BEVEZETÉS A FUNKCIONÁLIS PROGRAMOZÁSBA

Az előadássorozat áttekintése

- Bevezetés. Az SML nyelv alapjai.
- Egyszerű és összetett adattípusok. Programfejlesztés.
- Polimorfizmus. Listaműveletek. A legfontosabb programkönyvtárak.
- Programhelyesség, programbizonyítás.
- Magasabbrendű függvények.
- Modulok. Absztrakt adattípusok. Paraméterezhető modulok.
- Nemlineáris rekurzív adattípusok.
- Nagyobb SML-példák.
- Új irányzatok a funkcionális programozásban.

A funkcionális programozás motivációi

- Rekurzió, teljes indukció (vö. gépi kód, Fortran, Basic) – 1950-es évek
- Lineáris rekurzív adatszerkezet (lista, vö. ciklus)
- Függvények – vissza a matematikához! (vö. mellékhatás) – 1960-as évek
- Erős típusok, ellenőrzés fordításkor (vö. típusnélküli nyelvek) – 1970-es évek
- Rekurzív adattípusok (fa, vö. láncolt adatszerkezetek)
- Absztrakt adattípusok (vö. objektumok)
- Végrehajtható specifikációk (vö. tesztelés) – 1990-es évek

Mi az alapvető különbség a deklaratív és az imperatív programozás között?

- A deklaratív programozás *időtlen*, nem törődik az idővel.
- Idő $\longrightarrow$ állapot $\longrightarrow$ emlékezet.

A funkcionális programozás rövid története

- A függvényfogalom fejlődése – l. külön fóliákon: `fffp.pdf`.
- Euler (1748): $\mathrm{sin}.x$ később $\sin x$ vagy $\sin(x)$
- Alfred N. Whitehead, Bertrand Russel (1910) ... Alonzo Church: λ-***kalkulus***, λ-jelölés: $\lambda x.x + x$
- Church, 1936: λ-kalkulus (funkcionális) $\equiv$ Turing-gép (imperatív) $\longrightarrow$ funkcionális programozás $\equiv$ imperatív programozás
- Church-tétel: kiszámítható függvények halmaza $\equiv$ rekurzív függvények halmaza – ez a funkcionális programozás alapja
- 1960: ALGOL (ALGOrithmic Language) – rekurzív eljárás és függvényeljárás (!)
- 1960: LISP (LISt Processing language) – alapja a λ-kalkulus, eredeti célja: ***szimbolikus differenciálás***
- 1962-től: APL, ML, HOPE, ERLANG, Miranda, SML, Haskell, gofer, clean stb.

Az ML (Meta Language) rövid története és jelene

Az ML rövid története

- ML, Edinborough 1977, tételbizonyításra (kijelentések igazolására)
- Definition of Standard ML, 1990
  - Alapnyelv (Core Language)
  - Modulnyelv (Module Language)
- Revised Definition of Standard ML, 1997
- SML Basis Library (Alapkönyvtár), 1997

SML-megvalósítások

- Moscow ML (mosml): `http://www.dina.kvl.dk/ sestoft/mosml.html`
- Standard ML of New Jersey (sml): `http://cm.bell-labs.com/cm/cs/what/smlnj`

Információk a funkcionális programozásról

Hálózati információforrások:

Comp.Lang.ML FAQ

`http://www.cis.ohio-state.edu/hypertext/faq/usenet/meta-lang-faq/faq.html`

Andrew Cumming: A Gentle Introduction to ML

`http://www.dcs.napier.ac.uk/course-notes/sml/manual.html`

Stephen Gilmore: Programming in Standard ML '97

`http://www.dcs.ed.ac.uk/home/stg`

Robert Harper: Programming in Standard ML

`http://www.cs.cmu.edu/People/rwh/`

Fox project at CMU

`http://foxnet.cs.cmu.edu/sml.html`

SML-irodalom (csak angolul)

Forrásművek az előadásokhoz

Jeffrey D. Ullman: ***Elements of ML Programming*** (2nd Edition, ML97)
MIT Press 1997
`http://www-db.stanford.edu/ ullman/emlp.html`

Lawrence C. Paulson: ***ML for the Working Programmer*** (2nd Edition, ML97)
Cambridge University Press 1996
`http://www.cl.cam.ac.uk/users/lcp/MLbook/`

Richard Bosworth: ***A Practical Course in Functional Programming Using Standard ML***
McGraw-Hill 1995

A FÜGGVÉNY FOGALMA ÉS TULAJDONSÁGAI

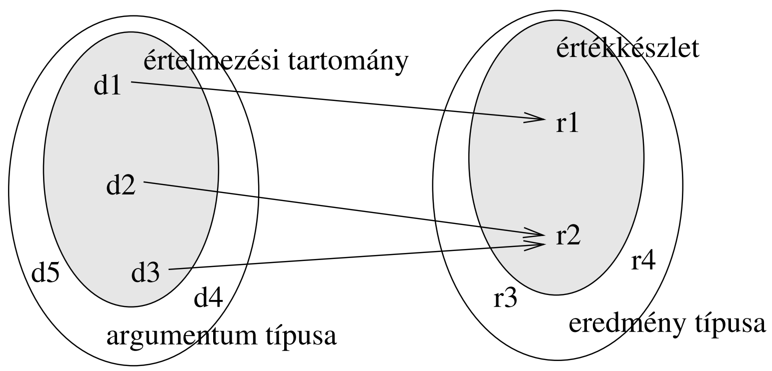
A típus és a függvény fogalma

- A típus fogalma
  - A típus értékek egy halmaza (pl. egész típus = az egész számok halmaza)
  - Jelölése: α, β, ... (az ún. ***típuselméletben*** így használják)
- A függvény fogalma
  - A függvény valamely D halmaznak valamely R halmazba való olyan ***egyértelmű*** leképzése, amelyet meghatároz a $(d; r)$ rendezett párok halmaza, ahol $d \in D$ és $r \in R$.
  - A d a függvény argumentuma (paramétere), az r az eredménye
  - A D a függvény értelmezési tartománya, az R az értékkészlete
  - A típusos nyelvekben d is, r is ***meghatározott*** típusú
  - Függvény értelmezési tartománya $\subseteq$ argumentum típusa
  - Függvény értékkészlete $\subseteq$ eredmény típusa

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 – érték: ***függvényérték***
  - Fontos: a függvényérték ***nem*** a függvény ***alkalmazásának*** az eredménye!
  - Példák függvényértékre
    - sin (a típusa: ***valós*** $\rightarrow$ ***valós***)
    - round (a típusa: ***valós*** $\rightarrow$ ***egész***)
    - $f \circ g$ (a típusa: $\alpha \rightarrow \beta$)
  - Példa függvényalkalmazásra
    - $\mathrm{round}\ 5.4 = 5$, azaz ennek a függvényalkalmazásnak egy ***egész*** típusú érték az eredménye

A függvény mint leképzés



Függvények tulajdonságai és osztályozása

- Parciális függvény: értelmezési tartomány $\subset$ argumentum típusa

  Figyelem: ez hibák forrása lehet!
- Teljes függvény: értelmezési tartomány = argumentum típusa
- Szürjektív függvény: értékkészlet = eredmény típusa
- Nem-szürjektív függvény: értékkészlet $\subset$ eredmény típusa
- Injektív függvény: a leképzés kölcsönösen egyértelmű
- Az $f : \alpha \rightarrow \beta$ injektív függvény inverze: $f^{-1} : \beta \rightarrow \alpha$
- Bijektív = injektív + szürjektív, azaz f bijektív, ha f^{-1} teljes függvény

Függvények alkalmazása

- ***Függvényalkalmazást*** jelöl az f és e jelek egymás mellé írása (***„juxtapozicionálása"***): $f\,e$ azt jelenti, hogy f-et alkalmazzuk e-re.
- Általánosabban: az fe kifejezésben az e tetszőleges olyan kifejezés, amelynek az értéke az f értelmezési tartományába esik.
- Még általánosabban: az fe kifejezésben az f függvényértéket eredményező tetszőleges kifejezés, e pedig tetszőleges olyan kifejezés, amelynek az értéke az f értelmezési tartományába esik.

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.
  2. A függvényt több egymás utáni lépésben alkalmazzuk az argumentumokra, pl. $f12 \equiv (f1)2$ 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 $f12$ függvényalkalmazás (vég)eredményét.
- Infix jelölés: $x \oplus y \equiv$ az $\oplus$ függvény alkalmazása az (x,y) párra mint argumentumra

FÜGGVÉNYEK AZ SML-BEN

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 (␣, \t, \n stb.). Nulla db formázó karakter elegendő pl. a (előtt és a) után.
- A szeparátor a legerősebb balra kötő infix operátor az SML-ben.
- Példák:
  `Math.sin 1.00, (Math.cos)Math.pi, round(3.17), 2 + 3, (real) (3 + 2 * 5)`
- Függvények egy csoportosítása az SML-ben
  - Beépített függvények, pl. `+`, `*` (infix), `real`, `round` (prefix)
  - Könyvtári függvények, pl. `Math.sin`, `Math.cos`, `Math.pi`
  - Felhasználó által definiálható függvények, pl. `terulet`, `/\`, `head`

SML-példa: Egyszeres Hamming-távolságú ciklikus kód

- A függvényt ***táblázattal*** adjuk meg:

00	01
01	11
11	10
10	00

```
fn 00 => 01
 | 01 => 11
 | 11 => 10
 | 10 => 00
```

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

SML-példa: modulo n alapú inkrementálás

- A függvényt most ***algoritmussal*** adjuk meg, nem táblázattal
  - `n` nem lehetne változó, túl sok változatot kellene felírni stb.
- `fn i => (i + 1) mod n`
  - az `i` ún. kötött változó, a névtelen függvény argumentuma
  - az `n` ebben a kifejezésben szabad változó, és nincs értéke (!)
  - az `n`-et is le kell kötni mint a függvény argumentumát
- `fn i => fn n => (i + 1) mod n`
- A függvény néhány alkalmazása:
  - `(fn i => (fn n => (i + 1) mod n) 4) 1)`
  - `(fn i => (fn n => (i + 1) mod n) 128) 111`
  - `(fn i => (fn n => (i + 1) mod n) 4) ~7`
  - `(fn i => (fn n => (i + 1) mod n) 128) 6.0` – hibás!

Értékdeklaráció SML-ben: függvényérték deklarálása

- Név kötése függvényértékhez
  - `val incMod = fn i => fn n => (i + 1) mod n`
  - `val kovKod = fn 00 => 01 | 01 => 11 | 11 => 10 | 10 => 00`
- Szintaktikai édesítőszerrel
  - `fun incMod n i = (i + 1) mod n`
    - Figyelem: `i` és `n` sorrendje megfordult!
  - ```
    fun kovKod 00 = 01
      | kovKod 01 = 11
      | kovKod 11 = 10
      | kovKod 10 = 00
    ```
- Alkalmazásuk argumentumra
  - `incMod 128 111`
  - `kovKod 01`

Fejkomment

Legyen ***fejkomment*** minden (függvény)érték-deklarációhoz!

- ```
  (* incMod n i = (i+1) modulo n szerint
     PRE: n > 0, n > i >= 0
  *)
  fun incMod n i = (i + 1) mod n
  ```
- ```
  (* kovKod cc = a kétbites, egyszeres Hamming-távolságú, 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
  ```
  ```

A FÜGGVÉNYFOGALOM KIALAKULÁSA

A függvényfogalom: absztrakció

Absztrakció, oksági viszony

- A függvényfogalom minden bizonnyal olyan absztrakció, amelynek eredete az emberek által már igen régen észrevett oksági viszonyban gyökrezik. Ennek a mind tudatosabbá váló ok-okozati viszonynak már a korai matematikában jelentkeztek különböző megfogalmazásai.

Már a régi görögök, sőt már az egyiptomiak, babiloniak is...

- Lényegében ezt fejezték ki a számolást megkönnyítő egyiptomi és babiloni táblázatok. Az ógörög matematikusok a mennyiségi törvényeket kutatva, a függvényfogalmat rejtő összefüggések tömegét fogalmazták meg...

A XVII. század – Descartes, Fermat, Leibniz, Bernoulli testvérek

Végül is a francia Descartes (René, du Peron, 1596-1650) nagy matematikai tette volt a függvényfogalom első definíciója. Csodálatos lényeglátással a függvényt megfeleltetésnek definiálta, bár ő még csak az algebrai műveletekkel meghatározott függvényekkel foglalkozott. Descartes, és vele egy időben és ugyanolyan érdemekkel, a francia Fermat (Pierre, 1601-1665) megteremtette a változó mennyiségek matematikáját.
A függvényfogalom további alakítása a német Leibniz (Gottfried Wilhelm, 1646-1716) nevéhez fűződik, de az ő értelmezése szűkebb Descartes-énál. Ő használta először 1692-ben a latin *functio* szót valamely görbe egy pontjához tartozó olyan szakaszra, amely változik, ha a pont végigfut a görbén (ordináta, abszcissza, szubtangens stb.). Ekkor vezette be a paraméter, az állandó, a változó és más kifejezéseket is.
A XVII. század végétől, a XVIII. század elejétől függvénynek tekintették azt az analitikus kifejezést, amely kifejezte a változók és az állandók közötti kapcsolatot. Ilyen értelemben használta a függvény szót a két svájci Bernoulli testvér (Jacob 1654-1705, Johann 1667-1748), és ezt a fogalmat Johann B. zárójel nélkül φx-szel jelölte.

A XVIII. század – Euler

Ezt az értelmezést vette át a svájci Euler (Leonhard, 1707-1783) is, aki a φ betű helyett az f-et kezdte használni, és megengedte a komplex változókat is. Euler szerint tehát függvényen értjük a változók és a konstansok közötti kapcsolatot leíró kifejezést, ha az analitikus műveleteket (négy alapművelet, hatványozás, gyökvonás, sorbafejtés, differenciálás, integrálás) tartalmaz. Ez még mindig a függvényeknek csak azt az osztályát ölelte át, amelyeket teljes értelmezési tartományukban már meghatároz grafikonjaik bármilyen kicsiny darabja. Ezenkívül azonban Euler foglalkozott az e^x, az $\ln x$ és a trigonometrikus függvényekkel is...
Euler kezdetben azt hitte, hogy minden függvény hatványsorba, azaz

$$fz = a_0 + a_1 z + a_2 z + a_3 z + \cdots$$

alakba fejthető. A differenciálegyenletek vizsgálatánál azonban olyan függvényekre bukkant, amelyeket megadhatott tetszőleges alakú grafikon is. Ezekről azt gondolta, hogy nem analítikus függvények, azaz nem fejthetők hatványsorba...

A XIX. század – Bolzano, Dirichlet, Cauchy, Weierstrass, Gauss, Riemann

A függvényelmélet voltaképpen a cseh Bolzano (Bernhard, 1781-1848) és a német Dirichlet (Peter Gustav Lejeune, 1805-1859) eredményei alapján indult igazán fejlődésnek, és a XIX. században az előzmények biztos talaján a francia Cauchy (Augustin Louis 1789-1857) és a német Weierstrass (Karl, 1815-1897) a legnagyobb szabatossággal önthette szavakba a függvénytani tulajdonságokat és fogalmakat.
A valós függvénytan kialakulása után a komplex változós függvénytan is biztos alapokra talált a komplex számoknak a német Gauss (Carl Friedrich, 1777-1855) alkotta elmélete segítségével.
A komplex változójú függvények elméletének megteremtésében Cauchy és Weierstrass mellett nagy szerepe volt a német Riemann-nak (Georg Friedrich Bernhard, 1826-1866) is, aki a geometriai függvénytan életre hívásával a komplex függvénytan új megalapozását tette lehetővé.

A XX. század első fele – Volterra, Fréchet, Riesz, Hilbert

A függvényfogalom halmazelméleti definíciója nem teszi szükségessé, hogy a megfeleltetés számok halmazait kapcsolja össze. Az értelmezési tartomány és az értékkészlet elemei tetszőleges matematikai objektumok lehetnek.
Az absztrakciónak ez a további fokozata a függvénytan új területeit hozta létre. Ha az értelmezési tartomány függvények halmaza és az értékkészlet számok halmaza, akkor az egymáshoz rendelést funkcionálnak nevezzük. Ha pedig az értelmezési tartomány és az értékkészlet is függvények halmaza, akkor a megfeleltetés neve operátor.
A funkcionálokkal és az operátorokkal foglalkozó funkcionálanalízis különálló kutatási területnek az olasz Volterra (Vito, 1860-1940) munkássága óta számít.
A funkcionálelmélet kibontakozásában kimagasló érdemei vannak a francia Fréchet-nek (Maurice, 1878-1973), a magyar Riesz Frigyesnek (1880-1956) és a német Hilbertnek (David, 1862-1943).

A függvény

A függvény két halmaz között olyan megfeleltetés, amely az egyik halmaz minden eleméhez hozzárendeli egy másik halmaz pontosan egy elemét.
Formálisabb megfogalmazások:

- A függvény olyan $(x; y)$ rendezett párok halmaza, amelyben minden x-hez pontosan egy y tartozik.
- A függvény olyan bináris reláció, amelyre igaz, hogy ha $(x; y)$ és $(x; y')$ mindegyike eleme a relációnak, akkor $y = y'$.
- A függvény valamely X halmaznak valamely Y halmazba való olyan egyértelmű leképzése, amelyet meghatároz az $(x; y)$ rendezett párok halmaza, ahol $x \in X$ és $y \in Y$.

Irodalom

- Sain Márton: Nincs királyi út! Matematikatörténet. Gondolat, Budapest, 1986., pp. 697-702.
- Sain Márton: Matematikatörténeti ABC. Tankönykiadó, Budapest, 1987., p. 122.

TÍPUSOK ÉS ÉRTÉKEK AZ SML-BEN

Típusok

- Típusok és programozási nyelvek
  - Típus nélküli nyelvek, pl. assembly, LISP, Prolog
  - Gyengén típusos nyelvek, pl. Fortran, Algol, BASIC, C, C++, Pascal
  - Erősen típusos nyelvek, pl. Ada, SML, clean
  - Erős típus: a típusok (∼ halmazok) diszjunktak (nincs közös elemük)
- Egyszerű SML-típusok
  - `int` – előjeles egész szám, a Z egy részhalmaza
  - `word`, `word8` – előjel nélküli pozitív egész, az N_0 egy részhalmaza
  - `real` – előjeles racionális (valós?!) szám, a Q egy részhalmaza
  - `bool`, `char`, `order`, `unit`
  - `string`
- Összetett SML-típusok (példák)
  - `rekord`
  - `lista`

Értékdeklaráció az SML-ben: név kötése tetszőleges értékhez

- Függvényértéket így kötöttünk tetszőleges névhez:

```
val incMod = fn i => fn n => (i + 1) mod n
```

- Tetszőleges típusú érték köthető tetszőleges névhez:

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

- Típusmegkötés:

`val id = fn (n : int) => n`	**Példák:** `id 3;`, `id 4.5;`
`val telenek = [0w65, 0wx41 : word8]`	**Típusa:** `word8 list`

EGYSZERŰSÍTETT SML-SZINTAXIS

SML-szintaxis: különleges állandó

- Előjeles egész állandó

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

- Valós állandó

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

- Előjel nélküli egész állandó

Példák:	`0w0`	`0w4`	`0w999999`	`0wxFFFF`	`0wx1ff`	
Ellenpéldák:	`0w0.0`	`~0w4`	`-0w4`	`0w1E0`	`0wXFFFF`	`0WxFFFF`

- Füzérállandó: `"`-ek között álló nulla vagy több nyomtatható karakter, szóköz vagy \ jellel kezdődő *escape-szekvencia* (l. a táblázatot a következő lapon).
- Karakterállandó: `#` jelet közvetlenül követő, egykarakteres füzérállandó.

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

SML-szintaxis: escape-szekvenciák

- 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).
`\u`*xxxx*	A *xxxx* kódú karakter (*x* hexadecimális számjegy).
`\"`	Idézőjel (").
`\\`	Hátratört-vonal (\).
`\`$f \cdots f$`\`	Figyelmen kívül hagyott sorozat. $f \cdots f$ nulla vagy több formázókaraktert (szóköz, HT, LF, VT, FF, CR) jelent.

SML-szintaxis: név

- Alfanumerikus: kis- és nagybetűk, számjegyek, percjelek (') és aláhúzás-jelek (_) olyan sorozata, amely betűvel vagy percjellel kezdődik
  - Példák: `tothGyorgy` `Toth_3_Gyorgy` `toth'gyorgy`
- Szimbolikus: az alábbi jelek tetszőleges, nem üres sorozata

  `! % & $ # + - / : < = > ? @ \ ~ ' ^ | *`
  - Példák: `++` `<->` `|||` `##` `|=|`
- Speciális a szerepe az alábbi fenntartott jeleknek

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

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


SML-szintaxis: szintaktikai kategóriák (egyszerűsítve)

- A nevek és más azonosítók *szintaktikai kategóriákba* sorolhatók

vid	értéknév	value identifier	long
tyvar	típusváltozó	type variable	
tycon	típuskonstruktor	type constructor	long
lab	mezőnév	record label	
strid	struktúranév	structure identifier	long
sigid	szignatúranév	signature identifier	
unitid	állománynév	unit identifier	

- Az *értéknév* tetszőleges név; jelölhet állandó értéket, függvényértéket, adatkonstruktort, kivételkonstruktort. Példák: `pi` `+` `sin` `nil` `true` `Match`
- A *típusváltozó* percjellel kezdődő alfanumerikus név. Példa: `'a`.
- A *típuskonstruktor* tetszőleges név; jelölhet típusállandót vagy típusfüggvény-értéket. Példák: `int` `order` `$` `*` `->` `list`
- A *mezőnév* tetszőleges név vagy (nem 0-val kezdődő) pozitív egész szám. Példák: `num` `2`

SML-szintaxis: szintaktikai kategóriák (folyt.)

- Minden, az előző felsorolásban „long”-gal megjelölt X szintaktikai kategóriának van egy *longX* párja. A *longX* szintaktikai kategóriába tartozó nevek rövid és hosszú (ún. minősített) alakban is felírhatók. A *rövid alak* csak egy névből, a *hosszú alak* egy hosszú struktúranévből, egy pontból és egy névből áll:

longx ::=	*x*	név	identifier
	longstrid.x	minősített név	qualified identifier

Példák:

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

SML-szintaxis: szintaktikai kategóriák (folyt.)

- A *struktúranév* és a *szignatúranév* a *modulnyelv* fogalomkörébe tartozó tetszőleges nevek.
  Példák: `Char` `Int` `List` `TextIO`
- Az *állománynév* a *modulnyelv* fogalomkörébe tartozó tetszőleges olyan név, amelyet az adott operációs rendszer is megenged; forráskodú vagy tárgykódú struktúra- vagy szignatúra-állományt azonosít.
- A *strid* struktúranév a `unitid.uo` tárgykódú struktúra-állományra hivatkozik, ahol `unitid` = *strid*. A `unitid.sml` struktúra-állomány fordításakor már léteznie kell a `unitid.ui` tárgykódú szignatúra-állománynak, összeszerkesztésekor pedig már léteznie kell a `unitid.uo` tárgykódú struktúra-állománynak.
- A *sigid* szignatúranév a `unitid.ui` tárgykódú szignatúra-állományra hivatkozik, ahol `unitid` = *sigid*. A `unitid.ui` tárgykódú szignatúra-állományt a `unitid.sig` forráskodú szignatúra-állomány lefordításával kell előállítani.

Struktúra, szignatúra, tárgykódú és forráskódú állományok

Példák

- Struktúra a megfelelő szignatúrával
  - `structure Rat :> Rat = struct` *implementáció* `end`
  - `signature Rat = sig` *specifikáció* `end`
- A `Rat` struktúrát és szignatúrát tartalmazó állományok
  - `Rat.sml`: a forráskódú struktúra-állomány (a `.sml` kiterjesztés használata ajánlott, de nem kötelező)
  - `Rat.sig`: a forráskódú szignatúra-állomány (a `.sig` kiterjesztés használata kötelező)
  - `Rat.uo`: a tárgykódú struktúra-állomány (a `.uo` kiterjesztés használata kötelező)
  - `Rat.ui`: a tárgykódú szignatúra-állomány (a `.ui` kiterjesztés használata kötelező)

Függvényjel helyzete és kötése

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

Függvényjel helyzete és kötése az SML-ben

- Kifejezések és típuskifejezések az SML-ben
  - Az SML-ben a szokásos kifejezések mellett vannak *típuskifejezések* is.
  - A függvényeket *értékekre*, a típusfüggvényeket *típusokra* alkalmazhatjuk.
- Függvényjel és típusfüggvényjel helyzete és kötése az SML-ben
  - Függvényjel: *prefix* vagy *infix*.
  - Típusfüggvényjel: *infix* vagy *postfix*.
  - Az *infix* helyzetű függvényjel és típusfüggvényjel (szokásos nevén operátor, ill. típusoperátor) vagy balra, vagy jobbra köt.
- Infix helyzetben csak a két beépített típusoperátor (`*` és `->`) lehet.
- A `*` balra, a `->` jobbra köt. A `*` erősebben köt, mint a `->`.
- A típusoperátorok erősebben kötnek az összes többi operátornál.

Függvényjel helyzete és kötése az SML-ben

- Tetszőleges kétargumentumú függvényjelet lehet adott preferenciájú (infix helyzetű) operátorként deklarálni az `infix` vagy az `infixr` direktívával.
- Az `infix` balra, az `infixr` jobbra kötő operátort deklarál.
- Egy minősített nevet, vagy egy olyan nevet, amelyet az `op` direktíva előz meg, csak *prefix* helyzetben lehet alkalmazni.
- A `nonfix` direktíva az (infix helyzetű) operátort tartósan prefix helyzetűvé alakítja. (Az `op` direktíva csak átmenetileg teszi prefix helyzetűvé.)
- A *d* 0 és 9 közötti számjegy, az operátor precedenciája (opcionális, alapértelmezés szerinti értéke 0). Nagyobb szám erősebb kötést jelent (éppen fordítva, mint a Prologban!).
- Az $\mathtt{id}_i$ tetszőleges név ($n \geq 1$).

`infix`	$<d>$	$id_1 \ldots id_n$	balra köt	binds to the left
`infixr`	$<d>$	$id_1 \ldots id_n$	jobbra köt	binds to the right
`nonfix`		$id_1 \ldots id_n$	prefix	prefix

A beépített operátorok és precedenciájuk az SML-ben

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

`wordint` = `int`, `word`, `word8`. `num` = `int`, `real`, `word`, `word8`.

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

Prec.	*Operátor*	*Típus*	*Eredmény*	*Kivétel*
7	`*`	`num * num -> num`	szorzat	`Overflow`
	`/`	`real * real -> real`	hányados	`Div`, `Overflow`
	`div`, `mod`	`wordint * wordint -> wordint`	hányados, maradék	`Div`, `Overflow`
	`quot`, `rem`	`int * int -> int`	hányados, maradék	`Div`, `Overflow`
6	`+`, `-`	`num * num -> num`	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	
	`<`, `<=`	`numtxt * numtxt -> bool`	kisebb, kisebb-egyenlő	
	`>`, `>=`	`numtxt * numtxt -> bool`	nagyobb, nagyobb-egyenlő	
3	`:=`	`'a ref * 'a -> unit`	értékadás	
	`o`	`('b -> 'c) * ('a -> 'b) -> ('a -> 'c)`	a két függvény kompozíciója	
0	`before`	`'a * 'b -> 'a`	a bal oldali argumentum	

SML-szintaxis: nemterminális szimbólumok, nyelvtani jelölések

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

Xseq ::=	***X***	egyelemű sorozat	singleton sequence
		üres sorozat	empty sequence
	X_1 , . . . , X_n	sorozat, $n \geq 1$	sequence, $n \geq 1$

- A változatokat prioritásuk csökkenő sorrendjében soroljuk föl.
- A változatokat számozzuk, a példákban utalunk az alkalmazott változatra.
- A függvényjelek és operátorok általában balra kötnek, az eltérést jelezzük.
- Minden ismétlődő konstrukció (pl. a ***klózsorozat)*** a lehető legmesszebb terjeszkedik jobbra. Ezért pl. egy `case`-kifejezést egy másik `case`- vagy `fn`-kifejezésen, valamint egy `fun`-definíción belül zárójelbe kell tenni.

SML-szintaxis: kifejezések és klózsorozatok (egyszerűsítve)

- Kifejezés (***exp:*** expression)

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

- Példák:

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

SML-szintaxis: kifejezések és klózsorozatok (folyt.)

- Infix kifejezés (*infexp:* infix expression)

(6)	*infexp* ::=	*appexp*		
(7)		$infexp_1$ *id* $infexp_2$	infix alkalmazás	infixed application

- Applikatív kifejezés (*appexp:* applicative expression)

(8)	*appexp* ::=	*atexp*		
(9)		*appexp atexp*	(prefix) alkalmazás	(prefixed) application

- **Példák:**

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

SML-szintaxis: kifejezések és klózsorozatok (folyt.)

- Atomi kifejezés (*atexp:* atomic expression)

(10)	*atexp* ::=	*scon*	különleges állandó	special constant
(11)		<`op`> *longvid*	értéknév	value identifier
(12)		`{`<*exprow*>`}`	rekord	record
(13)		`#` *lab*	rekordszelektor	record selector
(14)		`(`exp_1 `,` exp_2`)`	pár	pair
(15)		`()`	nullas	0-tuple
(16)		`[`exp_1`,` ... `,` exp_n`]`	lista, $n \geq 0$	list, $n \geq 0$
(17)		`(`*exp*`)`	kifejezés zárójelben	parenthesized expr.

- **Példák:**

`1.12, #"Z", 0w123` vö. (10)
`Math.pi, false, Math.sin, sin` vö. (11)
`#den {num=1, den=2}` vö. (12), (13), (18)
`(2, 3.5), (), [1, 2, 3]` vö. (14), (15), (16)

SML-szintaxis: kifejezések és klózsorozatok (folyt.)

- Kifejezéssor (*exprow:* expression row)

(18) *exprow* ::= *lab* `=` *exp* <`,` *exprow*>

- Klózsorozat (*match*)

(19) *match* ::= *mrule* <`|` *match*>

- Klóz (*mrule:* match rule)

(20) *mrule* ::= *pat* `=>` *exp*

- **Példák:**

`num=1, den=2` vö. (18)
`00 => 01 | 01 => 11 | 11 => 10 | 10 => 00` vö. (19), (20)

SML-szintaxis: deklarációk és kötések

- Deklaráció (*dec:* declaration)

(20)	*dec* ::=	`val` *tyvarseq valbind*	értékdeklaráció	value declaration
(21)		`fun` *tyvarseq fvalbind*	függvénydeklaráció	function declaration
(22)		`type` *typbind*	típusdeklaráció	type declaration
(23)			üres deklaráció	empty declaration
(24)		dec_1 <`;`> dec_2	deklaráció-sorozat	sequential declaration
(25)		`infix` <*d*> $id_1 \ldots id_n$	infix-direktíva, $n \geq 1$	infix (left) directive
(26)		`infixr` <*d*> $id_1 \ldots id_n$	infixr-direktíva, $n \geq 1$	infix (right) directive
(27)		`nonfix` $id_1 \ldots id_n$	nonfix-direktíva, $n \geq 1$	nonfix directive

- **Példák:**

`val xy = "XY"; fun ++ x y = x ^ y` vö. (20), (21), (24)
`type Rat = {num : int, den : int}` vö. (22)
`infixr 4 ++; fun x ++ y = x ^ y` vö. (21), (26)

SML-szintaxis: deklarációk és kötések (folyt.)

- Értékkötés (***valbind:*** value binding)

(28) ***valbind*** ::=	***pat*** = ***exp*** <and ***valbind***>	értékkötés	value binding
(29)	rec ***valbind***	rekurzív kötés	recursive binding

- Függvényérték-kötés (***fvalbind:*** function value binding)

$$\begin{array}{lll|l} (30) & \textit{fvalbind} ::= & \langle\texttt{op}\rangle\ \textit{var}\ \textit{atpat}_{11} \ldots \textit{atpat}_{1n}\ \langle :\textit{ty}\rangle = \textit{exp}_1 & m,n \ge 1 \\ & | & \langle\texttt{op}\rangle\ \textit{var}\ \textit{atpat}_{21} \ldots \textit{atpat}_{2n}\ \langle :\textit{ty}\rangle = \textit{exp}_2 & \\ & | & \ldots & \\ & | & \langle\texttt{op}\rangle\ \textit{var}\ \textit{atpat}_{m1} \ldots \textit{atpat}_{mn}\ \langle :\textit{ty}\rangle = \textit{exp}_m & \\ & & \langle\texttt{and}\ \textit{fvalbind}\rangle & \end{array}$$

Megjegyzés: Ha *var* infix, akkor egy *fvalbind* definícióban vagy infix helyzetben kell használni, vagy elé kell írni az op direktívát; azaz a definícióban a bal oldalon (*atpat var atpat'*) vagy op *var* (*atpat*, *atpat'*) írható. A zárójelek elhagyhatók, ha *atpat'* után közvetlenül :*ty* vagy = áll.

- Példák:

```
val even = fn 0 => true | x => not(odd(x-1))
and odd = fn 0 => false | y => not(even(y-1));  vö. (28)
fun (f o g) x = g(f x);                          vö. (30)
```


SML-szintaxis: típuskifejezések

- Típus (***ty:*** type)

(31) ***ty*** ::=	***tyvar***	típusváltozó	type variable
(32)	***tycon***	típuskonstruktor	type constructor
(33)	{ <***tyrow***> }	rekordtípus-kifejezés	record type expression
(34)	ty_1 * ty_2	pár-típus	pair type
(35)	ty_1 -> ty_2	függvénytípus-kifejezés	function type expression
(36)	(***ty***)	típus zárójelben	parenthesized type

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

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

- Példák:

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


SML-szintaxis: minták

- Atomi minta (***atpat:*** atomic pattern)

(38) ***atpat*** ::=	_	mindenesjel	wildcard
(39)	***scon***	különleges állandó	special constant
(40)	<op> ***longvid***	értéknév	value identifier
(41)	{ <***patrow***> }	rekord	record
(42)	(pat_1 * pat_2)	pár	pair
(43)	(), {}	nullas	0-tuple
(44)	[pat_1, ..., pat_n]	lista, $n \ge 0$	list, $n \ge 0$
(45)	(***pat***)	minta zárójelben	parenthesized pattern

- Példák:

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


SML-szintaxis: minták (folyt.)

- Mintasor (***patrow:*** pattern row)

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

- Példák:

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

SML-szintaxis: minták (folyt.)

- Minta (*pat:* pattern)

(49)	*pat* ::= *atpat*	atomi minta	atomic pattern
(50)	<op> *longvid atpat*	értékkonstrukció	value construction
(51)	pat_1 *vid* pat_2	infix értékkonstrukció	infixed value constr.
(52)	*pat* : *ty*	minta típusmegkötéssel	typed pattern
(53)	<op> *var* <: *ty*> as *pat*	réteges minta	layered pattern

- Példa:

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


SML-szintaxis: szintaktikai korlátozások

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

RACIONÁLIS SZÁMOK

Példa: racionális számok

- A racionális számokat *rekordként* ábrázoljuk; az új (gyenge) típus neve `rat`.

```
type rat = {num : int, den : int};
```

- Nevet adunk néhány állandónak.

```
val ratZero = {num = 0, den = 1}; val ratOne   = {num = 1, den = 1};
val ratHalf = {num = 1, den = 2}; val ratThird = {num = 1, den = 3};
```

- A `rat` típusú számokat *normalizált* alakban tároljuk, különben pl. $\frac{1}{2}$ és $\frac{2}{4}$ nem lenne egyenlő. A normalizáláshoz szükségünk van a számláló és a nevező legnagyobb közös osztójára (`gcd`). A közös osztó egyik fontos tulajdonsága, hogy $d|n$ és $d|m \Rightarrow d|n \bmod m$.

```
(* gcd : int -> int -> int
   gcd n m = n és m legnagyobb közös osztója
*)
fun gcd n 0 = abs n
  | gcd n m = gcd (abs m) (abs(n mod m));
```

Példa: racionális számok (folyt.)

- gcd ún. ***részlegesen alkalmazható*** függvény. Ha összes argumentumánál kevesebbre alkalmazzuk, ***függvényértéket*** ad eredményül.
- Sajnos, a `normalize` függvényben `n` és `m` legnagyobb közös osztóját kétszer is kiszámoljuk: később látni fogjuk, hogyan javíthatunk a hatékonyságán.

```
(* normalize : rat -> rat
   normalize r = r normalizált alakban
*)
fun normalize {num = n, den = 0} = raise Domain
  | normalize {num = n, den = d} = {num = n div (gcd n d), den = d div (gcd n d)}
```

- Két egészből ***konstruktorfüggvénnyel*** (`toRat`) érdemes létrehozni a racionális számot, különben a normalizált tárolás követelménye sérülhet.

```
(* toRat : int -> int -> rat
   toRat n d = n nevezőjű és d számlálójú racionális szám, normalizált alakban
*)
fun toRat n d = normalize {num = n, den = d};
```


PÁR ÉS TÍPUSA

Kitérő: pár és típusa

Mi a +-szal jelölt összeadás-művelet típusa SML-ben?

- A + kétoperandusú művelet, argumentuma egy ***pár,*** pl. `3 + 4`.
- `+ : int * int -> int` vagy `+ : real * real -> real`, ahol `*` egy újabb típusművelet, a ***keresztszorzat*** (***Descartes-szorzat***) jele.
- A + műveleti jel (függvényjel) ***többszörös terhelésű.***
- + prefix helyzetben is használható, ha eléírjuk az `op` kulcsszót, pl. `op+(3, 4)`. Ilyenkor az operandusait ***párként, zárójelbe zárva*** kell megadni.

A beépített infix típusoperátorok precedenciája és kötése

- Két beépített infix típusoperátor van az SML-ben: `->` (leképzés) és `*` (keresztszorzat). A `*` precedenciája a nagyobb. A `*` balra, a `->` jobbra köt.
- Példák: `'a * 'b * 'c` = `('a * 'b) * 'c`
  `'a -> 'b -> 'c` = `'a -> ('b - 'c)`
  `'a * 'b -> 'c` = `('a * 'b) -> 'c`

RACIONÁLIS SZÁMOK

Példa: racionális számok – a négy alapművelet

```
(* **, //, ++, -- : rat * rat -> rat
   r1 ** r2 = az r1 és r2 racionális számok szorzata
   r1 // r2 = az r1 és r2 racionális számok hányadosa
   r1 ++ r2 = az r1 és r2 racionális számok összege
   r1 -- r2 = az r1 és r2 racionális számok különbsége
*)
infix 7 ** //; infix 6 ++ --;

fun (r1 : rat) ** (r2 : rat) = toRat (#num r1 * #num r2) (#den r1 * #den r2);

fun (r1 : rat) // (r2 : rat) = toRat (#num r1 * #den r2) (#num r2 * #den r1);

fun {num= n1, den= d1} ++ {num= n2, den= d2} = toRat (n1*d2 + n2*d1) (d1*d2);

fun {num= n1, den= d1} -- {num= n2, den= d2} = toRat (n1*d2 - n2*d1) (d1*d2);
```


Példa: racionális számok – relációs műveletek

- Az `=` és a `<>` relációt ***készen kapjuk:*** két összetett érték strukturálisan összehasonlítható, ha az elemeiken az egyenlőségvizsgálat elvégezhető.

```
(* <<, >>, <<=, >>= : rat * rat -> bool
   r1 << r2 = igaz, ha r1 kisebb r2-nél
   r1 >> r2 = igaz, ha r1 nagyobb r2-nél
   r1 <<= r2 = igaz, ha r1 nem nagyobb r2-nél
   r1 >>= r2 = igaz, ha r2 nem nagyobb r1-nél
*)
infix 4 << >> <<= >>=;

fun (r1 : rat) << (r2 : rat) = #num r1 * #den r2 < #num r2 * #den r1;

fun (r1 : rat) >> (r2 : rat) = #num r1 * #den r2 > #num r2 * #den r1;

fun r1 <<= r2 = not(r1 >> r2);    fun r1 >>= r2 = not(r1 << r2);
```


POLIMORFIZMUS

Polimorfizmus

- Nézzük az identitásfüggvényt: `fun id x = x`.
- Mi az `x` típusa? Bármilyen típusú lehet: 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***.

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, annyifélét; ez az ***ad-hoc*** vagy ***többszörös terheléses*** polimorfizmus.
- A polimorfizmus harmadik változata az ***öröklődéses polimorfizmus*** (vö. objektum-orientált programozás).

KÉT FÜGGVÉNY KOMPOZÍCIÓJA

Kitérő: két függvény kompozíciója

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

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

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

- Az `o` típusa `? * ? -> ?` szerkezetű. Mit írjunk a `?`-ek helyébe? Vezessük le!
  - A függvénydefiníció jobb oldalán álló kifejezés elemzésével kezdjük.

    `x : 'a      g : 'a -> 'b      f : 'b -> 'c`
  - A függvénydefinícióban az egyenlőségjel (=) bal és jobb oldalán álló kifejezéseknek azonos értéket kell eredményül adniuk, ezért `f o g` és `f` eredményének azonos a típusa (azaz `'c`).

    `(f o g) : 'a -> 'c      o : ('b -> 'c) * ('a -> 'b) -> ('a -> 'c)`
- **Példa:** `round : real -> int,  chr : int -> char`

  `chr o round : real -> char`

RACIONÁLIS SZÁMOK

Példa: racionális számok (folyt.)

- A racionális számokon értelmezett `<<=` és `>>=` másképpen:

```
val op<<= = not o op>>;    val op>>= = not o op<<;
```

- Egy racionális számot füzérré alakítás után írunk ki a képernyőre.

```
(* toString : rat -> string
   toString r = az r racionális szám füzérként (számláló/nevező alakban,
                ha a nevező = 1, egyébként egészként)
*)
fun toString {num, den = 1} = Int.toString num
  | toString {num, den}     = Int.toString num ^ "/" ^ Int.toString den
```

- Példák `rat` típusú értékek használatára

```
normalize (toRat 15 3);       toString(toRat 2 3 ** toRat 5 4);
normalize (toRat 15 ~3);      toString(toRat 2 3 // toRat 5 3);
normalize (toRat ~15 3);      toString(toRat 1 4 ++ toRat 3 10);
normalize (toRat ~15 ~3);     toString(toRat 3 10 -- toRat 1 4);
```

Példa: racionális számok (folyt.)

- Példák `rat` típusú értékek használatára (folyt.)

```
toRat 2 3 << toRat 5 4;         toRat 2 3  >> toRat 5 3;
toRat 1 4 << toRat 3 10;        toRat 3 10 >> toRat 1 4;

infix 8 /-/;
fun n /-/ d = toRat n d;

toString(2/-/3 **  5/-/4);        2/-/3  <<   5/-/4;    1/-/4  <<   3/-/10;
toString(2/-/3 //  5/-/3);        2/-/3  <<   2/-/3;    3/-/10 >>   1/-/4;
toString(1/-/4 ++  3/-/10);       2/-/3  <<=  2/-/3;
toString(3/-/10 -- 1/-/4);        2/-/3  >>   5/-/3;    3/-/10 >>=  3/-/10;
```

- Példák `gcd` részleges alkalmazására

```
(* gcd120 : int -> int                                   gcd120 45;
  gcd m = m legnagyobb közös osztója 120-szal            gcd120 48;
*)                                                       gcd120 ~96;
val gcd120 = gcd 120;                                    gcd120 630;
```


POLIMORFIZMUS

Paraméteres polimorfizmus

- Az identitásfüggvény és típusa: `fun id x = x`, `id : 'a -> 'a`.
  Az `mosml` válasza: `val 'a id = fn : 'a -> 'a`. Az `id` ***politípusú*** név.
- Az `=` és a `<>` műveletet ***készen kapjuk*** a legtöbb típusra (vö. `rat`).
  A típusuk: `=`, `<>` : `''a * ''a -> bool`. A `''` ***egyenlőségi típust*** jelöl, az ilyen típusú értékeken az egyenlőségvizsgálat elvégezhető.
- Az egyenlőségvizsgálat ***korlátozottan*** polimorf: nem minden értékre végezhető el. Pl. egy f és egy g függvény akkor és csak akkor egyenlő, ha $\forall x \cdot fx = gx$. Ezt ***általánosságban*** lehetetlen eldönteni.
- Mi a `<`, `>`, `<=`, `>=` típusa?
  Pl. az `op<=`-re az `mosml` válasza: `val it = fn : int * int -> bool`.
  E négy művelet ***ad-hoc*** módon polimorf, a nevek ***többszörösen terhelhetők***, alapértelmezés szerint `int` típusú értékekre alkalmazhatók.
- Az `=` részlegesen alkalmazható változata legyen: `fun eq x y = x = y`.
  Típusa: `eq : ''a -> ''a -> bool`.

Példák eq használatára (`''a eq : ''a -> ''a -> bool`)

```
A kifejezés               Az mosml válasza
eq 3 3;                   > val it = true : bool
eq "id" "idn";            > val it = false : bool
eq id id;                 ! Toplevel input:
                          ! eq id id;
                          !    ^^
                          ! Type clash: expression of type
                          !   'e -> 'e
                          ! cannot have equality type ''f
eq 3;                     > val it = fn : int -> bool
eq "id";                  > val it = fn : string -> bool
val eqStr_id = eq "id";   > val eqStr_id = fn : string -> bool
```

- Az `id` függvény, típusa (`'e -> 'e`) nem egyenlőségi típus!
- Az `eq "id"` függvényértéket ad eredményül, ezért az `eqStr_id` függvényt jelöl. Olyan függvényt, amely az `"id"` füzérre alkalmazva `true`, minden más esetben `false` értéket ad eredményül.

Példák `id` használatára (`'a id : 'a -> 'a`)

A kifejezés	Az `mosml` válasza
`id 3;`	`> val it = 3 : int`
`id "id";`	`> val it = "id" : string`
`id round;`	`> val it = fn : real -> int`

`id id;`	`! Warning: Value polymorphism:`
	`! Free type variable(s) at top level in value identifier it`
	`> val it = fn : 'b -> 'b`

`id id 6.9;`	`> val it = 6.9 : real`
`fn x => id id x;`	`> val 'b it = fn : 'b -> 'b`

- Az SML ún. ***érték-polimorfizmust*** használ.
  - Az SML a típusváltozókat, ahol csak tudja, általánosítja (pl. `fn x => id id x`).
  - Az `mosml` a nem általánosítható típusváltozókat meghagyja ***szabad típusváltozónak*** (pl. `id id`).

Érték-polimorfizmus

- Tekintsük a `val x = e` deklarációt.
  - Az SML az `x` típusában előforduló szabad típusváltozókat akkor általánosítja, ha `e` ún. ***nem-expanzív*** kifejezés.
  - Ez csupán ***szintaktikai*** követelmény: egy kifejezés ***nem-expanzív***, ha megfelel a ***nexp*** szintaktikai kategóriát leíró nyelvtani szabályoknak.

Nem-expanzív kifejezés (egyszerűsítve)

- Nem-expanzív kifejezés (***nexp:*** non-expansive expression)

nexp ::=	***scon***	különleges állandó	special constant
	longvid	(esetleg minősített) értéknév	(possibly qualified) value identifier
	{ <***nexprow***> }	nem-expanzív elemekből álló rekord	record of non-expansive expressions
	(***nexp***)	nem-expanzív kifejezés zárójelben	parenthesized non-expansive expression
	nexp : ***ty***	nem-expanzív kifejezés típusmegkötéssel	typed non-expansive expression
	`fn` ***match***	függvénykifejezés	function expression

- Nem-expanzív kifejezéssor (***nexprow:*** non-expansive expression row)

nexprow ::= ***lab*** = ***nexp*** <, ***nexprow***>

Példák nem-expanzív és expanzív kifejezésekre

- Egy nem-expanzív kifejezés egyszerűen: érték (azaz tovább nem egyszerűsíthető, ún. ***kanonikus*** kifejezés).

```
  val x = length;
> val 'a x = fn : 'a list -> int
```

`length` egy név, ezért nem-expanzív. Az `x` típusát leíró `'a list -> int` típuskifejezésben az `'a` szabad típusváltozó általánosítható, ezt tükrözi a definíció bal oldalán az `'a x`.

- Az `(fn f => f) length` kifejezés értéke is `length`, de expanzív, mert nem vezethető le a fenti nyelvtani szabályok alapján.

```
  val x = (fn f => f) length;
! Warning: Value polymorphism:
! Free type variable(s) at top level in value identifier x
> val x = fn : 'a list -> int
```

Példák nem-expanzív és expanzív kifejezésekre (folyt.)

Az 'a típusváltozót az SML nem általánosítja. Az `mosml` meghagyja szabad típusváltozónak, és majd csak az `x` *első alkalmazásakor* köti le.

```
  x ["abc", "def"];
! Warning: the free type variable 'a has been instantiated to string
> val it = 2 : int
  x;
> val it = fn : string list -> int
```

- Ha már az 'a-t lekötöttük, más típushoz nem köthető; `x` nem politípusú név.

```
  x [123, 456, 789];
! Toplevel input:
!   x [123, 456, 789];
!      ^^^
! Type clash: expression of type
!   int
! cannot have type
!   string
```


η-expanzió

- A típusváltozó általánosítása mindig kikényszeríthető a deklaráció jobb oldalának *η-expanziójával.*

  Az η-expanzió az `e` kifejezést a nem-expanzív `fn y => e y` kifejezéssel helyettesíti.

```
  val x1 = fn y => ((fn f => f) length) y;
> val 'b x1 = fn : 'b list -> int
```

  A fenti deklarációban a külső zárójelpár el is hagyható:

```
  val x1 = fn y => (fn f => f) length y;
```

- Az `x1` politípusú név.

```
  x1 ["abc", "def"];
> val it = 2 : int
  x1 [123, 456, 789];
> val it = 3 : int
```


LISTÁK

Lista: definíciók, konstruktorok

- 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` *konstruktorállandó.* `nil` típusa `'a list`.
  - A `::` *konstruktoroperátor* új listát hoz létre egy elemből és egy (esetleg üres) listából (infix, 5-ös precedenciájú, jobbra köt, típusa `'a * 'a list -> 'a list`).
  - A `nil` helyett általában a `[]` jelet használjuk (szintaktikai édesítőszer).
  - A `::`-ot négyespontnak vagy *cons*-nak olvassuk (vö. *constructor*, ami a függvény hagyományos neve a λ-kalkulusban és egyes funkcionális nyelvekben).

Lista: jelölések, minták

- Példák
  - Lista létrehozása konstruktorokkal

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

  - Szintaktikus édesítőszer lista jelölésére

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

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

SML	Prolog		SML	Prolog	
`[]`	`[]`	azonos	`(x::xs)`	`[X|Xs]`	különböző
`[1,2,3]`	`[1,2,3]`	azonos	`(x::y::z::zs)`	`[X,Y,Z|Zs]`	különböző

- Minták

  A `[]` és a `nil` állandók, a `::` operátor, valamint a `[x1, x2, ..., xn]` listajelölés mintában is alkalmazhatók.

Lista: fej (hd), farok (tl)

- A nem-üres lista első eleme a lista *feje*.

```
(* hd : 'a list -> 'a *)
fun hd (x :: _) = x;
```

- A nem-üres lista első utáni elemeiből áll a lista *farka*.

```
(* tl : 'a list -> 'a list *)
fun tl (_ :: xs) = xs;
```

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

  Fontos: a parciális függvények nem tévesztendők össze a parciálisan (azaz részlegesen) alkalmazható függvényekkel!

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

- Egy lista hosszát adja eredményül a már látott `length` függvény (l. `List.length`).

```
(* length : 'a list -> int *)
fun length (_ :: xs) = 1 + length xs
  | length []        = 0;
```

- Egy egész számokból álló lista elemeinek összegét adja eredményül `isum`.

```
(* isum : int list -> int *)
fun isum (x :: xs) = x + isum xs
  | isum []        = 0;
```

- Egy valós számokból álló lista elemeinek szorzatát adja eredményül `rprod`.

```
(* rprod : real list -> real *)
fun rprod (x :: xs) = x * rprod xs
  | rprod []        = 1.0;
```


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`

Lista: adott transzformáció alkalmazása minden elemre (`map`)

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

  `map Math.sqrt [1.0, 4.0, 9.0, 16.0] = [1.0, 2.0, 3.0, 4.0]`
- Általában: `map f` $[x_1, x_2, \ldots, x_n]$ = $[f\ x_1, f\ x_2, \ldots, f\ x_n]$
- A függvény típusa: `map : ('a -> 'b) -> 'a list -> 'b list`
- Egy-egy klózt írunk a triviális és a nem-triviális eset lefedésére
  - `map f [] = []`
  - `map f (x :: xs) = f x :: map f xs`

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

- `map` típusa, ha egyargumentumú függvénynek tekintjük (ui. `->` jobbra köt):

  `map : ('a -> 'b) -> ('a list -> 'b list).`

  Azaz ha `map`-et egy `'a -> 'b` típusú függvényre alkalmazzuk, akkor olyan függvényt ad eredményül, amelyet egy `'a list` típusú listára alkalmazva egy `'b list` típusú listát kapunk.

PROGRAMHELYESSÉG

A program helyességének igazolása a `map` példáján

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

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

  - 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 farok transzformálásával kapott lista elé fűzve valóban a várt eredményt kapjuk.
  - A kiértekelés véges számú lépésben befejeződik, mert a lista véges, a `map` függvényt a ***rekurzív ágban*** minden lépésben egyre rövidülő listára alkalmazzuk, és gondoskodtunk a rekurzió leállításáról (a ***triviális eset*** kezeléséről, ui. van nem rekurzív ág).

LISTÁK

Lista: adott predikátumot kielégítő elemek kiválogatása (filter)

- Kitérő: `explode`, `implode`
  - `explode : string -> char list`, pl. `explode "abc" = [#"a", #"b", #"c"]`
  - `implode : char list -> string`, pl. `implode [#"a", #"b", #"c"] = "abc"`
- Példa: gyűjtsük ki a kisbetűket egy karakterlistából!

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

- Általában: ha `p` x_1 `= true`, `p` x_2 `= false`, `p` x_3 `= true`, ..., `p` x_n `= true`, akkor `filter p` $[x_1, x_2, x_3, \ldots, x_n]$ `=` $[x_1, x_3, \ldots, x_n]$.
- A függvény típusa: `filter : ('a -> bool) -> 'a list -> 'a list`
- Egy-egy klózt írunk a triviális és a nem-triviális eset lefedésére
  - `filter p [] = []`
  - `filter p (x :: xs) = if p x then x :: filter p xs else filter p xs`

Lista: filter (folyt.)

- Ezzel `filter` definíciója

```
fun filter p (x :: xs) =
      if p x then x :: filter p xs else filter p xs
  | filter _ [] = [];
```

- `filter` típusa, ha egyargumentumú függvénynek tekintjük (`->` jobbra köt!):

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

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

Lista redukciója kétoperandusú művelettel (foldr, foldl)

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

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

- Jelöljön $\oplus$ tetszőleges kétoperandusú infix operátort. Akkor

  `foldr op`$\oplus$ `e` $[x_1, x_2, \ldots, x_n]$ `=` $(x_1 \oplus (x_2 \oplus \ldots \oplus (x_n \oplus e) \ldots))$

  `foldr op`$\oplus$ `e [] = e`

  `foldl op`$\oplus$ `e` $[x_1, x_2, \ldots, x_n]$ `=` $(x_n \oplus \ldots \oplus (x_2 \oplus (x_1 \oplus e)) \ldots)$

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

Példák foldr és foldl alkalmazására

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

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

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

Példák foldr és foldl alkalmazására (folyt.)

- 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 [x_1, x_2, x_3] = (x_1 :: (x_2 :: (x_3 :: ys)))

  foldl op:: ys [x_1, x_2, x_3] = (x_3 :: (x_2 :: (x_1 :: ys)))

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

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

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

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

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

Lista: foldr és foldl definíciója

- foldr op$\oplus$ e [x_1, x_2, ..., x_n] = (x_1 $\oplus$ (x_2 $\oplus$ ... $\oplus$ (x_n $\oplus$ e) ...))

  foldr 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;
```

- foldl op$\oplus$ e [x_1, x_2, ..., x_n] = (x_n $\oplus$ ... $\oplus$ (x_2 $\oplus$ (x_1 $\oplus$ e)) ...)

  foldl 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;
```


Lista redukciója bal oldali egységelemű függvénnyel (foldL)

- A kivonás művelete balra köt: $x_1 - x_2 - x_3 - x_4 = ((x_1 - x_2) - x_3) - x_4$.
- Nem feleltethető meg sem foldr-nek, sem foldl-nek.

  foldr op$\oplus$ e [x_1, x_2, ..., x_n] = (x_1 $\oplus$ (x_2 $\oplus$ ... $\oplus$ (x_n $\oplus$ e) ...))

  foldl op$\oplus$ e [x_1, x_2, ..., x_n] = (x_n $\oplus$ ... $\oplus$ (x_2 $\oplus$ (x_1 $\oplus$ e)) ...)

- Nevezzük foldL-nek a listában *balról jobbra* haladó, alábbi specifikációjú függvényt. Vegyük észre, hogy $\oplus$ bal oldali egységelemet vár.

  foldL op$\oplus$ e [x_1, x_2, ..., x_n] = (... ((e $\oplus$ x_1) $\oplus$ x_2) $\oplus$ ... $\oplus$ x_n)

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

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


Példák listaelemek különbségének és hányadosának képzésére

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

```
foldL op- 20 [] = 20;                  foldL (op div) 180  [] = 180;
foldL op- 20 [5, 6, 7] =               foldL (op div) 180  [2, 3, 5] =
        (((20 - 5) - 6) - 7);                  (((180 div 2) div 3) div 5);
```

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

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

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

Listaelemek különbsége és hányadosa `foldl`-lel és `foldr`-rel

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

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

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

- `foldr` és `foldl` típusa, ha egyparaméteres függvénynek tekintjük őket (a `->` jobbra köt!):

  `foldr, foldl : ('a * 'b -> 'b) -> ( 'b -> 'a list -> 'b)`

  Azaz ha `foldr`-t vagy `foldl`-t egy `'a -> * 'b -> 'b` típusú függvényre alkalmazzuk, akkor olyan függvényt ad eredményül, amelyet egy `'b` típusú egységelemre és egy `'a list` típusú listára alkalmazva `'b` típusú (redukált) értéket kapunk.

KIFEJEZÉSEK KIÉRTÉKELÉSE

Mohó kiértékelés: faktoriális kiszámítása naív rekurzióval

- A faktoriális matematikai definíciója és megvalósítása SML-ben

  $\text{fac}\ 0 = 1 \qquad \text{fac}\ n = n * \text{fac}\ (n-1)$

```
(* fac : int -> int       (-- fontos a klózok sorrendje! --)
   fac n = n!
   PRE n >= 0 *)
fun fac 0 = 1 | fac n = n * fac(n-1);
```

- `fac` mohó kiértékelése $n = 4$ esetén (egyes triviális lépéseket elhagyunk).

  `fac 4` → `4 * fac (4-1)` → `4 * fac 3` → `4 * (3 * fac (3-1))` →
  → `4 * (3 * fac (2))` → ··· → `4 * (3 * (2 * (1 * 1)))` → ··· → `24`

- A rekurzív kiértékelés követi a matematikai definíciót.
- Rontja a hatékonyságot, hogy a rekurzív végrehajtás során minden részeredményt a veremben tárolni kell.
- Ha a szorzás asszociativitását kihasználjuk, nem kell tárolni az összes tényezőt, csak az aktuális részeredményt.

Faktoriális kiszámítása jobbrekurzióval

- Először egy *akkumulátort* (gyűjtőargumentumot) használó *segédfüggvényt* definiálunk. Vegyük észre, hogy a rekurzív hívás *jobbrekurzív:* eredménye közvetlenül, további műveletek elvégzése nélkül adja a végeredményt.

```
(* faci : int -> int -> int    (-- fontos a klózok sorrendje! --)
   faci n p = p * n!           (-- p az akkumulátor --)
*)
fun faci 0 p = p
  | faci n p = faci (n-1) (n*p);
```

- `faci`-t felhasználjuk az egyparaméteres `fac` függvény definiálására. Az akkumulátornak alkalmas *kezdőértéket* adunk.

```
(* fac : int -> int
   fac n = n!
   PRE n >= 0
*)
fun fac n = faci n 1;
```

Faktoriális kiszámítása jobbrekurzióval (folyt.)

- `fac` nem rekurzív, ezért csak `faci` kiértékelését vizsgáljuk (egyes triviális lépéseket összevonunk).

  A függvény: `fun faci 0 p = p | faci n p = faci (n-1) (n*p)`

  `faci 4 1` → `faci (4-1) (4*1)` → `faci 3 4` → `faci (3-1) (3*4)` →
  → `faci 2 12` → ··· → `faci 0 24` → `24`
- Kiértékelés közben a `p` ***akkumulátor*** gyűjti a részeredményt, ezért `faci` tárigénye állandó.
- A kiértékelés ***iteratív***.
- A jó fordítóprogram felismeri a jobbrekurziót, és hatékony tárgykódot állít elő: az argumentumokat frissíthető lokális változókban tárolja, a rekurziót iterációval helyettesíti.
- A jobbrekurziót ***terminális rekurziónak*** is nevezik (angolul: ***tail*** vagy ***terminal*** recursion).
- `foldl` jobbrekurzív, `e` argumentuma akkumulátorként viselkedik.

Lokális kifejezés

- ***Lokális kifejezést*** használunk, ha ismétlődő részkifejezéseket ***csak egyszer*** akarunk kiszámítani, vagy akkor, ha bizonyos értékeket a program többi része elől ***el akarunk rejteni.***
- Szintaxisa: `let` *d* `in` *e* `end`, ahol
  - *d* nemüres deklarációsorozat,
  - *e* nemüres kifejezés.
- Példa:

```
fun fac n =
  let
      fun faci 0 p = p
        | faci n p = faci (n-1) (n*p)
  in
      faci n 1
  end
```


Lokális deklaráció

- ***Lokális deklarációt*** használunk olyan értékek bevezetésére, amelyeket a program többi része elől ***el akarunk rejteni.***
- Szintaxisa: `local` *d1* `in` *d2* `end`, ahol
  - *d1* és *d2* nemüres deklarációsorozatok.
- Példa:

```
local
    fun faci 0 p = p
      | faci n p = faci (n-1) (n*p)
in
    fun fac n = faci n 1
end
```


LOGIKAI MŰVELETEK

Logikai műveletek

- Típusnév: `bool`, adatkonstruktorok: `false`, `true`, beépített függvény: `not`.
- ***Lusta kiértékelésű*** beépített operátorok
  - Három argumentumú: `if b then e1 else e2`.
    Nem értékeli ki az `e2`-t, ha a `b` igaz, ill. az `e1`-et, ha a `b` hamis.
  - Két argumentumúak:
    `e1 andalso e2` : nem értékeli ki az `e2`-t, ha az `e1` hamis.
    `e1 orelse e2` : nem értékeli ki az `e2`-t, ha az `e1` igaz.
- Mind a három csupán szintaktikai édesítőszer!
  - `if b then e1 else e2` ≡ `(fn true => e1 | false => e2) b`
  - `e1 andalso e2` ≡ `(fn true => e2 | false => false) e1`
  - `e1 orelse e2` ≡ `(fn true => true | false => e2) e1`
  - `fun ifThenElse b = (fn true => e1 | false => e2) b; ifThenElse true;`
- Tipikus hiba: `if` *exp* `then true else false` !!!

Logikai műveletek (folyt.)

- Nyilvánvaló: `andalso` és `orelse` kifejezhető `if-then-else`-szel is.
  - `if e1 then e2 else false` ≡ `e1 andalso e2`
  - `if e1 then true else e2` ≡ `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:

```
(* && (a, b) = a /\ b
   && : bool * bool -> bool
*)
fun op&& (a, b) = a andalso b;
infix 2 &&;
```

```
(* || (a, b) = a \/ b
   || : bool * bool -> bool
*)
fun op|| (a, b) = a orelse b;
infix 1 ||;
```


LISTÁK

Listák összefűzése és megfordítása

- Listák összefűzése és megfordítása beépített függvényekkel: `@`, `rev` és `revAppend` (`List` könyvtár).
  - `@` a `fun append (xs, ys) = foldr op:: ys xs` beépített megfelelője: infix, 5-ös precedenciájú, jobbra köt, típusa `'a list * 'a list -> 'a list`.
  - `revAppend` a `fun revApp (xs, ys) = foldl op:: ys xs` beépített megfelelője: prefix, típusa `'a list * 'a list -> 'a list`.
  - `rev` a `fun rev xs = foldl op:: [] xs` beépített megfelelője: prefix, típusa `'a list -> 'a list` (vö. `revApp`).
- Az $[m, n)$ tartományba eső egészek listája: a kézenfekvő megoldás

```
(* upto m n = az [m, n) tartományba eső egészek listája
   upto : int -> int -> int list *)
fun upto m n =  if m < n then m :: upto (m+1) n else [];
```

Listák összefűzése és megfordítása

- Az $[m, n)$ tartományba eső egészek listája: jobbrekurzív megoldás

```
fun upto m n =
      let (* az up számára az n állandó érték,
             ezért nem kell argumentumként átadni *)
          fun up zs m = if m < n then up (m::zs) (m+1) else rev zs
      in  up [] m
      end;
```

- Az $[m, n)$ tartományba eső egészek listája: hatékony jobbrekurzív megoldás

```
fun upto m n =
      let (* hátulról visszafelé haladva építjük föl a listát,
             ezért a végén nem kell megfordítani *)
          fun up zs n = if m < n  then up (n-1::zs) (n-1) else zs
      in up [] n
      end;
```


Lista legnagyobb elemének megkeresése

- Egy egészlista legnagyobb elemének kiválasztásához szükségünk van az `Int.max` függvényre.
  - Üres listának nincs legnagyobb eleme,
  - egyelemű listában az egyetlen elem a legnagyobb,
  - legalább kételemű lista legnagyobb elemét úgy kapjuk, hogy az első elem és a maradéklista elemeinek legnagyobbika közül kiválasztjuk a legnagyobbat.

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

- `max` egy változata egészekre

```
fun max (n, m) = if n > m then n else m
```


Lista legnagyobb elemének megkeresése (folyt.)

- Hogyan tehető polimorffá a `maxl` függvényt? Magasabbrendű, ún. `generikus` függvényként definiáljuk: *argumentumként* kapja azt a többszörösen terhelhető függvényt, amely két érték közül a nagyobbikat kiválasztja.

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

- `max` mindig ugyanaz, mégis újra és újra átadjuk argumentumként a rekurzív ágban. Javítja a hatékonyságot, ha *lokális kifejezést* használunk. (Lokális deklaráció használata most nem segítene. Miért nem?)

```
fun maxl max ns = let fun mxl [n] = n
                        | mxl (n::ns) = max(n, mxl ns)
                        | mxl [] = raise Empty
                  in mxl ns end;
```


Lista (folyt.)

- Változatok `max`-ra

```
(* charMax : char * char -> char *)
fun charMax (n, m) = if ord n > ord m then n else m;

(* pairMax : ((int * real) * (int * real)) -> (int * real)
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;
```

- `concat xss` = az `xss`-beli listákat egy listába fűzi. Könyvtári változata: `List.concat`.

```
(*  concat : 'a list list -> 'a list *)
fun concat xss = foldr op@ [] xss;
```

- `ListPair.zip` két lista páronkénti elemeiből álló párok listáját, `ListPair.unzip` párok listájából két listát ad eredményül.

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

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

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

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

- Könyvtári változatuk, List.take és List.drop $i < 0$ vagy $i >$ length xs esetén Subscript kivételt jelez.

Halmazműveletek

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

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

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

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

newMem, ha a sorrendtől eltekintünk, halmazt hoz létre.

Halmazműveletek (folyt.)

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

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

- Szerencsésebb a halmazokat a megszokott halmazműveletekkel kezelni. Öt halmazműveletet definiálunk:
  - unió (union, $S \cup T$),
  - metszet (inter, $S \cap T$),
  - részhalmaza-e (isSubset, $T \subseteq S$),
  - egyenlők-e (isSetEq, $S = T$),
  - hatványhalmaz (powerset, pS).

Halmazműveletek (folyt.)

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

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

- Két halmaz metszete

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

Halmazműveletek (folyt.)

- Részhalmaza-e egy halmaz egy másiknak?

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

- Két halmaz egyenlősége

  A listák egyenlőségvizsgálata beépített művelet az SML-ben. Halmazokra mégsem használható, mert pl. `[3, 4]` és `[4, 3, 4]` listaként ugyan különböznek, de halmazként egyenlők.

```
(* isSetEq(xs, ys) = az xs és ys elemeiből álló halmazok egyenlők-e
   isSetEq : ''a list * ''a list -> bool *)
fun isSetEq (xs, ys) = (xs isSubset ys) andalso (ys isSubset xs);
```


Halmazműveletek (folyt.)

- Halmaz hatványhalmaza

  A hatványhalmaz egy halmaz ***összes*** részhalmazának a halmaza, az eredeti halmazt és az üres halmazt is beleértve.

  Jelöljük S-sel az eredeti halmazt. S hatványhalmazát úgy állíthatjuk elő, hogy S-ből kiveszünk egy x elemet, és azután ***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.

  A `pws` függvényben a `base` argumentum gyűjti a hatványhalmaz elemeit; kezdetben üresnek kell lennie.

```
(* pws(xs, base) = az xs halmaz hatványhalmazának és
                   a base halmaznak az uniója
   pws : 'a list * 'a list -> 'a list list *)
fun pws (x::xs, base) = pws(xs, base) @ pws(xs, x::base)
  | pws ([], base) = [base];
```


Halmazműveletek (folyt.)

- Halmaz hatványhalmaza (folyt.)

  A `pws(xs, base) @ pws(xs, x::base)` kifejezésben `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)` ugyancsak rekurzív módon `base`-ben gyűjti az `x` elemeket, vagyis előállítja az összes olyan halmazt, amelyben `x` benne van.

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