1. Lista elemeinek szorzata és összege (ismétlés)

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

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

A listaelemek *összege* hasonlóan képezhető (0 az egységelem). A megoldás legyen jobbrekurzív.

```
(* sum ns = az ns egészlista elemeinek összege
   sum :  int list -> int
*)
local (* s(ns, i) = i és az ns egészlista eleminek
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 *kvantorok* (∃, ∀). Megvalósításuk SML-ben:

```
(* exists p xs = igaz, ha xs-nek van 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-nek
   isMem :  ''a * ''a list -> bool
*)
```



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

Az `isMem` függvényt újradefiniálhatjuk `exists` alkalmazásával:

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

Megjegyzés: `''a` *egyenlőségi típus* típusváltozója.
`forall` segítségével definiálhatunk pl. oszthatóságvizsgálatot:

```
(* 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 tömörebben (jól látható a részlegesen alkalmazható függvényjelölés előnye):

```
val divisible3 = forall (fn x => x mod 3 = 0)
```

`forall` segítségével definiálhatjuk 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 összegyűjti és egy új listába fűzi azokat az elemeket, amelyek a paraméterként átadott *predikátumot* 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
```

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 definiálható `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 (**l**eft to right), `foldr` jobbról balra (**r**ight to left) haladva egy *kétargumentumú prefix* függvényt (pl. `op+`, `op*`) alkalmaz egy lista minden elemére. A két függvény specifikációját *infix* operátorral ($\oplus$) írjuk föl, mert *infix* jelöléssel könnyebb megérteni a működésüket. ($\oplus$ tetszőleges infix operátort helyettesít.)

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

Itt `e` az ún. egységelem. Látható, hogy ha elvégezzük a kijelölt műveleteket, mindkét függvény *egyszerűsíti* az eredeti kifejezést. Kifejezéseket általában – de nem mindig! – jobbról balra haladva egyszerűsítünk, ezért `foldr`-t néha `reduce` néven definiálják.

Asszociatív műveletek esetén mindegy, hogy `foldl`-t vagy `foldr`-t alkalmazzuk-e, *nemasszociatív műveletek* esetén azonban az *egységelem kötésétől* függ a dolog: ha $e \oplus x = x$, az egységelem bal oldali, ha $x \oplus e = x$, az egységelem jobb oldali. Ha mindkét egyenlőség fennáll – és asszociatív műveletek esetén ez a helyzet -, egyszerűen egységelemről beszélünk.

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

`foldl`-t és `foldr`-t nem specifikáltuk arra az esetre, amikor a lista üres, pótoljuk:

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

Jól látszik, hogy `e` valóban az op$\oplus$ művelet egységeleme, hiszen op$\oplus$-t az üres listára alkalmazva `e`-t kapjuk eredményül. Most már definiálhatjuk `foldl` és `foldr` SML-változatát.

```
(* 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

(* 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
```


Mindkét függvénynek `'a * 'b -> 'b` típusú függvény az első argumentuma. Vegyük észre, hogy `foldl` *iteratív* függvény: második argumentuma, amely kezdetben az egységelem, gyűjtőargumentumként viselkedik. Sajnos, `foldr` a veremben gyűjti a részeredményeket, és majd csak a lista kiürülésekor hajtja végre a kijelölt műveleteket.

Számos függvény írható fel `foldl` és `foldr` alkalmazásával, ha megadjuk a lista szomszédos elemein végrehajtandó műveletet és az egységelemet. Lássunk néhányat:

```
(* sum xs = xs elemeinek összege
   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 xss = az xss részlistáinak konkatenálásával
              előálló lista
   flat :  'a list list -> 'a list
*)
fun flat xss = foldr op@ [] xss
```

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
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 "egységelemnek" azt a listát vesszük, amelyhez a bal oldali lista elemeit – jobbról balra haladva – egyesével fűzzük hozzá .

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

Beszúró rendezés foldr-rel:

```
(* ins(x, ys) = az ys rendezett egészlista elemeiből és

                az ys-be a <= reláció szerint beszúrt
                x-ből álló lista
   ins :  int * int list -> int list
*)
fun ins (x, []) = [x]
  | ins (x, y::ys) =
              if x <= y then x::y::ys else y::ins(x,
ys)

(* inssort ls = az ls egészlista elemeinek <= szerint
                rendezett listája
   inssort :  int list -> int list
*)
fun inssort xs = foldr ins [] xs
```

2.4. curry és uncurry

Most már könnyen definiálhatunk egy-egy olyan függvényt, amelyik egy részlegesen alkalmazható függvényt részlegesen nem alkalmazható függvénnyé, ill. egy részlegesen nem alkalmazható függvényt részlegesen alkalmazható függvénnyé 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

Kezdjük egy példával!

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

person néven új összetett típust hoztunk létre. Az új típusnak négy *adatkonstruktora* (röviden: *konstruktora)* van: King, Peer, Knight és Peasant; közülük King ún. *adatkonstruktorállandó*, a másik három ún. *adatkonstruktorfüggvény.* Az adatkonstruktoroknak is van típusa:


```
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), 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 "
^ ter
  | title (Knight name) = "Sir " ^ name
  | title (Peasant name) = name
```

Itt pl. a title (Peasant name) a *minta* (pattern), és benne a name a *mintaazonosító* (pattern identifier).
Minden esetet le kell fedni mintával, különben hibaüzenetet kapunk. A minták tetszőlegesen összetettek lehetnek (lehetnek bennük ennesek, listák, rekordok stb.). Például a sirs függvény az összes Knight nevét összegyűjti a person típusú személyek egy listájából:

```
(* sirs ps = az összes Knight nevének 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 nemcsak King-re, Peer-re és Peasant-ra illeszkedne (ti. ezek helyett áll itt!), hanem Knight-ra is.
Az összes diszjunkt eset fölsorolása segíti az algoritmus helyességének belátását, bizonyítását. Miért vontunk össze mégis három esetet egyetlen változatban? Azért, mert a három eset részletezése hosszabbá tenné a program szövegét is, végrehajtását is. A bizonyítás sem okoz gondot, ha a harmadik sort *feltételes egyenletnek* tekintjük:

sirs(p::ps) = sirs ps **if** $\forall s \cdot p \neq$ Knight s

A sorrend még fontosabb az alábbi példában, amelyben személyek hierarchiáját vizsgáljuk. Itt 16 helyett csak 7 esetet kell megkülönböztetnünk: 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 (Peer _, Knight _) = true
  | superior (Peer _, Peasant _) = true
  | superior (Knight _, Peasant _) = true
  | superior _ = false
```

3.2. A felsorolásos típus

Gyakori, hogy egy név csak néhány különböző értéket vehet fel (azaz a név által felvehető értékek halmaza kis számosságú), ilyen esetben érdemes bevezetni a *felsorolásos típust* (enumeration type). A `datatype` deklaráció használható felsorolásos típus létrehozására is, pl. így

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

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

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

`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` típust és hozzá a `Not` függvényt például így is deklarálhatnánk, ill. definiálhatnánk:

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


3.3. Polimorf adattípusok

Láttuk, hogy `list` nem típus, hanem *postfix* pozíciójú *típusoperátor,* `int list`, `int list list`, `(string * string) list` stb. azonban már típusok. A `datatype` deklarációval *típusoperátort* is létrehozhatunk.
A belső `'a list` típushoz hasonló `'a List` listát és vele együtt a `Nil` és a `Cons` *adatkonstruktorokat* például így definiálhatnánk:

```
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 `:::` (hatospont) *adatkonstruktoroperátort,* hogy kényelmesebb jelölést használhassunk:[1]

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

A `:::` adatkonstruktoroperátort közvetlenül a típusdeklarációban is definiálhatjuk:

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

Következő példánk legyen 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 például a Pascal variábilis rekordja is.)
A megkülönböztetett egyesítés lehetővé teszi, hogy különböző típusokat használjunk ott, ahol egyébként csak egyetlen típust használhatnánk (v.ö. az objektum-orientált programozással, ahol például egy *alakzat* osztálynak *téglalap, háromszög* vagy *kör* nevű leszármazottai lehetnek). Az SML-ben megkülönböztetett egyesítéssel tudunk létrehozni például *különböző típusú elemekből álló listát:*

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

[1] A `:::` és az = közé szóközt kell rakni, különben a fordítóprogram egyetlen (írásjelekből álló) névnek tekinti a jelsorozatot.

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

```
  (* concat d = a d diszjunkt unió In1 címkéjű elemeinek
konkatenációja
     concat :  (string, 'a) disun list -> string
  *)
  fun concat [] = ""
    | (In1 s ::  ls) = s ^ concat ls
    | (In2 _ ::  ls) = concat ls
  - concat [In1 "Ó! ", In2 (1040, 1057), In1 "Skócia"];
  > val it = "Ó! Skócia :  string
```

Nézzük, mi a típusa az `In1 "Ó! Skócia"` kifejezésnek!
Az `In1` konstruktorfüggvény `'a -> ('a, 'b) disun` típusú, ezért `string` típusú argumentumra alkalmazva `(string, 'b) disun` típusú értéket ad eredményül. Az `In2 King` kifejezés típusa nyilvánvalóan `('a, person) disun` lesz.
Az `[In2 King, In1 "Skócia"]` kifejezésben mindkét alaptípust lekötjük, ezért ennek a kételemű listának a típusa a fent is látható `((string, person) disun) list`. Ugyanez lesz a háromelemű `[In1 "Ó", In2 King, In1 "Skócia"]` lista típusa is, hiszen az `'a` típusváltozót az `In1` konstruktorfüggvénnyel mindkét esetben `string`-nek adjuk meg.
Az `[In2 "Ó", In2 King, In1 "Skócia"]` kifejezés viszont hibajelzést eredményez, mert a `'b` típusváltozót nem lehet ugyanabban a kifejezésben egyszer így, másszor úgy lekötni.