alkalmazandó *predikátumként* adjuk át a futamot előállító függvénynek.
- A feladat: írjunk olyan SML függvényt, amely (az elemek eredeti sorrendjének megőrzésével) egy lista egymás utáni elemeiből képzett futamok listáját adja eredményül.
- Az első változatban egy-egy segédfüggvényt írunk egy lista első (prefix) futamának, valamint a maradéklistának az előállítására.
- A futam segédfüggvénynek két argumentuma van: az első egy predikátum, amely a kívánt feltételt megvalósítja, a második pedig egy pár. A pár első tagja az előző elem, a második tagja pedig az a lista, amelynek az előző elemmel induló futamát kell futam-nak előállítania.
- A maradék segédfüggvény két argumentuma azonos futam argumentumaival. Eredményül azt a listát kell visszaadnia, amelyet az első futam leválasztásával állít elő a pár második tagjaként átadott listából.
- A következő diákon a futam és a maradék segédfüggvények, valamint a futamok függvény különböző változatai láthatók.

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

- Első változat: futam és maradék előállítása két függvénnyel

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

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

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

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

---

### • Hatékonyságot rontó tényezők

1. futamok1 kétszer megy végig a listán: először futam, azután maradék,
2. p-t, bár sohasem változik, paraméterként adjuk át futam-nak és maradék-nak,
3. egyik függvény sem használ akkumulátort.

### • Javítási lehetőségek

1. futam egy párt adjon eredményül, ennek első tagja legyen a futam, második tagja pedig a maradék; a futam elemeinek gyűjtésére használjunk akkumulátort,
2. futam legyen lokális futamok2-n belül,
3. az `if null ms then [fs] else` szövegrész törölhető: a rekurzió egy hívással később mindenképpen leáll.

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

---

### • Második változat: futam és maradék előállítása egy lokális függvénnyel

```
(* 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 (* futam : ('a * 'a list) -> 'a list * 'a list
 futam (x, ys) zs = olyan pár, amelynek első tagja az x::ys p-t
 kielégítő első (prefix) futama 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
 fs :: futamok2 p ms
 end
```

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

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

```
(* 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 (* futamok : ('a * 'a list) -> 'a list -> 'a list * 'a list
 futamok (x, ys) zs zss = az x::ys p-t kielégítő futamaiból
 álló lista zss elé fűzve
 *)
 fun futamok (x, []) zs zss = rev(rev(x::zs)::zss)
 | futamok (x, yys as y::ys) zs 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;
```

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

- Példák a függvények alkalmazására:

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

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

```
futamok1 op<= [99,1] = [[99], [1]];
```

```
futamok1 op<= [99] = [[99]];
```

```
futamok1 op<= [] = [];
```

A 7. előadáson, 2004. november 2-án az október 28-ai nagyzárthelyi feladatainak megoldását beszéltük meg.

A javítási útmutató a tárgy honlapján <<http://dp.iit.bme.hu/dp04a>> megtalálható.

# ABSZTRAKCIÓ FÜGGVÉNYEKKEL (ELJÁRÁSOKKAL)

Absztrakció függvényekkel (eljárásokkal) FP-1..12-88

## Legnagyobb közös osztó

- Következő példánk  $a$  és  $b$  legnagyobb közös osztóját számolja ki az euklideszi algoritmussal.
- Az alapgondolat az, hogy ha  $a$ -t  $b$ -vel osztva  $r$  a maradék, akkor  $a$  és  $b$  közös osztói azonosak  $b$  és  $r$  közös osztóival.
- A matematikai definíciót most is pontosan követi az SML-függvény.

$$\begin{array}{l|l} \text{gcd}(a, 0) = a & \text{fun gcd (a, 0) = a} \\ \text{gcd}(a, b) = \text{gcd}(b, a \bmod b) & \quad | \quad \text{gcd (a, b) = gcd(b, a mod b)} \end{array}$$

- A *folyamat* iteratív. A lépések száma logaritmikusan nő.  
Pontosabban – a *Lamé-tétel* szerint – ha az euklideszi algoritmus egy számpár legnagyobb közös osztóját  $k$  lépésben számítja ki, akkor a számpár kisebbik tagja nem lehet kisebb a  $k$ -adik Fibonacci-számnál. (Ld. SICP, 1.2.5. szakasz.)  
Legyen  $n$  az algoritmus kisebbik paramétere. Ha a legnagyobb közös osztó kiszámításához  $k$  lépésre van szükség, akkor  $n \geq F(k) \approx \Phi^k / \sqrt{5}$ . Azaz a  $k$  lépésszám valóban az  $n$  ( $\Phi$  alapú) logaritmusával arányos.

## Prímteszt

---

- A `prime` predikátum egy  $n$  szám prím voltát teszteli. A `findDivisor` függvény 2-től kezdve megkeresi az  $n$  szám legkisebb osztóját. Az  $n$  szám prím, ha a legkisebb osztó az  $n$  szám maga.
- Az  $n$  osztóit 2-től  $\sqrt{n}$ -ig kell keresni, így a lépések száma  $O(\sqrt{n})$ .

```

fun prime n =
 let
 infix divides
 fun smallestDivisor n = findDivisor(n, 2)
 and findDivisor (n, testDivisor) =
 if square testDivisor > n
 then n
 else if testDivisor divides n
 then testDivisor
 else findDivisor(n, testDivisor+1)
 and square x = x * x
 and a divides b = b mod a = 0
 in
 n = smallestDivisor n
 end

```

### Gyakorló feladat.

`prime` egyesével lépkedve keresi meg az  $n$  legkisebb osztóját. Írjon gyorsabb megoldást!

## Prímteszt (folyt.)

---

- A következő SML-predikátum egy szám prím voltát *valószínűségi módszerrel* teszteli. A lépések száma  $O(\lg n)$ .
- Az algoritmus a kis Fermat-tételre alapul, amely azt mondja ki, hogy: ha  $n$  prím és  $0 < a < n$ , akkor  $a^n$  modulo  $n$  szerint *kongruens*  $a$ -val, azaz  $a^n \bmod n = a$ .
  - Két szám akkor *kongruens* modulo  $n$  szerint, ha  $n$ -nel osztva mindkettőnek ugyanaz a maradéka. Egy  $a$  szám  $n$ -nel való osztásának maradékát  $a$  modulo  $n$  szerinti maradékának, vagy röviden  $a$  modulo  $n$ -nek is nevezik.
- Ha  $n$  nem prím, akkor az  $a < n$  számok nagy hányadára nem teljesül a fenti reláció.
- A prímteszt algoritmus a ezek után a következő :
  - Adott  $n$ -re véletlenszerűen válasszunk egy  $a < n$  számot: ha  $a^n \bmod n \neq a$ , akkor  $n$  nem prím. Ellenkező esetben nagy a valószínűsége, hogy  $n$  prím.
  - Válasszunk véletlenszerűen egy másik  $a < n$  számot: ha  $a^n \bmod n = a$ , akkor növekedett annak a valószínűsége, hogy az  $n$  prím. Újabb és újabb  $a$  értékeket választva egyre biztosabbak lehetünk abban, hogy az  $n$  prím.

## Prímteszt (folyt.)

---

- Az `expmod` segédfüggvény a `base` szám `exp`-edik hatványának modulo `m` szerinti maradékát adja eredményül.

```
(* expmod (base, exp, m) = base exp-edik hatványa modulo m
*)
fun expmod (_, 0, _) = 1
 | expmod (b, e, m) =
 if even e
 then square(expmod(b, e div 2, m)) mod m
 else b * expmod(b, e-1, m) mod m
and even n = n mod 2 = 0
and square x = x * x;
```

- Nagyon hasonló felépítésű `exptFast`-hoz. A lépések száma a kitevő logaritmusával arányos.
- Szükségünk van véletlenszámok előállítására. Részletek az SML alapkönyvtárából:

```
Random.range (min, max) gen = an integral random number in the
 range [min, max). Raises Fail if min > max.
Random.newgen () = a random number generator, taking the seed
 from the system clock.
```

## Prímteszt (folyt.)

---

- Betöltjük a `Random` könyvtárat:

```
load "Random";
```

- `fermatTest` generál egy álvéletlen-számot, és egyszer elvégzi a vizsgálatot:

```
(* fermatTest n = false if n is not prime *)
fun fermatTest n =
 let fun tryIt a = expmod(a, n, n) = a
 in tryIt(Random.range (1, n) (Random.newgen()))
 end
```

- `fastPrime times`-szor megismétli a vizsgálatot:

```
(* fastPrime (n, times) = true if n passes the prime test
 times times
*)
fun fastPrime (n, 0) = true
 | fastPrime (n, t) = fermatTest n andalso fastPrime(n, t-1)
```

- Ez a megoldás csak nagy valószínűséggel, de nem teljes bizonyossággal ad választ a kérdésre. Például az 561 átmegy a Fermat-teszten, bár nem prím.

## Függvények mint általános számítási módszerek

---

- Láttuk, hogy a függvény (ill. általában az eljárás) olyan *absztrakció*, amely – a paraméterként átadott adatok konkrét értékétől függetlenül – összetett műveleteket ír le.
- Az olyan magasabbrendű függvény, amelynek függvény a paramétere, még *magasabb szintű* absztrakció, hiszen az általa megvalósított összetett műveletet nemcsak egyes konkrét adatoktól, hanem egyes konkrét műveletektől is függetlenné tesszük.
- A magasabbrendű függvény (eljárás) tehát valamilyen *általános számítási módszert* fejez ki.
- A következő lapokon két nagyobb példát ismertetünk: általános számítási módszert függvények *zérushelyeinek* és *fixpontjának* a megtalálására.

## Egyenlet gyökeinek meghatározása intervallumfelezéssel

---

- Az intervallumfelezés módszere hatékony eljárás az  $f(x) = 0$  egyenlet gyökeinek megtalálására, ahol  $f$  folytonos függvény.
- A közismert alapötlet a következő:
  - Megfelelően megválasztott  $a$ -ra és  $b$ -re, amelyekre  $f(a) < 0 < f(b)$ ,  $f$ -nek legalább egy zérushelye van  $a$  és  $b$  között.
  - A zérushely megtalálásához legyen  $x = a + b/2$ . Ha  $f(x) > 0$ , akkor  $f$  zérushelyét  $a$  és  $x$  között, ha  $f(x) < 0$ , akkor  $x$  és  $b$  között kell keresnünk.
  - A keresést – a rekurziót – akkor hagyjuk abba, amikor két egymás utáni közelítő érték *eltérése* egy előre meghatározott értéknél kisebb lesz.
- Mivel az eltérés minden lépésben a felére csökken, az  $f$  zérushelyének megtalálásához szükséges lépések száma  $O(L/T)$ , ahol  $L$  az intervallum hossza kezdetben, és  $T$  a megengedett eltérés.
- A leírt algoritmust valósítja meg a `search` függvény (ld. a következő lapon):
 

```
(* search (f, negPoint, posPoint) = root of f x in the
 negPoint <= x <= posPoint interval
 PRE: f negPoint <= 0 and f posPoint >= 0
 *)
```

## Egyenlet gyökeinek meghatározása intervallumfelezéssel (folyt.)

---

```

fun search (f, negPoint, posPoint) =
 let val midPoint = average(negPoint, posPoint)
 in
 if closeEnough(negPoint, posPoint)
 then midPoint
 else let val testValue = f midPoint
 in
 if positive(testValue)
 then search(f, negPoint, midPoint)
 else if negative(testValue)
 then search(f, midPoint, posPoint)
 else midPoint
 end
 end
 end

and average (x, y) = (x+y)/2.0
and closeEnough (x, y) = abs(x-y) < 0.001
and positive x = x >= 0.0
and negative x = x < 0.0

```

## Egyenlet gyökeinek meghatározása intervallumfelezéssel (folyt.)

---

- Az előfeltételek betartását célszerű search alkalmazásakor ellenőrizni, nehogy rossz választ kapjunk az SML értelmezőtől.

```

● - search(Math.sin, 4.0, 2.0) (* Helyes az eredménye *)
 > val it = 3.14111328125 : real

```

```

● - search(Math.sin, 2.0, 4.0) (* Hibás az eredménye *)
 > val it = 2.00048828125 : real

```

- A halfIntervalMethod függvény elvégzi az ellenőrzést, és jelzi, ha negPoint vagy posPoint kezdeti értéke nem jó.

```

(* halfIntervalMethod (f, a, b) = root of f x in the
 a <= x <= b interval
*)

```

- Figyeljük meg az *ügyek szétválasztása* elv alkalmazását: search a gyökkeresési stratégiát valósít meg, halfIntervalMethod pedig az előfeltételeket meglétét ellenőrzi.



## Egyenlet gyökeinek meghatározása intervallumfelezéssel (folyt.)

---

```

• fun halfIntervalMethod(f, a, b) =
 let val aValue = f a
 val bValue = f b
 in
 if negative aValue andalso positive bValue
 then search(f, a, b)
 else if negative bValue andalso positive aValue
 then search(f, b, a)
 else print ("Values " ^ makestring a ^ " and " ^
 makestring b ^ " are not of opposite sign.\n")
 end

```

- A `makestring` függvény (típusa `numtxt -> string`) tetszőleges numerikus (`int`, `real`, `word`, `word8`), `char` és `string` típusú értéket `string` típusvá alakít.
- A függvénynek ez a változata hibás, mert az `if-then-else` feltételes kifejezés összes ágának *ugyanolyan típusú* eredményt *kell* adnia, márpedig `print` eredménye nem `int` típusú.
- Megoldás az  $(e; f)$  alakú ún. *szekvenciális kifejezés* használata: az értelmező kiértékeli  $e$ -t és  $f$ -et a felírt sorrendben, eredményül pedig  $f$  értékét adja.

## Egyenlet gyökeinek meghatározása intervallumfelezéssel (folyt.)

---

```

fun halfIntervalMethod(f, a, b) =
 let val (aValue, bValue) = (f a, f b)
 in
 if negative aValue andalso positive bValue
 then search(f, a, b)
 else if negative bValue andalso positive aValue
 then search(f, b, a)
 else (print ("Values " ^ makestring a ^ " and " ^
 makestring b ^ " are not of opposite sign.\n");
 0.0)
 end;

```

```

- halfIntervalMethod(Math.sin, 2.0, 4.0);
> val it = 3.14111328125 : real
- halfIntervalMethod(fn x => x*x*x-2.0*x-3.0, 1.0, 2.0);
> val it = 1.89306640625 : real
- halfIntervalMethod(Math.sin, 2.0, 2.5);
Values 2.0 and 2.5 are not of opposite signs
> val it = 0.0 : real

```

## Függvény fixpontjának meghatározása

---

- Az  $f(x) = x$  egyenletet kielégítő  $x$  az  $f$  függvény *fixpontja*.
- Egy  $f$  függvény valamely fixpontját megfelelő kezdőértékből kiindulva  $f$  rekurzív alkalmazásával határozhatjuk meg:  
 $f x, f(f x), f(f(f x)), f(f(f(f x))), \dots$   
 A rekurzió akkor fejezhető be, amikor már elhanyagolható mértékű a változás.
- A `fixedPoint` függvény paramétere egy pár; ennek első tagja egy függvény, amelynek a fixpontját keressük, a második tagja pedig a fixpont egy első közelítése.

```
(* fixedPoint (f, firstGuess) = fixpoint of f in the proximity
 of firstGuess with tolerance tolerance
*)
```

- Szükségünk van még a közelítés megkívánt pontosságára:

```
val tolerance = 0.00001;
```

## Függvény fixpontjának meghatározása (folyt.)

---

```
fun fixedPoint (f, firstGuess) =
 let
 fun closeEnough (v1, v2) = abs(v1-v2) < tolerance
 fun try guess =
 let
 val next = f guess
 in
 if closeEnough(guess, next)
 then next
 else try next
 end
 in
 try firstGuess
 end;

load "Math";
fixedPoint(Math.cos, 1.0);
fixedPoint(fn y => Math.sin y + Math.cos y, 1.0);
```

## Függvény fixpontjának meghatározása (folyt.)

- A fixpontszámítás hasonlít a négyzetgyökvonás korábban megbeszélte folyamatára: mindkettő azon alapul, hogy addig finomítjuk a közelítést, amíg valamilyen feltétel nem teljesül.
- A négyzetgyökvonás könnyedén megfogalmazható fixpontszámításként: ha  $x$  négyzetgyöke  $y$ , akkor  $y * y = x$ , azaz  $y = x/y$ . Az  $f y = x/y$  függvény fixpontja tehát az  $x$  négyzetgyöke.  

```
fun sqrt x = fixedPoint (fn y => x/y, 1.0);
```
- A megoldásunk rossz, ugyanis nem konvergál! Könnyen belátható:  
 Legyen  $x$  négyzetgyökének első közelítése  $y_1$ , a második  $y_2 = x/y_1$ , a harmadik  $y_3 = x/y_2 = x/(x/y_1) = y_1$ . Látható, hogy a folyamat sohasem ér véget.
- Az oszcillációt pl. úgy gátolhatjuk meg, hogy *korlátozzuk* két közelítő érték között a változás mértékét.
- Mivel a helyes válasz mindig az  $y$  közelítő érték és  $x/y$  között van,  $y$ -hoz  $x/y$ -nál *közelebb eső* új közelítő értéként  $y$  és  $x/y$  átlagát választhatjuk:  $y \leftarrow (y + x/y)/2$ .  

```
fun sqrt x = fixedPoint (fn y => (y+x/y)/2.0, 1.0);
```
- Ezt a gyakran használható módszert *átlagsillapításnak* (angolul *average damping*) nevezik.

## Függvény mint visszatérési érték

- A függvényekről mint absztrakciós eszközökről szólva eddig olyan magasabbrendű függvényeket használtunk, amelyeknek más függvények voltak a paraméterei.
- Most olyan magasabbrendű függvényeket mutatunk be, amelyek *függvényt* (pontosabban *függvényértéket*) adnak eredményül.
- A korábban bemutatott *átlagsillapítás* sokszor használható módszer, ezért érdemes önálló függvényként megírni: ha adott az  $f$  függvény, elő kell állítani  $x$  és  $f x$  átlagát.  

```
(* averageDamp f = f valamely x értékre alkalmazva
 előállítja x és f x átlagát *)
fun averageDamp f = fn x => (x + f x) / 2.0;
```
- Jól látható, hogy `averageDamp`, ha csak egyetlen paraméterre alkalmazzuk, függvényértéket ad eredményül. `averageDamp` részlegesen alkalmazható függvény.
- Példa `averageDamp` alkalmazására:  

```
(averageDamp (fn x => x*x)) 10.0; (* 10.0 és 100.0 átlaga *)
```
- A kiértékelés sorrendje miatt a külső zárójelpár el is hagyható:  

```
averageDamp (fn x => x*x) 10.0;
```

## Függvény mint visszatérési érték (folyt.)

---

- averageDamp definíciója felírható (szintaktikai édesítőszerszerrel).

```
fun averageDamp f x = (x + f x) / 2.0;
```

- sqrt averageDamp-pel felírt változata explicitté teszi a *fixpontmeghatározás* és az *átlagcsillapítás* módszerét, továbbá az  $y = x/y$  egyenlet használatát.

```
fun sqrt x = fixedPoint(averageDamp (fn y => x/y), 1.0);
sqrt 4.0;
```

- Tanulság: egy folyamatot sokféle eljárással leírhatunk, de a *lényeg* sokkal könnyebb megérteni, ha *megfelelően megválasztott absztrakciókat* vezetünk be.

- Még egy példa a bemutatottak alkalmazására: az  $x$  köbgyöke az  $y \mapsto x/y^2$  – SML-jelöléssel az  $\text{fn } y \Rightarrow x / (y * y)$  – függvény fixpontja. A megoldás már kész is van!

```
fun cubeRoot x = fixedPoint(averageDamp (fn y => x/y/y), 1.0);
cubeRoot 8.0;
```

## Függvény mint visszatérési érték (folyt.): az általános Newton-módszer

---

- Ha  $x \mapsto g(x)$  egy differenciálható függvény, akkor a  $g(x) = 0$  egyenlet az  $x \mapsto f(x)$  függvény egy fixpontja, ahol  $f(x) = x - g(x)/g'(x)$  és  $g'(x)$  a  $g$   $x$  szerinti deriváltja.
- Az *általános Newton-módszer* a fixpontmódszer egy alkalmazása az  $f$  függvény egy fixpontjának megtalálására. Számos  $g$  függvényre és megfelelően megválasztott  $x$  értékre a Newton-módszer gyorsan konvergál.
- Először is azt a *deriv* függvényt kell definiálnunk, amelynek (az *averageDamp* függvényhez hasonlóan) függvény a paramétere, és függvényt ad eredményül.
- Ha  $g$  függvény és  $dx$  egy kis szám, akkor a  $g$  függvény  $g'$  deriváltja az a függvény, amelynek értéke bármely  $x$  számra a következő:  $g'(x) = (g(x + dx) - g(x))/dx$ .

```
(* deriv g = g deriváltja
*)
val dx = 0.00001;
fun deriv g = fn x => (g(x+dx) - g x) / dx;
```

- Például az  $x \mapsto x^3$  függvény deriváltja  $x = 5$ -re (pontos értéke 75):

```
let fun cube x = x*x*x in deriv cube 5.0 end;
```

## Függvény mint visszatérési érték (folyt.): a Newton-módszer fixpont-folyamatként

- `deriv` felhasználásával az általános Newton-módszer definiálható *fixpont-folyamatként*:

```
fun newtonTransform g x = x - (g x / deriv g x)
and newtonsMethod g guess = fixedPoint(newtonTransform g, guess)
```

- Példa `newtonsMethod` használatára:

```
fun sqrt x = newtonsMethod (fn y => y*y-x) 1.0;
sqrt 16.0;
```

- Két általános módszer egy-egy alkalmazását láttuk egy szám négyzetgyökének kiszámítására: az egyik a fixpont-, a másik a Newton-módszer.
- Mivel az utóbbi is a fixpontmódszeren alapul, valójában a fixpontmódszer kétféle alkalmazását láttuk.
- Mindkét esetben egy függvényből indulunk ki, és kiszámítjuk valamely transzformáltjának egy fixpontját.
- Ezt az általános módszert is definiálhatjuk eljárásként (függvényként), ezt mutatjuk be a következő diákon.

## Függvény mint visszatérési érték (folyt.): a fixpontmódszer kétféle alkalmazása

- (\* `fixedPointOfTransform (g, transform, guess) =`  
a fixed point of (transform g) with the initial guess guess  
\*)  

```
fun fixedPointOfTransform (g, transform, guess) =
 fixedPoint(transform g, guess)
```

- Ez volt `sqrt` fixpontkeresésén alapuló első változata:

```
fun sqrt x = fixedPoint(averageDamp (fn y => x/y), 1.0)
```

- Átírva az általános módszert megvalósító függvénnyel:

```
fun sqrt x = fixedPointOfTransform (fn y => x/y,
 averageDamp, 1.0)
```

- Ez volt `sqrt` Newton általános módszerét használó második változata:

```
fun sqrt x = newtonsMethod (fn y => y*y-x) 1.0;
```

- Átírva az általános módszert megvalósító függvénnyel:

```
fun sqrt x = fixedPointOfTransform (fn y => y*y-x,
 newtonTransform, 1.0)
```

# ABSZTRAKCIÓ ADATOKKAL

Absztrakció adatokkal FP-1..12-108

## Adatabsztrakció: racionális számok

- A következő előadásokon összetett adatokkal és adatabsztrakcióval foglalkozunk.
- Az adatabsztrakció lényege: összetett adatokkal dolgozó programjainkat úgy építjük föl, hogy
  - az adatokat felhasználó programrészek az adatok szerkezetéről ne tétélezzenek fel semmit, csak az előre definiált műveleteket használják,
  - az adatokat definiáló programrészek az adatokat felhasználó programrészektől függetlenek legyenek.
  - A program e két része közötti interfész *konstruktorokból* és *szelektorokból* áll.
- Az összetett adatok közül eddig ennesekkel és listákkal találkoztunk.
- Első nagyobb példánkban a racionális számok és a rajtuk végezhető műveletek megvalósítását mutatjuk be.
- A racionális számot ábrázolhatjuk egy olyan párral, amelynek az első tagja a számláló (*numerator*) és a második a nevező (*denominator*).
- Megvalósítjuk a négy aritmetikai alpműveletet: `addRat`, `subRat`, `mulRat`, `divRat`, továbbá az egyenlőségvizsgálatot: `equRat`.

## Adatabsztrakció: racionális számok (folyt.)

- Tegyük föl, hogy
  - van olyan *konstruktorműveletünk*, amely egy  $n$  számlálóból és egy  $d$  nevezőből létrehozza a racionális számot: `makeRat (n, d)`, továbbá
  - van egy-egy olyan *szelektorműveletünk*, amelyek egy  $q$  racionális szám számlálóját, ill. nevezőjét előállítják: `num q`, `den q`.
- A jól ismert műveleteket írjuk át SML-programmá:

$$n_1/d_1 + n_2/d_2 = (n_1d_2 + n_2d_1)/(d_1d_2), \quad n_1/d_1 - n_2/d_2 = (n_1d_2 - n_2d_1)/(d_1d_2),$$

$$(n_1/d_1)(n_2/d_2) = (n_1n_2)/(d_1d_2), \quad (n_1/d_1)/(n_2/d_2) = (n_1d_2)/(d_1n_2),$$

$$n_1/d_1 = n_2/d_2 \text{ akkor és csak akkor, ha } n_1d_2 = n_2d_1.$$

```

fun addRat(x, y) =
 makeRat(num x * den y + num y * den x, den x * den y)
fun subRat(x, y) =
 makeRat(num x * den y - num y * den x, den x * den y)
fun mulRat(x, y) = makeRat(num x * num y, den x * den y)
fun divRat(x, y) = makeRat(num x * den y, den x * num y)
fun equRat(x, y) = num x * den y = den x * num y

```

## Adatabsztrakció: racionális számok (folyt.)

- Az SML-ben az *ennes* létrehozására van *konstruktorműveletünk*: a tagokat kerek zárójelek között, vesszővel elválasztva felsoroljuk, és
- van az *ennes* egy-egy tagját kiválasztó *szelektorműveletünk*: `# i`, ahol  $i$  az  $i$ -edik tag *pozicionális címkéje*, 1-től kezdve.
- Példák: `(3, 4)`; `#1(3, 4)`; `#2(3, 4)`;
- Az *ennes* tagjai *mintaillesztéssel* is köthetők névhez, pl. `val (n, d) = (3, 4)`.
- *Gyenge absztrakcióval* valósítjuk meg a racionális szám típusát, a konstruktort és szelektorokat:

```

type rat = int * int;
fun makeRat (n, d) = (n, d) : rat;
fun num (q : rat) = #1 q;
fun den q = #2 (q : rat);

```

- A *gyenge absztrakció* nevet ad egy objektumnak, de *nem rejt* el a megvalósítás részleteit.
- Szükségünk lesz kiíróműveletre is az  $n/d$  alakú racionális szám kiírásához.

```

fun printRat q =
 print(makestring(num q) ^ "/" ^ makestring(den q) ^ "\n");

```

## Adatabsztrakció: racionális számok (folyt.)

---

- Ezzel racionális számokat megvalósító programunk első változata kész is van. A teljes program:

```

type rat = int * int;
fun makeRat (n, d) = (n, d) : rat;
fun num (q : rat) = #1 q;
fun den q = #2 (q : rat);

fun addRat(x, y) =
 makeRat(num x * den y + num y * den x, den x * den y)
fun subRat(x, y) =
 makeRat(num x * den y - num y * den x, den x * den y)
fun mulRat(x, y) = makeRat(num x * num y, den x * den y)
fun divRat(x, y) = makeRat(num x * den y, den x * num y)
fun equRat(x, y) = num x * den y = den x * num y

fun printRat q =
 print(makestring(num q) ^ "/" ^ makestring(den q) ^ "\n");

```

## Adatabsztrakció: racionális számok (folyt.)

---

- Néhány példa a program használatára:

```

val oneHalf = makeRat(1,2);
val oneThird = makeRat(1,3);
val twoThird = makeRat(2,3);

printRat oneHalf;
printRat(addRat(oneHalf, oneThird));
printRat(mulRat(oneHalf, oneThird));
printRat(addRat(oneThird, oneThird));

equRat(addRat(oneThird, oneThird), twoThird);

oneThird = oneThird;
addRat(oneThird, oneThird) = twoThird;

```



## Adatabsztrakció: racionális számok (folyt.)

---

- Az utolsó példából, ha kipróbáljuk, láthatjuk, hogy programunk nem *normalizálja*, azaz nem a lehető legegyszerűbb alakban tárolja, ill. írja ki a racionális számokat.
- Segíthetünk a dolgon, ha a konstruktorműveletben a számlálót és a nevezőt a legnagyobb közös osztójukkal elosztjuk:

```
fun makeRat (n, d) =
 let val g = gcd(n, d) in (n div g, d div g) : rat end;
```

A szelektorműveleteken nem változtatunk.

- A racionális számokat normalizált alakjukban tároljuk, ezért nemcsak a kiírás, hanem az egyenlőségvizsgálat is helyes eredményt ad:

```
printRat(addRat(oneThird, oneThird));
addRat(oneThird, oneThird) = twoThird;
```

- A normalizáláshoz csak egyetlen helyen kellett változtatni a programon!

## Adatabsztrakció: racionális számok (folyt.)

---

*Adatabsztrakciós korlátok a racionális számok csomagban*

```

Racionális számot használó programok

Racionális szám a feladattérben

addRat subRat mulRat divRat equRat

Racionális szám mint számláló és nevező

konstruktor: makeRat; szelektorok: num, den

Racionális szám mint pár

konstruktor: (,) ; szelektorok: #1, #2

A pár megvalósítása SML-ben
```

## Adatabsztrakció: racionális számok (folyt.)

---

- Az absztrakciós korlátok elszigetelik egymástól a program egyes részeit.
- Előnye, hogy a programokat egyszerűbb karbantartani és módosítani, pl. az adatok ábrázolását megváltoztatni.
- Pl. a racionális szám normalizálható a létrehozása helyett akkor, amikor a számlálójára vagy a nevezőjére van szükségünk. Ha gyakran hozunk létre racionális számokat, de csak ritkán használjuk a számlálóját vagy a nevezőjét, akkor az utóbbi megoldás a hatékonyabb.

```
fun num (q : rat) =
 let val (n, d) = q; val g = gcd(n, d) in n div g end;
fun den (q : rat) =
 let val (n, d) = q; val g = gcd(n, d) in d div g end;
```

- A makeRat függvény nem normalizáló változatát használjuk; a program többi része nem változik.

```
printRat(addRat(oneThird, oneThird));
addRat(oneThird, oneThird) = twoThird;
equRat(addRat(oneThird, oneThird), twoThird) = true;
```

## Adatabsztrakció: racionális számok (folyt.)

---

- *Adatokról* szólva nem elég annyit mondanunk, hogy „adat az, amit az adott konstruktorok és szelektorok megvalósítanak”.
- Nyilvánvaló, hogy konstruktorok és szelektorok csak bizonyos halmazra alkalmas pl. a racionális számok megvalósítására.
- Racionális számok esetén a konstruktoroknak és a szelektoroknak garantálniuk kell az alábbi feltételek (axiómák) teljesülését:

```
(* PRE : d > 0 *)
x = makeRat(n, d);
n = num x
d = den x
```

- Eggyel alacsonyabb absztrakciós szinten a pár-ábrázolásnak is ki kell elégítenie a következő feltételeket:

```
q = (x, y)
x = #1 q
y = #2 q
```

## Adatabsztrakció: racionális számok (folyt.)

---

- Bármely megvalósítás, amely ezeket a feltételeket kielégíti, megfelel, például a következő is:

```
exception Cons of string;
fun cons (x, y) =
 let fun dispatch 0 = x
 | dispatch 1 = y
 | dispatch _ = raise Cons "argument not 0 or 1"
 in dispatch
 end;
fun fst z = z 0;
fun snd z = z 1;
```

- A tulajdonságleíró egyenletek

```
q = cons(n, d)
n = fst q
d = snd q
```

- Vegyük észre, hogy a racionális számot megvalósító cons objektum: *függvény!* fst és snd *üzenetet küld* az objektumnak. Ennek a programozási stílusnak ezért *üzenetküldés* a neve.

## Adatabsztrakció: racionális számok (folyt.)

---

- Példa:

```
val q = cons(1, 2);
fst q = 1; snd q = 2;
```

- A konstruktor és a szelektorok megvalósítása üzenetküldéssel:

```
fun makeRat (n, d) =
 let val g = gcd(n, d) in cons(n div g, d div g) end;
fun num q = fst q;
fun den q = snd q;
```

- Racionális számokat megvalósító csomagunk nagy hibája, hogy *gyenge absztrakciót* valósít meg, azaz nem rejti el a megvalósítás részleteit; a programozóra bízta, hogy az absztrakciós korlátokat milyen mértékben tartja be. Ez hibák forrása.
- A megvalósítás részleteit *erős absztrakcióval*, modulok segítségével rejthetjük el a külvilág elől. Az „implementációs” modul neve az SML-ben: structure, az (opcionális) „interfészmodul” neve pedig: signature.

```
structure name = struct ... end
signature name = sig ... end
```

## Adatabsztrakció modulokkal: racionális számok

---

```

structure Gcd = struct
 fun gcd (a, 0) = a
 | gcd (a, b) = gcd(b, a mod b) end

structure Rat =
struct
 type rat = int * int;
 fun makeRat (n, d) = let val g = Gcd.gcd(n, d) in (n div g, d div g) : rat end
 fun num (q : rat) = #1 q
 fun den q = #2 (q : rat)
 fun addRat(x, y) = makeRat(num x * den y + num y * den x, den x * den y)
 fun subRat(x, y) = makeRat(num x * den y - num y * den x, den x * den y)
 fun mulRat(x, y) = makeRat(num x * num y, den x * den y)
 fun divRat(x, y) = makeRat(num x * den y, den x * num y)
 fun equRat(x, y) = num x * den y = den x * num y
 fun printRat q = print(makestring(num q) ^ "/" ^ makestring(den q) ^ "\n");
 val one = makeRat(1,1)
 val zero = makeRat(0,1)
 val oneHalf = makeRat(1,2)
 val oneThird = makeRat(1,3)
 val twoThird = makeRat(2,3)
end;

```

Az absztrakció még nem elég erős: a részletek nincsenek eléggé elrejtve!

## Adatabsztrakció modulokkal: racionális számok (folyt.)

---

Ez a megvalósított Rat struktúra tényleges szignatúrája:

```

> structure Rat :
 {type rat = int * int,
 val addRat : (int * int) * (int * int) -> int * int,
 val den : int * int -> int,
 val divRat : (int * int) * (int * int) -> int * int,
 val equRat : (int * int) * (int * int) -> bool,
 val makeRat : int * int -> int * int,
 val mulRat : (int * int) * (int * int) -> int * int,
 val num : int * int -> int,
 val one : int * int,
 val oneHalf : int * int,
 val oneThird : int * int,
 val printRat : int * int -> unit,
 val subRat : (int * int) * (int * int) -> int * int,
 val twoThird : int * int,
 val zero : int * int}

```

Kilátszik a rat típus két komponensének int típusa.

## Adatabsztrakció modulokkal: racionális számok (folyt.)

---

A szignatúra létrehozása és a struktúrához kötése *korlátozza* a megvalósított értékek láthatóságát:

```
signature Rat =
sig
 type rat
 val makeRat : int * int -> rat
 val num : rat -> int
 val den : rat -> int
 val addRat : rat * rat -> rat
 val subRat : rat * rat -> rat
 val mulRat : rat * rat -> rat
 val divRat : rat * rat -> rat
 val equRat : rat * rat -> bool
 val printRat : rat -> unit
 val one : rat
 val oneHalf : rat
 val oneThird : rat
 val twoThird : rat
 val zero : rat
end;
```

## Adatabsztrakció modulokkal: racionális számok (folyt.)

---

```
structure Rat_1 :> Rat = (* ez ún. áttetsző szignatúrakötés *)
struct
 type rat = int * int;
 fun makeRat (n, d) = let val g = Gcd.gcd(n, d)
 in
 (n div g, d div g) : rat
 end
 fun num (q : rat) = #1 q
 fun den q = #2 (q : rat)

 fun addRat(x, y) = makeRat(num x * den y + num y * den x, den x * den y)
 fun subRat(x, y) = makeRat(num x * den y - num y * den x, den x * den y)
 fun mulRat(x, y) = makeRat(num x * num y, den x * den y)
 fun divRat(x, y) = makeRat(num x * den y, den x * num y)
 fun equRat(x, y) = num x * den y = den x * num y
 fun printRat q = print(makestring(num q) ^ "/" ^ makestring(den q) ^ "\n");

 val one = makeRat(1,1)
 val zero = makeRat(0,1)
 val oneHalf = makeRat(1,2)
 val oneThird = makeRat(1,3)
 val twoThird = makeRat(2,3)
end;
```



## Adatabsztrakció modulokkal: racionális számok (folyt.)

```
signature Rat =
sig
 eqtype rat
 val makeRat : int * int -> rat
 val num : rat -> int
 val den : rat -> int
 val addRat : rat * rat -> rat
 val subRat : rat * rat -> rat
 val mulRat : rat * rat -> rat
 val divRat : rat * rat -> rat
 val equRat : rat * rat -> bool
 val printRat : rat -> unit
 val one : rat
 val oneHalf : rat
 val oneThird : rat
 val twoThird : rat
 val zero : rat
end;

Rat2 néven a Rat struktúra egy változatát így is létrehozhatjuk a fenti, egyenlőségi típust használó Rat szignatúrával:

structure Rat2 :> Rat = Rat;
```

```
> signature Rat =
 /\=rat.
 {type rat = rat,
 val makeRat : int * int -> rat,
 val num : rat -> int,
 val den : rat -> int,
 val addRat : rat * rat -> rat,
 val subRat : rat * rat -> rat,
 val mulRat : rat * rat -> rat,
 val divRat : rat * rat -> rat,
 val equRat : rat * rat -> bool,
 val printRat : rat -> unit,
 val one : rat,
 val oneHalf : rat,
 val oneThird : rat,
 val twoThird : rat,
 val zero : rat}
```

## Adatabsztrakció modulokkal: racionális számok (folyt.)

- A Rat struktúrában definiált értékekre teljes nevükkel kell hivatkozni:

```
Rat.printRat(Rat.mulRat(Rat.oneHalf, Rat.oneThird));
Rat.printRat(Rat.addRat(Rat.oneThird, Rat.oneThird));
```

- open-nel – a szignatúra által korlátozott mértékben – láthatóvá tehetjük a struktúra tartalmát:

```
open Rat2;
equRat(addRat(oneThird, oneThird), twoThird);
addRat(oneThird, oneThird) = twoThird;
```

- A láthatóvá tétel lehet lokális (deklaráció, ill. kifejezés lokális deklarációval):

```
local open Rat2
 val q1 = addRat(oneThird, oneThird); val q2 = twoThird
in val ratPair = (q1, q2)
end;

let open Rat2
in printRat(addRat(oneThird, oneThird))
end;
```

## Adatabsztrakció modulokkal: racionális számok (folyt.)

---

- Válasszunk a matematikában megszokotthoz közelebb álló neveket a függvényeknek:

```
signature Rat =
sig
 eqtype rat
 val rat : int * int -> rat
 val num : rat -> int
 val den : rat -> int
 val ++ : rat * rat -> rat
 val -- : rat * rat -> rat
 val ** : rat * rat -> rat
 val // : rat * rat -> rat
 val == : rat * rat -> bool
 val toString : rat -> string
 val one : rat
 val oneHalf : rat
 val oneThird : rat
 val twoThird : rat
 val zero : rat
end;
```

## Adatabsztrakció modulokkal: racionális számok (folyt.)

---

```
structure Rat3 :> Rat =
struct
 type rat = int * int;
 fun rat (n, d) =
 let val g = Gcd.gcd(n, d) in (n div g, d div g) : rat end
 fun num (q : rat) = #1 q
 fun den q = #2 (q : rat)
 fun op++(x, y) = rat(num x * den y + num y * den x, den x * den y)
 fun op--(x, y) = rat(num x * den y - num y * den x, den x * den y)
 fun op**(x, y) = rat(num x * num y, den x * den y)
 fun op//(x, y) = rat(num x * den y, den x * num y)
 fun op==(x, y) = num x * den y = den x * num y
 fun toString r = makestring(num r) ^ "/" ^ makestring(den r)
 val one = rat(1,1)
 val zero = rat(0,1)
 val oneHalf = rat(1,2)
 val oneThird = rat(1,3)
 val twoThird = rat(2,3) end;
```



## Adatabsztrakció modulokkal: racionális számok (folyt.)

---

- Az új műveleti jelek prefix pozícióban használhatók:

```
let open Rat3
in
 print(toString(++(**(oneThird, oneHalf), oneThird)) ^ "\n");
 ++(oneThird, oneThird) = twoThird
end;
```

A ( és a \*\* közé legalább egy szóköz kell, különben az mosml *megjegyzés* kezdetének veszi!

- Vagy akár infix pozíciójává alakíthatók:

```
let open Rat3
 infix 6 ++ --
 infix 7 ** //
in
 print(toString(oneThird ** oneHalf ++ oneThird) ^ "\n");
 oneThird ++ oneThird = twoThird
end;
```

## Adatabsztrakció modulokkal: racionális számok (folyt.)

---

- A szokásos alapműveleti jeleket is újradefiniálhatjuk.
- Eredeti jelentésük nem vész el, de a műveletek teljes nevét kell használnunk *prefix* pozícióban:

```
load "Int";
let open Rat3
 val op+ = ++
 val op- = --
 val op* = **
 val op/ = //
in
 print(toString oneHalf ^ "\n");
 print(toString(oneHalf + oneThird) ^ "\n");
 print(toString(oneHalf * oneThird) ^ "\n");
 print(toString(oneThird - oneThird) ^ "\n");
 print(toString(twoThird / oneThird) ^ "\n");
 oneThird + oneThird = twoThird;
 Int.+(1,2)
end;
```

Jegyezzük meg, hogy Int . + infix pozícióban nem használható:

```
1 Int.+ 2 (* hibás! *)
```

## Adatabsztrakció modulokkal: racionális számok (folyt.)

---

- Új típust és konstruktorokat hozhatunk létre a datatype deklarációval:

```

structure Rat4 :> Rat =
struct
 datatype rat = Rat of int * int
 fun rat (n, d) = let val g = Gcd.gcd(n, d) in Rat(n div g, d div g)
 fun num (Rat q) = #1 q
 fun den (Rat q) = #2 q
 fun op++(x, y) = rat(num x * den y + num y * den x, den x * den y)
 fun op--(x, y) = rat(num x * den y - num y * den x, den x * den y)
 fun op**(x, y) = rat(num x * num y, den x * den y)
 fun op//(x, y) = rat(num x * den y, den x * num y)
 fun op==(x, y) = num x * den y = den x * num y
 fun toString r = makestring(num r) ^ "/" ^ makestring(den r);
 val one = rat(1,1)
 val zero = rat(0,1)
 val oneHalf = rat(1,2)
 val oneThird = rat(1,3)
 val twoThird = rat(2,3) end;

```

## Adatabsztrakció modulokkal: racionális számok (folyt.)

---

- Az adatkonstruktor mintaillesztésre *szelektorként* is felhasználható (és használni is kell):

```

structure Rat5 :> Rat =
struct
 datatype rat = Rat of int * int;
 fun rat (n, d) = let val g = Gcd.gcd(n, d) in Rat(n div g, d div g)
 fun num (Rat(n, _)) = n
 fun den (Rat(_, d)) = d
 fun op++(x, y) = rat(num x * den y + num y * den x, den x * den y)
 fun op--(x, y) = rat(num x * den y - num y * den x, den x * den y)
 fun op**(x, y) = rat(num x * num y, den x * den y)
 fun op//(x, y) = rat(num x * den y, den x * num y)
 fun op==(x, y) = num x * den y = den x * num y
 fun toString r = makestring(num r) ^ "/" ^ makestring(den r);
 val one = rat(1,1)
 val zero = rat(0,1)
 val oneHalf = rat(1,2)
 val oneThird = rat(1,3)
 val twoThird = rat(2,3) end;

```

## Adatabsztrakció modulokkal: racionális számok (folyt.)

---

- Az adatkonstruktorfüggvény *valóban* használható új érték létrehozására:

```

structure Rat6 :> Rat =
struct
 datatype rat = Rat of int * int;
 val rat = Rat;
 fun num (Rat(n, _)) = n
 fun den (Rat(_, d)) = d
 fun op++(x, y) = rat(num x * den y + num y * den x, den x * den y)
 fun op--(x, y) = rat(num x * den y - num y * den x, den x * den y)
 fun op**(x, y) = rat(num x * num y, den x * den y)
 fun op//(x, y) = rat(num x * den y, den x * num y)
 fun op==(x, y) = num x * den y = den x * num y
 fun toString r = makestring(num r) ^ "/" ^ makestring(den r);
 val one = rat(1,1)
 val zero = rat(0,1)
 val oneHalf = rat(1,2)
 val oneThird = rat(1,3)
 val twoThird = rat(2,3) end;

```

## Összefoglalás: gyenge és erős adatabsztrakció

- Gyenge absztrakció: a név szinonima, az adatszerkezet részei továbbra is hozzáférhetők.
- Erős absztrakció: a név új dolgot (entitást, objektumot) 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.

## 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 előtt.
- Szintaxisa:
 

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

- Példa:

```
(* len : '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
```

## 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 *adatkonstruktora* (röviden: *konstruktora*) 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 (főnemest) 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.)

---

- 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 egyetlennek* tekintjük:

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

## 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
```

## 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
 | Peer of degree * string * int
 | Knight of string
 | Peasant of string
```

## Felsorolós 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

---

- 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
```

- Az infix hatospontot magában a típusdeklarációban is definiálhatjuk:

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



## 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.

## 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 címkéjű
 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.

ESETSZÉTVÁLASZTÁS, OPCIONÁLIS ÉRTÉK

---

## Esetszétválasztás (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 édesítőszer, ui. helyettesíthető fn-jelöléssel:

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

Például a lady függvényt így is definiálhattuk volna:

|                                                                                                                                                                                                                                                                                                                                                     |                                                                                                                                                                                                                                                                                          |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>datatype degree = Duke   Marquis   Earl   Viscount   Baron (* lady : degree -&gt; string    lady p = p főnemes            hitvesének rangja *) fun lady p =   case p of     Duke      =&gt; "Duchess "     Marquis   =&gt; "Marchioness"     Earl      =&gt; "Countess"     Viscount  =&gt; "Viscountess"     Baron     =&gt; "Baroness"</pre> | <pre>(* lady : degree -&gt; string    lady p = p főnemes            hitvesének rangja *) fun lady p =   (fn     Duke      =&gt; "Duchess "     Marquis   =&gt; "Marchioness"     Earl      =&gt; "Countess"     Viscount  =&gt; "Viscountess"     Baron     =&gt; "Baroness"   ) p</pre> |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

## 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.

## Peldak opcionalis ertek kezelésere

---

- Egszlista legnagyobb elemnek kivalasztasa

ures listnak nincs legnagyobb eleme; egyelem lista egyetlen eleme a „legnagyobb”; legalbb ketelem lista legnagyobb eleme az elso elem s a maradeklista elemei kozul a legnagyobb.

```
(* maxl : int list -> int option
 maxl ns = az ns egszlista legnagyobb eleme *)
fun maxl [] = NONE (* ures *)
 | maxl [n] = SOME n (* egyelem *)
 | maxl (n::ns) = (* legalbb ketelem *)
 SOME(Int.max(n, valOf(maxl ns)))
```

- Fuzer elejen allo karaktersorozat atalaktasa egsz szamma

```
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]+
```

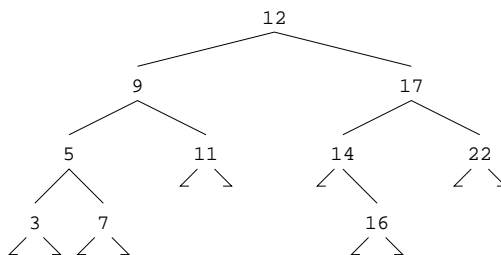
```
Int.fromString "1234"; Int.fromString "-1234"; Int.fromString "~1234";
Int.fromString "+1234"; Int.fromString "+007"; Int.fromString "alma"
```

## Bináris fák datatype deklarációval

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

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

- Tekintsük például az alábbi fát:

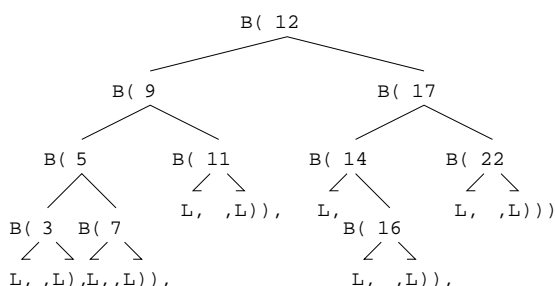


- Az 'a tree adattípus L és B adatkonstruktorokkal ez a fa pl. a következő lapon látható módon írható le.

## Bináris fák datatype deklarációval (folyt.)

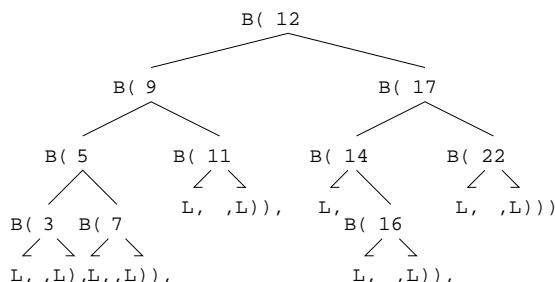
```
B(B(B(B(L, 3, L),
 5,
 B(L, 7, L)
),
 9,
 B(L, 11, L)
),
 12,
 B(B(L,
 14,
 B(L, 16, L)
),
 17,
 B(L, 22, L)
)
);
```

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



## 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 tr5 = B(tr3,5,tr7);
val tr9 = B(tr5,9,tr11);
val tr14 = B(L,14,tr16);
val tr17 = B(tr14,17,tr22);

val tr7 = B(L,7,L);
val tr11 = B(L,11,L);
val tr16 = B(L,16,L);
val tr22 = B(L,22,L);
val tr12 = B(tr9,12,tr17);

```

## 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 csonkokat 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 ágaknak (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.)

---

- 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
```

# BINÁRIS FÁK

Bináris fák FP-1..12-160

## 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)
```



## 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!

## 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
```

## 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.

## take és drop egyetlen függvénnyel: take 'ndrop

- Írjunk take 'ndrop néven olyan függvényt, amelynek egy xs listából és egy k egészből 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:

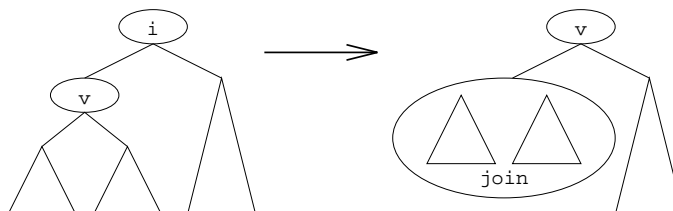
```
(* balPreorder: 'a list -> 'a tree
 balPreorder xs = az xs lista elemeiből álló, preorder ... *)
fun balPreorder [] = L
 | balPreorder (x::xs) =
 let val k = length xs div 2
 val (ts, ds) = take'ndrop(xs, k)
 in N(x, balPreorder ts, balPreorder ds)
 end
```

## Bináris fa előállítása 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 xs 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 meghagyjuk gyakorló feladatnak.

## 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 szétválasztott részfa 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.

## Elem rekurzív törlése bináris fából (folyt.)

---

- A join-nal egyesítjük a törlés hatására létrejövő két részfát: a bal részfát lebontja, és közben az elemeit 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;
```

## 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 meghagyjuk gyakorló feladatnak.

## Kivételkezelés

---

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

## Kivételkezelés (folyt.)

---

- `raise` alkalmazásának eredménye az ún. *kivételcsomag*. Mivel a kivételcsomag polimorf típusú, bármely más típussal 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` ( $1 \leq i \leq n$ ) az első illeszkedő minta, akkor `Ei` a kivételkezelő eredménye.
  - Ha egyetlen minta sem illeszkedik a kivételcsomagra, a kivételkezelő továbbpasszolja.
- Példák kivétel kezelésére:
  - `erme :: valtas (erme::ermelista) (osszeg-erme)`  
`handle Valt => valtas ermelista osszeg`
  - `(fn i => kivKez i handle Hiba(c, i) => (print(str c); i-1)) 0`
- `handle` (hipotetikus) típusa: `exn -> 'a.`
- Legyen `Ex` `exn` típusú kivétel, `e` pedig tetszőleges kifejezés; ekkor az `e handle Ex => c` (kivételkezelőt tartalmazó) kifejezésben `c`-nek `e`-vel azonos típusúnak kell lennie.

## Kivételkezelés (folyt.)

---

- A következő programrészlet példa kivétel deklarálására, jelzésére és kezelésére

```
exception Hiba of char * int;

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

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

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

## Kivételkezelés (folyt.)

---

- Példa visszalépés programozására kivételkezeléssel

```
exception Valt;

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

váltás [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.

## *n* vezér a sakktáblán

### Hányféleképpen rakható *n* vezér egy $n * n$ méretű sakktáblára úgy, hogy ne üssék egymást?

- A sakktáblát egy olyan *n* hosszú sorvektorral írjuk le, amelynek egy-egy mezőjébe írt *s* szám a sakktábla egy-egy oszlopába lerakott vezér sorának a sorszáma ( $0 \leq s < n$ ).
- Példa  $n=4$  esetén:

```

+---+---+---+---+
| 2 | 0 | 3 | 1 |
+---+---+---+---+
0 <----> n-1

+---+---+---+---+
0 | | q | | |
+---+---+---+---+
| | | | |
+---+---+---+---+
| | | | |
+---+---+---+---+
v | q | | | |
+---+---+---+---+
n-1 | | | q | |
+---+---+---+---+

```

- A sorvektort listával valósítjuk meg.
- Egy listához *balról könnyű* új elemeket fűzni, ezért a táblát és a vezérek helyzetét leíró listát függőlegesen tükrözzük.

```

...+---+---+---+---+
| 3 | 0 | 2 |
...+---+---+---+---+
n-1 <---- 0

...+---+---+---+---+
0 | | q | | |
...+---+---+---+---+
| | | | |
+---+---+---+---+
v | | | | |
+---+---+---+---+
n-1 | q | | | |
...+---+---+---+---+

```

- Egy *n* hosszú sorvektor *i*-edik eleme az  $n - (i + 1)$ -edik elem a listában.

## *n* vezér a sakktáblán (folyt.)

Azt, hogy az új vezért ütik-e másik vezérek a táblán, a *sorvektor* vizsgálatával dönthetjük el: az egyes elemek sorszáma által meghatározott oszlopban az értékük által meghatározott sorban vezér van. A lerakás feltételei a következők:

1. Az új vezér nem kerülhet egy sorba egyetlen más vezérrel sem, azaz az új listaelem *értéke* nem fordulhat elő a lista már felépített részében.
2. Az új vezér átlós irányban sem lehet egy vonalban más vezérrel. Ez azt jelenti, hogy ha a lista elejére (a 0. elemébe!) az *s* sorindexet akarjuk írni, akkor az *i*-edik listaelem értéke, feltéve, hogy van ilyen elem, nem lehet  $s - i$ , sem  $s + i$ .  
A következő példa megvilágítja az esetet.

Ha a tábla 1. sorába akarjuk lerakni az új vezért, akkor az *x*-szel megjelölt mezőket kell megvizsgálunk. Az új mezővel együtt a listának már három eleme van. Az 1-es indexű elem nem lehet  $s - 1$ , sem  $s + 1$ , a 2-es indexű elem pedig nem lehet  $s - 2$ , sem  $s + 2$ .

```

...+---+---+---+---+
| 1 | | | |
...+---+---+---+---+

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

```

A lista rekurzív algoritmussal dolgozható fel.

## *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 legalább egy
 rs-beli vezér 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: 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 (* vissza kell lépni, ha z=0 és ütésben van *)
 z = 0 andalso utesbenVan zs orelse
 (* vissza kell lépni, ha már minden sort megpróbált *)
 z = n
 then raise Zsakutca
 else if length zs = n
 then rev zs (* megvan egy megoldás *)
 else (* folytatja a 0. sortól a következő vezér lerakásával,
 és ha elakad, visszalép a következő sorra *)
 vez 0 (z::zs) handle Zsakutca => vez (z+1) zs
 in
 (* a 0. sorral kezd: ilyenkor nem lehet ütésben *)
 vez 0 []
 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 (* vissza kell lépni, ha z=0 és ütésben van, vagy ha már
 minden sort megpróbáltunk *)
 z = 0 andalso utesbenVan zs orelse z = n
 then raise Zsakutca
 else if length zs = n
 then [rev zs] (* megvan egy megoldás, listában adja vissza *)
 else (* folytatja a következő sorral, majd hozzáfüzi ... *)
 (vez (z+1) zs handle Zsakutca => []) @
 (* ... a 0. sortól kezdve a következő vezér lerakásával
 kapott megoldásokat *)
 (vez 0 (z::zs) handle Zsakutca => [])
 in
 (* a 0. sorral kezd: ilyenkor nem lehet ütésben *)
 vez 0 []
 end
end

```

## ***n* vezér a sakktáblán: több megoldás előállítása listák listájával**

---

Az előző megoldás sémája sokszor használható, de ebben az egyszerű esetben felesleges: a kivétel jelzése helyett üres listát adhatunk eredményül, hiszen a kivételkezelők is csak ezt teszik.

```

(* 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 (z+1) zs @ vez 0 (z::zs)
 in
 vez 0 []
 end
end

```

## ***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
 end
```

## 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 op isMem (_, []) = false
 | op isMem (x, y::ys) = x = y orelse op isMem(x, ys)
infix isMem
```

Megjegyzés: az op operátor nélkül az infix deklarációt követően a függvénydefiníciót nem lehetne újrafordítani.

- 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,  $S \cap T$ ),
- részalmaz-e (isSubset,  $T \subseteq S$ ),
- egyenlők-e (isSetEq,  $S = T$ ),
- hatványhalmaz (powerSet,  $pS$ ).

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

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

- Két halmaz uniója

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

- Két halmaz metszete

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

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

- Részhalmaza-e egy halmaz egy másiknak?

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

infix isSubset
```

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

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

## Halmazműveletek: „halmaz hatványhalmaza” (powerSet)

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

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

## Halmazműveletek: „halmaz hatványhalmaza” (folyt.)

- Ezzel a `pws` függvény:

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

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

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



## Halmazműveletek: „halmaz hatványhalmaza”, hatékonyabban

---

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

- Az insAll segédfüggvény egy elemet szúr be egy 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

---

## Egyidejű deklaráció

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

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

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

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

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

## Egyidejű deklaráció (folyt.)

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

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

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

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

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

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

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

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

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

# AZ ORDER TÍPUS

Az order típus FP-1..12-198

## Az order típus

---

Az order típus definíciója (ld. `General.sig`)

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

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

Példák az SML-alapkönyvtárból (SML Basis Library)

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

# LISTÁK RENDEZÉSE

---

Listák rendezése FP-1..12-200

## Listák rendezése

---

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

## Beszűrő 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];
```

## Beszűrő 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 _ [] = []
```

## Beszűrő 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
```

## Beszűrő 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;
```

## A futási idők mérése, összehasonlítása (folyt.)

---

- A futási időt az alábbi függvénnyel mérhetjük:

```
app load ["Timer", "Time", "Int"];

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

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

## A futási idők mérése, összehasonlítása (folyt.)

---

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

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

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

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

## Kiválasztó rendezés

---

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

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

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



## Kiválasztó rendezés (folyt.)

```

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

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

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

```

## Gyorsrendezés akkumulátor nélkül

```

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

```

## Gyorsrendezés akkumulátorral

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

## A futási idők mérése, összehasonlítása

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

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

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

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

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

## Összefésülő rendezések

---

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

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

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

## Fölülről lefelé haladó összefésülő rendezés

---

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

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

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