

## 1. Lista elemeinek szorzata és összege (ismétlés)

Az üres lista nemlétező elemeinek szorzata legyen 1, a szorzás egységleme.

```
(* prod ns = az ns egészszám lista elemeinek szorzata
prod : int list -> int
*)
fun prod (n::ns) = n * prod ns
| prod [] = 1
```

A listalemekek összege hasonlónan képezhető (0 az egységlelem). A megoldás legyen jobbrekurzív.

```
(* sum ns = az ns egészszám lista elemeinek összege
sum : int list -> int
*)
local (* s(ns, i) = i és az ns egészszám lista elemeinek
összege
s: int list * int -> int
*)
fun s (n::ns, i) = s(ns, i+n)
| s ([] , i) = i
in
  fun sum ns = s(ns, 0)
end
```

## 2. További magasabb rendű függvények

### 2.1. exists és forall

exists és forall a logikából jól ismert *kvantifikerek* ( $\exists, \forall$ ). Megvalósításuk SML-ben:

```
(* exists p xs = igaz, ha xs-nék vannak p-t kielégítő
eleme
exists : ('a -> bool) -> 'a list -> bool
*)
fun exists p [] = false
| exists p (x::xs) = p x orelse exists p xs

(* forall p xs = igaz, ha xs összes eleme kielégíti p-t
forall : ('a -> bool) -> 'a list -> bool
*)
fun forall p [] = true
| forall p (x::xs) = p x andalso forall p xs
```

Az isMem függvény megnézi, hogy egy adott elem előfordul-e egy listában.  
(Megjegyzés: isMem nem magasabb rendű!)

```
(* y isMem xs = igaz, ha y eleme xs-nék
isMem : "'a * 'a list -> bool
*)
fun filter p [] = []
| filter p (x::xs) = if p x
  then x :: filter p xs
  else filter p xs
```

infix isMem;
fun y isMem (x::xs) = y = x orelse y isMem xs
| \_ isMem [] = false

Az isMem függvény újra definíálhatjuk exists alkalmazásával:

```
(* y isMem xs = igaz, ha y eleme xs-nék
isMem : "'a * 'a list -> bool
*)
infix isMem;
fun y isMem xs = exists (sec1 y op=) xs
```

Megjegyzés: a *ezt követő típus váltója*,  
forall segítségével definíálhatunk pl. osztathatóságvizsgálatot:

```
(* divisible3 xs =
  igaz, ha xs minden eleme osztható 3-mal
divisible3 : int list -> bool
*)
fun divisible3 xs = forall (fn x => x mod 3 = 0) xs
```

Még tömöbben (jól látható a részleges alkalmazható függvényeljelölés elönnye):

```
val divisible3 = forall (fn x => x mod 3 = 0)
forall segítségével definíálhatjuk a halmazok diszjunkt voltát tesztelő disjoint
függvényt:
(* disjoint(xs, ys) = igaz, ha xs és ys metszete üres
disjoint : 'a list * 'a list -> bool
*)
fun disjoint (xs, ys) =
  forall (fn x => forall (fn y => x <> y) ys) xs
```

### 2.2. map és filter (ismétlés)

map egy paraméterként átadott függvényt alkalmaz egy lista minden elemére  
eredménye egy új lista. filter egy listából összeügyűli és egy új listába fűzi azokat  
az elemeket, amelyek a paraméterként átadott *predikátnak* kielégítik.

```
(* map f ls = az ls elemeiből az f transzformációval
előálló elemek listája
map : ('a -> 'b) -> 'a list -> 'b list
*)
fun map f [] = []
| map f (x::xs) = f x :: map f xs

(* filter p ls = ls elemei közül a p predikátumot
kielégítő elemek listája
filter : ('a -> bool) -> 'a list -> 'a list
*)
fun filter p [] = []
| filter p (x::xs) = if p x
  then x :: filter p xs
  else filter p xs
```

## else filter p xs

Lássunk néhány példát az alkalmazásukra!

```

- map (map (fn n => n * 2)) [[1], [2, 3], [4, 5, 6]];
  > val it = [[2], [4, 6], [8, 10, 12]]; : int list list
Két halmaz metszete ( $S \cap T$ ) például így definítható filter-rel (ss-ben  $S$ , ts-ben  $T$  elemeit tároljuk):

```

```

(* inter(ss, ts) = az ss és ts halmazok metszete
inter : "a list * "a list -> "a list
*)
fun inter (ss, ts) = filter (secr (op isMem) ts) ss

```

## 2.3. foldl és foldr

foldl balról jobbra (left to right), foldr jobbról balra (right to left) haladva egy kétargumentumú prefix függvényt (pl. op+, op\*) alkalmaz egy lista minden elemére. A két függvény specifikációja *infix* operátorral ( $\oplus$ ) írjuk fel, mert *infix* jelöléssel könnyebb megérteni a működésüket. ( $\oplus$  tetszőleges infix operátor)

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

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

Itt e az ún. egységelem. Látható hogy a elvezetik a kijelölt műveleteket, mindenki függvény *egyszentsítő* az eredeti kifejezést. Kifejezéshez általában – de nem minden! – jobbról balra haladva egyszerűsítünk, ezért foldr-tnha reduce néven definiálunk.

Aszociatív műveletek esetén mindenkor, hogy foldl-t vagy foldr-talkalmazzuk-e, nem szorosan az egységelem következő függ a dolog: ha  $e \oplus x = x$ , az egységemben azonban az egységem következő függ a dolog: ha mindenki egyenlőség fennáll – és azszociatív műveletek esetén ez a helyzet -, egyszerűen egy segelelmől beszélünk.

Közismert, hogy az összeadás és a szorzás aszociatív műveletek, a kivonás és az osztás azonban nem, ezért nem mindenkor, hogy pl.  $x - 0 \equiv 0 - x$  lesz-e a kivonás egyszerűsítésének az eredménye. E kis példából az is látható, hogy a kivonásnak – és hozzá hasonlóan az osztásnak – jobb oldali egységelene van.

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

Igl látszik, hogy e valóban az op $\oplus$  művelet egységeme, hiszen op $\oplus$ -t az összes lista alkalmazva e-t kapjuk eredményül. Most már definíálhatjuk foldl és foldr SML-változatát.

```

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

```

```

(* foldr f e xs = az xs elemeire jobbról balra haladva
  alkalmazott, kétoperandusú,
  e egységelő f művelet eredménye
foldr : ('a * 'b -> 'b) -> 'a list -> 'b
*)
fun foldr f e (x::xs) = f(x, foldr f e xs)
| foldr f e [] = e

```

alkalmazott, kétoperandusú,

e egységelő f művelet eredménye

$$\text{foldr} : ('a * 'b -> 'b) -> 'a list -> 'b$$

$$\text{fun foldr f e (x::xs) = f(x, foldr f e xs)}$$

$$|\text{foldr f e [] = e}$$

Mindkét függvények  $a * 'b -> 'b$  típusú függvény az első argumentumra. Vagyik őszne, hogy foldl iteratív függvény: második argumentumunkat viselkedik. Sajnos, foldr a veremben gyűjti a részeredményeket, és majd csak a lista kiürítésekor hajtja végre a kijelölt műveleteket.

Számos függvény írható fel foldl és foldr alkalmazásával, ha megadjuk a lista szomszéds eleméin végrehajtandó műveletet és az egységemet. Lásunk néhányat:

```

(* sum xs = xs elemeinek összegé
sum : int list -> int
*)
fun sum xs = foldl op+ 0 xs
(* prod xs = xs elemeinek szorzata
prod : int list -> int
*)
fun prod xs = foldl op* 1 xs
(* flat xs = az xs részlistáinak konkatenálásával
  előállította
flat : 'a list list -> 'a list
*)
fun flat xs = foldr op@ [] xs
A length függvény egy iteratív változata (inc olyan kétargumentumú
  segédfüggvény, amelyik nem használja a második argumentumát):
local (* inc(n, -) = n+1
  inc : int * 'a -> int
*)
  fun inc (n, _) = n + 1
  fun inc (n, -) = n + 1
in (* length ls = az ls lista hossza
  length : 'a list -> int
*)
  fun length ls = foldl inc 0 ls
end
Az append függvény is felírható így, de foldr-rel, mert a :: művelet nem
  asszociatív és jobbra köt. Jobb "egységelemek" azt a listát vesszi, amelyhez a bal
  oldali lista elemeit – jobbról balra haladva – egysével fűzzük hozzá .
(* append xs ys = az xs ys elé fűzésével előállító lista
append : 'a list -> 'a list -> 'a list
*)
fun append xs ys = foldr op:: ys xs

```

Beszűrő rendezés foldr-rel:

```
(* ins(x, ys) = az ys rendezett egészlista elemeiből és
   az ys-be a <= reláció szerint beszűrt
   ins : int * int list -> int list
   fun ins (x, []) = [x]
   | ins (x, y::ys) =
     if x <= y then x::y else y::ins(x,
                                         ys)
(* inssort ls = az ls egészlista elemeinek <= szerint
   rendezett listaja
   inssort : int list -> int list
   *)
   fun inssort xs = foldr ins [] xs
```

## 2.4. curry és uncurry

Most már könnyen definíálhatunk egy-egy olyan függvényt, amelyik egy részlegesen alkalmazható függvényt részlegesen nem alkalmazható függvényé, ill. egy részlegesen nem alkalmazható függvényt részlegesen alkalmazható függvényé alakít:

```
(* curry f x y = f részlegesen nem alkalmazható alakban
   curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c
   *)
   fun curry f x y = f(x,y)
```

```
(* uncurry f(x, y) = f részlegesen alkalmazható alakban
   uncurry : ('a -> 'b -> 'c) -> ('a * 'b) -> 'c
   *)
   fun uncurry f(x,y)= f x y
```

## 3. Adattípusok

### 3.1. A datatype deklaráció

Kedjük egy példával!

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

Az névben új összetett típusot hozunk létre. Az új típusnak négy adattípuskora (röviden konstruktora) van: King, Peer, Knight és Peasant, közülük King ün. adatkonstruktoralandó, a másik három ún. adattípuskoriggény. Az adatkonstruktorknak is van típusa:

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

King (király) csak egy van, ezért definíáltuk konstruktoralandókat. A Peer-t (önemest nemessé címe (string), birtokának neve (string) és sorzáma (int), a Knight-ot (lovagot) és a Peasant-ot (parasztot) csupán a neve (string)azonosítja. Példák:
```

```
- val persons = [King,
                 Peasant "Jack Cade",
                 Knight "Gawain",
                 Peer ("Duke", "Norfolk", 9)];
> val persons = [...] : person list
Mintaillesztéssel választhatók szét az esetek, pl. egy függvényben:
(* title p = p megszólítása
   title : person -> string
*)
fun title King = "His Majesty the King"
  | title (Peer (deg, ter, _)) = "The " ^ deg ^ " of "
  | title (Knight name) = "Sir " ^ name
  | title (Peasant name) = name
```

```
Itt pl. a title (Peasant name) a minita (pattern identifier). minden eset le kell fedni mintát, különösen hibábaütést kapunk. A minták tetszőleges összetevetek lehetnek (lehetnek bennük ennek listája, rekordok stb.). Példül a sirs függvény az összes Knight nevét összegyűjti a person típus személyek listájából:
```

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

Itt a változatok sorrendje fontos, mert ha más lenne, a ::ps minta nincs minden Knight-re, Peer-re és Peasant-ra illeszkedne (ti. ezek helyett áll it!), hanem Knight-ra is.

Az összes dizsunkt eset fölösrolása segíti az algoritmus helyességének belátását, bizonyítását. Miért vontunk össze mégis három esetet egyetlen változóban? Azért, mert a harmón eset részletezése hosszabbá tenné a program szöveget is, végrejárássát is. A bizonyítás sem okoz gondot, ha a harmadik sort felhérteles egyenlőtök tekintjük:

```
sirs(p::ps) = sirs ps #if vs-p#Knight s
```

A sorrend még fontosabb az alábbi példában, amelyben személyek hierarchiáját vizsgáljuk. Itt 16 helyett csak 7 esetet kell megkülönböztetni: azokat, amelyek igaz eredményt adnak:

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

### 3.2. A félsorolásos típus

Gyakori, hogy név csak néhány különböző értéket vehet fel (azaz a név által felvethető értékek halmaza kis számossági), ilyen esetben érdemes bevezetni a *félsorolásos típus* (enumeration type). A datatype deklaráció használható fél-sorolásos típus leírójára is, pl. így

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

A fél-sorolásos típusnak csak *konstruktordai* vannak. Az új típus alkalmazásához a person típusú típusa deklarálnunk kell:

```
datatype person = King
| Peer of degree * string * int
...
```

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

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

A belső bool típushoz hasonló Bool típus és hozzá a Not függvény például így is deklarálnánk, ill. definílhannánk.

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

### 3.3. Polimorf adattípusok

Látuk, hogy List nem típus, hanem *posícióú típusoperátor*, int list, int List list, (string \* string) List stb. azonban már típusok. A datatype deklarációval *tipusoperátor* is létrehozhatunk.

A belső 'a List típushoz hasonló 'a List listét és vele együtt a Nil és a Cons *adatkonstruktorokat* például így definíthetnánk:

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

A Cons *adatkonstruktőrű* egyéν alkalmazásával elég körülmenye 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* :::: (hatospont) *adatkonstruktőrűt*, hogy kényelmesebb lehölés használhassunk.!

```
infix 5 :::: ;
```

```
val op :::: = Cons
```

A :::: adatkonstruktőrűt közvetlenül a típusdeklarációban is definiálhatjuk:

```
infix 5 :::: ;
```

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

Következő példánk legyen két típus *megkülönböztetett* egysége, más néven diszjunkt uniója:

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

Itt három dolgot definíáltunk:

1. a kétfagytumú disun típusoperátor,

2. az In1 : 'a -> ('a, 'b) disun és

3. az In2 : 'b -> ('a, 'b) disun adatkonstruktőrűt egységeket.

('a, 'b) disun az 'a és 'b típusok megkülönböztetett egysései. Megkülönböztetének nevezik az egysést, mert később is bármikor meg tudjuk mondani, hogy egy ('a, 'b) disun típusú párt ezik vagy másik eleme melyik alaptípusból származik. Az új típusba tartozó értékek In1 × alaktak, ha x 'a típusú, és In2 Y alaktak, ha y 'b típusú. Az In1 és In2 konstruktőrűgevények olyan címekkel lekintethetők, amelyek az 'a típus megkülönböztetik a 'b típusát. (Megkülönbözik az 'a típusat a Pascal variabilis rekordja is.)

A megkülönbözik az 'a típusat használhatnánk (v.ö. az objektum-orientált programozással, ahol például egy *alakzat* osztálynak *legelálap*, *háromszög* vagy *kör* nevű leszármazottai lehetnek). Az SML-ben megkülönbözik egyséssel tudunk leírholni például *különböző típusú elemekből álló listát*:

```
[In2 King, In1 "Skócia"] : ((string, person) disun)
list
[In1 "zsarnok", In2 1040] : ((string, int) disun) list
|A :: : És az = kiözet szövek kell rakni, különben a feldolgozam gyellen (fájlsorokból álló) névnek tekinti a jelzorozatot.
```

