

1. Polimorf típusellenőrzés

A típus nélküli (Lisp, Prolog stb.), ill. a gyengén típusos nyelvek (C, Java stb.) a programozónak nagyobb szabadságot adnak, de a tévedések lehetősége is nagyobb. Az erősen típusos nyelvek (p. Pascal, Ada, SML) a programozóhoz jobban korlátozik a biztonságosabbak.

Az SML-ben *polimorf típusellenőrzés* van: a szigorú típusellenőrzés flexibilis, automatikus típuslevezetéssel (type derivation, type inference) társult.

Nézzük a következő definíciót!

```
fun id x = x
```

Milyen típusú ítt az x ? Mindegy! Az ítid függvény ún. *polimorf függvény*, az x pedig *politípusú* azonosító. A típusnem általában a politípus jele α, β, γ stb., az SML-ben a, b, c stb. Az SML-értelemező válaszra a fennt definiáció tehát a következő:

```
> val id = fn : 'a -> 'a
```

A *percejellel* kezdődő típusneveket ('a-t, 'b-t, 'c-t stb.) *tipusváltózonak* nevezik, és *alfának, betánnak, gammának* stb. olvassuk.

A polimorfizmus több változat alkalmaztak a programozásban.

- Egy *polimorf név egyenlő* olyan algoritmust azonosít, amely ilyen típusú argumentumra alkalmazható; ez a *paraméteres polimorfizmus*.

- Egy *többszörösen terhelt név több* algoritmust azonosít, ahány típusú argumentumra alkalmazható, amennyifél; ez az ad-hoc vagy többszörös terheléses *polimorfizmus*.

- A polimorfizmus harmadik változatot öröklőképes *polimorfizmusraknevezések*!¹

A polimorf típus: *tipussáma*. Amikor a típusváltózt konkrét típusral helyettesítjük, e néma egy-egy példányát kapjuk.

Az egyenlőségvizsgálatot is megegedő ún. *egyenlősségi típusok* (equality types) típusváltóinak jelölésére *két percejellel* kezdődő neveket használunk: ' a ', ' b ', ' c ' stb.

Végül nezzünk két újabb, nagyon egyszerű példát polimorf függvények definíciására! Egy pár először, ill. második tagjának kiválasztására használhatók az alábbi *projektios* függvények, ahol ' a ' és ' b ' nem feltétlenül különböző típusok:

```
(* fst : 'a * 'b -> 'a
*)
fun fst(x,_) = x
(* snd : 'a * 'b -> 'b
*)
fun snd(_,y) = y
```

2. Rekurzív függvények

Rekurzív megoldás esetén a megoldandó feladatot olyan részfeladataakra bontjuk, amelyek közül egyet vagy többet (de nem minden!) az eredeti feladathoz hasonló módon oldunk meg.

¹ Örökfideses polimorfizmust az objektum-orientált programozáshoz alkalmazzák.

A kiértékelése a segédváltózó bevezetésére iteratívvá vált. Nagyon sok rekurzív függvény (de nem minden!) iteratívat alkotható segédváltózó bevezetésével, és így tárterületet takaríthatunk meg. Sokszor a végrehajtási idő is csökken, de sajnos néha nőhet is. Ha nem nyilvánvaló a nyereség, a lehetségesetől

2.1. Faktoriális „naív” rekurzióval

A faktoriális matematikai definíciója:

```
fac 0 = 1
fac n = n*fac(n - 1)
```

A faktoriális-függvény megvalósítása SML-ben:

```
(* fac n = n!
fac : int -> int
*)
fun fac n = if n = 0 then 1 else n * fac(n - 1)
```

Mohó kiértékelésnek minden $n = 4$ mellett:

```
fac(4) -> 4*fac(4-1) -> 4*fac(3) -> 4*(3*fac(2)) -> ... -> 4*(3*(2*1)) -> ... -> 24
```

A rekurzív kiértékelés szigorúan követi a matematikai definíciót.

2.1.1. Faktoriális iterációval (jobbrekurzióval)

A fenti kiértékelésben az a rossz, hogy a rekurzív végrehajtás során minden részeredményt tárolni kell. Ha a szorzás asszociativitását kihasználunk, nem kellene tárolni az összes tényezőt, csak az aktuális részeredményt. A számítógép a szorzás tulajdonságát (és bármely más tulajdonságát) persze csak akkor alkalmazza utasításról. Írunk ilyen függvényt!

Először a

```
(* faci(n, p) = p^n!
faci : int * int -> int
*)
fun faci(n, p) = if n = 0 then p else faci(n-1, p*p)
```

segédfüggvényt definíáljuk, majd felhasználjuk az eredetivelazonos hívási felületű függvényt megvalósításhoz:

```
(* fac n = n!
fac : int -> int
*)
fun fac n = faci(n, 1)
```

Nézzük a kiértékelést egyes triviális lépéseket összevonunk:

```
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 segédváltózóban (az ún. *gyűjtőfüggőmentumban, akkumulátorban*) gyűlik a részeredményt, ezért a tárgény állando marad. A kiértékelés tehát iteratív jellegű. Az ilyen függvényeket *terminális rekurzióknak* vagy *jobbrekurzióknak* (angolul: *tail vagy terminal recursion*) nevezik. A jövődprogramok felismerik az iteratívra alakítható rekurziót, és még hatékonyabb tágfoglalókot állítanak elő. A faci(n-1, n*p) rekurzív hívás eredménye – további számítások nélkül – közvetlenül faci(n, 1) eredményére adja. Az ilyen, ún. *terminális hívási* végre lehet halani úgy, hogy az értelmező – vagy forditóprogram azonban a p lokális változóba betírja az n- és a p új értékeit, majd visszaugrás a kód elejére helyezett, hogy teljesen, újból meghívja a függvényt. Az elbőző változatban a fac(n-1) hívás *terminalis hívás* nincs, mert az eredményet még meg kell szorozni n-nel.

faci-1 rekurzív függvényként definíáltuk, ezért viszonylag könnyű belátni a helyességét, ugyanakkor a kiértékelése a segédváltózó bevezetésére iteratívvá vált. Nagyon sok rekurzív függvény (de nem minden!) iteratívat alkotható segédváltózó bevezetésével, és így tárterületet takaríthatunk meg. Sokszor a végrehajtási idő is csökken, de sajnos néha nőhet is. Ha nem nyilvánvaló a nyereség, a lehetségesetől

módon kell felírni az algoritmust. Esetünkben `fac` valamivel hatékonyabb `fac-nál`, ugyanakkor a működése nehezebb érhető meg.

2.2. Fibonacci-számok

A Fibonacci-számok jól ismert definíciója:

$$F_0 = 0,$$

$$F_1 = 1,$$

$$F_n = F_{n-2} + F_{n-1}, n > 1.$$

Ha ezt így, ahogy van, átirjuk SML-re, használhatatlan programot kapunk: pl. F_{33} -öt egy 133 MHz-es Pentium processzoros számítógépen, Linux alatt csaknem 35 s alatt számolja ki az MOSML és csaknem 12 s alatt az smlnj! Keressük jobb megoldást!

Eloszor is `nextfib` néven olyan függvényt írunk, amely egy Fibonacci-szám párból előállítja a következő Fibonacci-szám párt:

```
(* nextfib(p,c) = a (p,c) Fibonacci-szám párt követő
   Fibonacci -szám párral
   nextfib : int * int -> int * int
   fibpair : int -> int * int
   fun nextfib (prev, curr) = (curr, prev + curr)
   fun fibpair n =
     if n = 1 then (0, 1) else nextfib(fibpair(n-1))
     nextfib.nextfib (... (nextfib(0, 1) ...))
```

A kiértekelése elég nehezen követhető, ugyanis `fibpair` a hívássorozatot állítja el. Ugyanakkor a végreírásra nagyon gyors, például a `fibpair` 4. hívás azonnal kijön a (433494437, 701408733) számpárt, amelynek a második tagja az `int`.`MaxInt`-nál még éppen nem nagyobb Fibonacci-szám. A keresett Fibonacci-számot a számpár második tagjának kiválasztásával kapjuk, például:

```
- #2 (fibpair 44);
> val it = 701408733 : int
```

Szép, áttekinthető és gyors megoldást kapunk *iterációval*:

```
(* iterfib(n,p,c) = a (p,c) Fibonacci -szám párt követő
   iterfib : int * int * int -> int
   fun iterfib (l, prev, curr) = curr
   | iterfib (n, prev, curr) = iterfib(n-1, curr, prev + curr)
```

Az else ágban a rekurzió terminális. `fib` iterfib-et hívja meg az `n`-edik Fibonacci-szám elbírálásához:

```
(* fib n = az n-edik Fibonacci -szám
   fib : int -> int
   | iterfib (1, prev, curr) = curr
   fun fib 0 = 0
```

| `fib n = iterfib(n, 0, 1)`

Nézzük egy példát fib redukciójára:

```
fib 7->iterfib(7,0,1)->iterfib(6,1,1)->iterfib(5,1,2)
->...->iterfib(1,8,13)->13
```

3. Egyidejű deklaráció

Az *egyidejű* (más néven *szimultán*) deklaráció elsősorban kölcsönösen rekurzív függvények definíálására használható. Kölcsönösen rekurzív függvényeket szekvenciális deklarációval nem lehet declarálni.
val `idi` = E_i and ... and `idn` = E_n

Az egyidejű deklaráció előbb a számítási E_i -t (kiszámlításuk sorrendje, mivel nem lehet mellekeltetni), majd a kiszámlított értékeket balról jobbra haladva, rendre hozzárendeli a megfelelő id_i -hez. Példáken bemutatunk egy nem igazán hatékony megoldást az egész számok páros, ill. páratlan voltát tesznelő even, ill. odd függvény megvalósítására:

```
(* even n = igaz, ha n páros
   even : int -> bool
   odd n = igaz, ha n páratlan
   odd : int -> bool
   *)
   fun even 0 = true
   | even n = odd(n-1)
   and odd 0 = false
   | odd n = even(n-1)
```

Az egyidejű deklaráció azonostok értékének felcserélésére is használható, például:

```
- val alma = "PIROS";
- val korte = "Barna";
- val alma = korte and korte = alma;
> val alma = "Barna" : string
> val korte = "PIROS" : string
```

4. Listák

A lista azonos típusú elemek végeletén (gyakorlatilag véges) sorozata. Példák:

```
[3, 5, 9, 13, 17, 21]
["alma", "meggy", "szilva"]
```

A lista *rekurzív adatszerkezetek* is tekintetjük. A rekurzív definíció szerint a lista

- vagy üres.

- vagy egy elemből és ezt az elemet követő listából áll.

Az üres listát – a listaműveletek *egyégelemét* [-]-el vagy mi-1-el jelölik. A legalább egy elemből álló lista első elemét a lista *főjének*, a többi elemből álló listát a lista *farkának* nevezik.

A listában az elemek sorrendje fontos. Egyes elemek ismétlődhettek. Az elemek típusa tetszőleges, de egy listának csak azonos típusú elemei lehetnek.²

```
> val it = [3, 5, 9, 13, 17, 21] : int list
> val it = ["alma", "meggy", "szilva"] : string list
-
```

²Más funkcionális nyelvekben, pl. a LISP-ben a listának különböző típusú elemei is lehetnek.

4.1. Lista létrehozása

Két konstruktorművelet használható lista létrehozására: a `[]` (nil)-konstruktortállandó és az infix pozíciójú `: konstruktoroperátor`.³

Egy lista vagy üres (`[]`) lehet, vagy `x::xs` alakú, ahol `x`-sel a lista *fejét*, `xs`-sel pedig a lista *farkát*, azaz az eredménytől egyetlen rövidítéssel részletezhető jelöljük. Környen elnéhelyez egy lista *első*, sok munkával az utolsó elemre.

A `[3 , 5 , 9]` jelölés⁴ rövidítés, négedig a `3 :: (5 :: (9 :: nil))` jelölés rövidítése. Azért, hogy ne kelljen zárójelet használni, az *infix* :: \ (negyespont) operátor *jobbra kör*: `3 :: 5 :: 9 :: nil`. Egy lista elemeiknél teljesüleg kifejezettséket adhatók meg.

Első példánk a lista alkalmazására legyen egy olyan rekurzív függvény, amely az `m::n` közötti egészek listáját adja eredményül. Ha `m`, `n`, az eredmény legyen az üres lista.

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

4.2. Lista elemeinek szorzata

A rekurzív megoldást általában az előforduló esetek elmenzésével, a jellemző esetek szétfárasztásával találjuk meg: lisák esetén általában az *üres* és a *nem üres* lista esetet kell megkülönböztetni. Az üres lista nem létező elemeinek szorzata célszerű 1-nek választani (mivel is?): `az 1 a szorzás egysegeleme`.

A `[]` minta csak az üres listára illeszkedik. Az `n :: ns` minta csak olyan listára illeszkedik, amelynek legalább egy eleme van, a minden zárolójelhez kell rakhni, mert a függvényalkalmazás precedenciája nagyobb a negyesponton.⁵

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

Az üres, ill. a nem üres listát kezelő ágak (a Prolog szóhasználatával: *klózok*) a jelen esetben *kölcsönösen kitájíjk* egymást (a minthál diszjunktak), ezért a két ág sora rendje az *eredmény szempontjából* /közönböző. De nem közönböző a sorrendjük *hatékonyiségi szempontjából*: mivel a vizsgált lista minden tag nem üres, amíg a rekurzív felbőgzés során a nem fogynak az elemei, az üreslista-vizsgálat az utolsó eset kivételével meghiúsul. Ezért a hatékonyiségi értelekben a két ágat célszerű fordított sorrendben felírni:

```
fun prod (n::ns) = n * prod ns
| prod [] = 1
```

Ha a függvény meghívásakor már az első minta illeszkedik az argumentumra, a második ág kiértekelésére nem kerül sor. (Amint tudjuk, a kifejezések kiértekelése halvói jobbra és feltülről lefelé halad.) Hasonló esetekben minden törekedni fogunk arra, hogy hatékony megoldást alkalmazunk.

A listaelemek összegéhez hasonlóan képezhető. Az összeadás egységeleme a 0.

³ `[]` és `nil` jelentése azonos. A :: konstruktoreceptor helyett sokszor a vele azonos hatású, de prefix pozíciójú *cons konstruktör* függvénytől beszélünk, amely azonban minden másik függvényt definíálhat az SML-ben.

⁴ Az SML-lista színezési szabályai a Prologban ugyanúgy definiálhatók. A Prologban ui. `[5 | [6]]` és `[5 , 6]` azonos listát jelölnek. Az SML-ben az `[5 :: [6]]`-tel jelölt lista `[5 , 6]`-tal azonos.

4.3. Lista legnagyobb eleme

Kicsit más a feladat egy lista *legnagyobb* (legkisebb) elemenek megkeresésekor:

- üres listának nincs legnagyobbi eleme,
 - egyelemű listában az egyetlen elem a legnagyobbi,
 - legalább kételemű lista esetén a legnagyobbi elemet úgy kapjuk meg, hogy vesszük a két elem közötti kiválasztjuk a nagyobbat.
- ```
(* maxl ns = az ns egészlista legnagyobb eleme
 maxl : int list -> int
 *)
 fun maxl (m::n::ns) =
 if m > n then maxl(m::ns) else maxl(n::ns)
 | maxl [m] = m
```

Az SML-értelmezőre a függvénydefinicírás figyelemzettel tizenötöt válaszol:

<sup>1</sup> Warning: pattern matching is not exhaustive  
Megjegyzések:

1. maxl üres listára nem alkalmazható; erre figyelmeztet a fenti tizenet. Később megmutatjuk, hogyan kell kezelni az ilyen, ún. *kivételeket*.
2. az `[m]` minta csak egyetlen *elemből* álló listára illeszkedik.
3. az `(m :: n :: ns)` minta csak olyan listára illeszkedik, amelynek legalább két eleme van.
4. Az algoritmus szempontjából minden gyenge lenne, hogy a lista elemei minden típusúak, de a > reláció mint tudjuk, többszörösen terheliő módon polimorf (1.1 szakasz). Mivel a programozó nem hozhat le a többszörösen terheliő neveket az SML-ben, a függvény definíciója el kell dönteni, hogy a > relációinak melyik változatát kell beírni a `maxl`-be (alapértelmezés szerint írt a többszörösen terheliő műveletek argumentumainak típusa). Később megmutatjuk, hogyan kell ún. *generikus* algoritmusokat írni.

#### 4.4. Karakter, füzér és lista

Az SML-ben a füzér egydimenziós karaktertömb (karaktersorozat), nem lista. A füzér a rekurszív feldolgozás során esetleg többször át kell alakítani listává, majd vissza füzérre. Két belső függvény van erre a cérra az SML-ben:

```
- explode "mosml";
 > val it = [#"m", #"o", #"s", #"m", #"l"] : char list
```

A kapott lista minden eleme egyetlen karakter

```
- implode it;
 > val it = "mosml" : string
Füzérkéből álló lista elemeit egyesíteni a concat-tal lehet:
```

```
- concat ["mo", "ml"];
 > val it = "mosml" : string
```

A következő szakaszokban néhány fontos listaeleme függvényt definíálunk.

#### 4.5. Listák vizsgálata és darabokra szedése

Három függvény mutatunk be ebben a csoportban: a null egy lista üres voltát vizsgálja, a hd egy nem üres lista első elemét, a tl egy nem üres lista első részlistáját (a lista farkát) adja eredményül.

```
(* null xs = igaz, ha az xs lista üres
 null : 'a list -> bool
 *)
 fun null (_ :: _) = false
 | null [] = true
```

Az alábbiás ( $\perp$ ) a már ismert mindeniséjét. Az eredmény szempontjából a minták felirásának sorrendje közömbös ebben a függvényben is.

```
(* hd : a nem üres xs lista feje
 hd : 'a list -> 'a
 *)
 fun hd (_ :: _) = x
```

Ez a hd csak neműres lista alkalmazható, amire az SML-értelemező figyelmeztet. Később bemutatjuk az üres listát is kezelni képes változatát.

```
(* tl : a nem üres xs lista farka
 tl : 'a list -> 'a list
 *)
 fun tl (_ :: xs) = xs
```

Ez a tl is csak neműres lista alkalmazható, amire az SML-értelemező figyelmeztet. Később bemutatjuk az üres listát is kezelni képes változatát.

hd és tl szelketorfüggvény, null pedig tesztelőfüggvény.

#### 4.6. Listák és egész számok

Ebben a csoportban is három függvényt mutatunk be: length egy lista hosszát adja eredményül; take egy lista elejéről vett adott számú elemből, drop egy lista elejéről adott számú elem elhagyásával képezi eredménylistát. Ha

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

akkor

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

length csaknaival változata a következő:

```
(* nlenghth xs = xs elemeinek száma
 nlenghth : 'a list -> int
 *)
 fun nlength (_ :: xs) = 1 + nlength xs
 | nlength [] = 0
```

Egy példa a függvény alkalmazására:

```
- nlength [[1, 2, 3], [4, 5, 6]];
 ^hd a head (fej), tL a tail (fark) szókör származik. null, hd és tl fenti definíciója csak illusztráció, ugyanis minden a hármon
 belüli függvénynek van definíciója az SML-ben.
```

<sup>6</sup>Length belső függvény, take és drop pedig a List könyvtárban van definíciója az SML-ben.

> val it = 2 : int

nlength rossz hatékonyságú, mert nem iterativ: az 1-esek a veremben csak gyűlik, gyűlik, amíg a maradéklista ki nem ürül. A függvény javított, iterativ változata:

```
(* length xs = xs elemeinek száma
 length : 'a list -> int
 *)
 local
 fun addlen (n, _ :: xs) = addlen (n+1, xs)
 | addlen (n, []) = n
 in
 fun length xs = addlen(0, xs)
 end
```

A nagyon gyakran használt (pl. könyvtárazott) függvények hatékonysága és robosztusága<sup>7</sup> fontos, még ha kevésbé szépek, kevésbé olvashatók is. A speciális feladatakról írt, ritkábban használt függvények azonban legyenek könyven olvashatók, és a helyességükre is egyszerűen lehessen belátni, bizonyítani.

take elbővítője:

```
(* take(xs, i) = az xs első i db elemeiből álló lista, ha i >= 0;
 take : 'a list * int -> 'a list
 *)
 fun take (x :: xs, i) = if i > 0 then x :: take(xs, i-1) else []
 | take ([], _) = []
(* take(xs, i) = az xs első i db elemeiből álló lista, ha i <= 0;
 take : 'a list * int -> 'a list
 *)
 fun take (x :: xs, i) = if i < 0 then x :: take(xs, i+1) else []
 | take ([], _) = []
A bemutatott változat az if-then-else alkalmazása miatt nem olyan elegáns, de robusztus: negatív i-re az üres listát adjja eredményül. Majdnem igyanek valamivel szébb:
```

```
(* take(xs, i) = az xs első i db elemeiből álló lista, ha i >= 0;
 take : 'a list * int -> 'a list
 *)
 fun take (_ , 0) = []
 | take ([], _) = []
 | take (x :: xs, i) = x :: take(xs, i-1)
take második változata negatív i-re a teljes listát visszaadja. (Míért?) Figyelem: ebben a definíciótban az ágak sorrendje nem közömbös! (Míért nem?)8
```

Nézzünk egy példát take egyszerűsítésére (egyes trivialis lépéseket összevonunk):

```
take ([9, 8, 7, 6], 3) => 9 :: take([18, 7, 6], 2) => 9 :: 8 :: take([7, 6], 1)
-> ... > 9 :: 8 :: 7 :: [] => 9 :: 8 :: 7
```

Az egyszerűsítési folyamatot bemutató példában a négyespontról (:) nem lista létrehozására használjuk, mint a programokban, hanem olyan listakitejelezéstől fürt felé, amelyeket az egyszerűsítés során azaz meghibásztanak, kiszűrhetően viselkedik. Szélsőséges körülmények számára például, ha egy függvény rikán előfordul, extrem éretre alkalmazunk.

Az SML-ben egy függvény robsztussá azt jelenti, hogy a függvény az értelmezési tartományába eső minden lehetséges argumentumra specifikálva van. Es e specifikáció szenni viselkedik. Példül a belső nd és L1 függvény, ha üres lista alkalmaznak.

<sup>7</sup>Egy eljárás, függvény, program stb. akkor robusztus, ha szélsőséges körülmények között is a specifikációjának megfelelően. Robosztusnak tekintjük a take függvényt, ha minden esetben bemeneti kéről később tesz szó. Robosztusnak tekintjük a take függvényt, ha minden esetben biztosan befeléződik; ha i negativ, az else az üres listát adja eredményül. Ugyanakkor emelj a robusztus megrövidek hármon, ha minden esetben a második az eredmény. Akármaznak take-eket, ha példig valamennyi lista alkalmazására. Ezért a könyvtári List.take és List.drop közötti kivételt jelez.

SML-értelemezőnek ki kell értékelnie. A lista elemeit az SML-értelmező előbb egyesével berakjá a verembe, majd hátról visszafelé haladva megint elővezi őket az eredménylista elolvasásához.

Erdemes-e megírni `take` iteratív változatát? Probáljuk meg: a részrendményeket gyűjtjük az egyik argumentumban:

```
(* rtake(i, xs, zs) =
 rtake : int * 'a list * 'a list -> 'a list
*)
fun rtake (_, [], taken) = taken
| rtake (i, xs, taken) =
 if i>0 then rtake (i-1, xs, xs@taken) else taken
(* drop(xs, i) = az xs első i db elemének elhagyásával
 előállító lista, ha i>0; xs, ha i<0
drop : 'a list * int -> 'a list
*)
fun drop (_ , [])= []
| drop (i, xs) = if i>0 then drop (i-1, xs) else xs@:
EZ a megoldás készenfekvő és szerencsételő iteratív. Az első ágban lévő :xs lista ugyanaz, mint a drop második argumentuma; később nemutunk tömörebb jelölést – a réteges mintái –, amely ilyen esetekben alkalmazható.
```

#### 4.7. Listák összefűzése és megfordítása

Ebben a szakaszban két függvényt, az `append`-et és a `rev`-et mutatjuk be. Az `append infix` változatát a `@` jelű jelöljük. A két listát egymetűen `append` így specifikálható:

$$[x_1, \dots, x_m] @ [y_1, \dots, y_n] = [x_1, \dots, x_m, y_1, \dots, y_n]$$

Az `xs`-t elöször az elemeire bontjuk, majd hátról visszafelé haladva fűzzük az elemeket az `ys`-hez, után a listákat csak előiről tudjuk felülírni.

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

*Infix* változat pl. így definíálhatjuk:

```
- infix 5 @;
- val @ = append;
Itt is, akárcsak take-nél, a lista építésénél költsége meghaladja a veremhasználat – tí. az adatok veremben való tárolásának – költségét, ezért nem érdemes iteratív változatot kidolgozni.
A Pascal, a C-explicit mutatókkal kezeli a listákat, ezért az egyik lista végen a mutatót átfirányítható a másik listára. Az ilyen, ún. destruktív frissítés gyorsabb, mint a másoló frissítés. Csak hogyan ez veszélyes lehet! Pl. mi van akkor, ha mindenkit argumentum ugyanarra a listára mutat?
```

Most nézzük a listát megfordító `rev` egy háló megoldását:

```
(* rev xs = xs megfordítva
rev : 'a list -> 'a list
*)
```

és egy példán `rev` redukciójára:

```
(* nrev [] = []
| nrev (x::xs) = (nrev xs) @ [x];
EZ eddig n lépés volt. A továbbiakban az aktuális bal szélső listát magint elemeire kell bontani, majd összerakni 0, 1, 2, ..., n – 1 lépéshén.
nrev nagyon rossz hatékonyságú: $O(n^2)$. De emlékeztünk csak vissza rtake-re: ott megfordult a listamelemek sorrendje, bár nem akartuk. Most ponosan ezt akarjuk, használunk tehát segédargumentumot a revto segédfüggvényben:
```

```
(* revto(xs, ys) = xs elemei fordított sorrendben ys elé fűzve
revto : 'a list * 'a list -> 'a list
*)
fun revto ([] , ys) = ys
| revto (x::xs, ys) = revto(xs, x::ys);
revto lépésszáma arányos a lista hosszával. Segítségével rev definíciója (revto lokális is lehete rev ben):
(* rev xs = xs megfordítva
rev : 'a list -> 'a list
*)
fun rev xs = revto (xs, []);
Egy 1000 elemű listát rev 1000 lépében, rev $\frac{1000 \cdot 1001}{2} = 50500$ lépében fordít meg. Hatalmas a nyereség!
```

#### 4.8. Listákból álló lista, párokból álló lista

Ebben a szakaszban megint hárrom függvényt definíltunk. `flat` kétszeres mélységű listából egyszerű mélységeit készít; combine két azonos hosszúságú lista elemeiből egyetlen, párokból álló lista pedig combine inverz függvénye.

```
mbor.flat([x_1, x_2, ..., x_m], [y_1, y_2, ..., y_n]) = [x_1, x_2, ..., x_m, y_1, y_2, ..., y_n]
(* flat xs = a kétszeres mélységű xs lista részlistáinak
 elemeiből képzett lista
flat : 'a list -> 'a list
*)
fun flat [] = []
| flat (ls::ls) = ls @ flat ls;
Az algoritmus elég gyors, ha ls jóval rövidebb lss-nél, combine specifikációja és definíciója:
mbor.combine([x_1, x_2, ..., x_m], [y_1, y_2, ..., y_n]) = [(x_1, y_1), (x_2, y_2), ..., (x_m, y_m)]
(* combine(xs, ys) = az xs és ys elemeiből képzett párok lista
combine : 'a list * 'b list -> ('a * 'b) list
*)
```

```

fun combine ([] , []) = []
| combine (x::xs , y::ys) = (x,y)::combine(xs , ys);

Az SML-értelemző függvényezet: nem fedtünk le minden esetet (ui. az argumentumként átadott listák különböző hosszúságúak lehetnek). Az ilyen esetek kezelésére a később ismertendő kivételekkel célez alkalmazás.

split-et combine inverz függvényeket definíáltuk:
(* split xys = a párok listáából előállított listapár
 split : ('a * 'b) list -> 'a list * 'b list
*)
fun split [] = ([],[])
| split ((x, y)::pairs) =
 let val (xs, ys) = split pairs
 in (x::xs, y::ys)
 end;

```

Iráterív változata segédargumentumokban gyűjti tüköön-külön a két listát, de sajnos megfordítva:

```

(* rsplit : ('a * 'b) list * 'a list * 'b list ->
 'a list * 'b list
*)
fun rsplit ([] , xs , ys) = (xs , ys)
| rsplit ((x, y)::pairs , xs , ys) = rsplit(pairs , x::xs , y::ys);

```

## 5. Részlegesen alkalmazható függvények

Tudjuk, hogy az SML-ben egy függvénynek csak egyetlen argumentuma van, de ez egy pán, egy ennes, egy másik függvény stb. is lehet. Több argumentumú függvényt olyan függvénynel is megvalósíthatunk, amely függvényt ad eredményül.

A részlegesen alkalmazható (partially applicable) függvényeket H. B. Curry amerikai matematikus után *címzések*nek nevezik, mohna a jelölést egy másik amerikai matematikustak, Schönfinkelnek köszönhetjük. Egy részlegesen alkalmazható függvény argumentumait egymástól egy vagy több szakköz jellegű karakterrel kell elválasztani.

Nézzük a következő függvénydefiníciókat:

```

(* prefix pre post = pre és post konkatenációja
*)
fun prefix pre post =
 let fun cat post = pre ^ post
 in cat post
 end;

```

> val prefix = fn : string -> (string -> string)

```

- val prefix = fn pre => (fn post => pre ^ post);
 > val prefix = fn : string -> (string -> string)

```

A két definíció ekvivalens, mindenkető a részlegesen alkalmazható prefix függvény definíciója. A függvény típusát létrehozó függvényt kiolvasható, hogy ha a prefix függvény strig típusú argumentumra alkalmazható és string típusú érték ad eredményt.

Egy részlegesen alkalmazható függvény alkalmazható csak az első, az első és a második stb. argumentumra. A részleges alkalmazható függvény minden argumentumát viselkedik. Nézzük

A részlegesen alkalmazható prefix függvény két argumentumú függvényként viselkedik. Nézzük néhány példát a használatára:

```

- prefix "Sir ";
 > val it = fn : string -> string
 - it "Georg Solti"
 > val it = "Sir Georg Solti" : string

prefix fenit definíció nehézsések, az alábbi változatjaval olvashatóbb:
- val knightify = prefix "Sir "
 - val dukify = prefix "The Duke of "
 - val lordify = prefix "Lord"

Természetesen a részlegesen alkalmazható függvények is lehetnek rekurzívak, pl.
(* replist n x = n db x értékből álló lista
 replist : int -> 'a -> 'a list
*)
fun replist 0 x = []
| replist n x = x::replist (n-1) x
 replist olyan függvény, amelyet írt típusú értékre alkalmazva olyan függvényt kapunk, amely 'a típusú értéket ad eredményül. Gyűjtőargumentummal javíthatunk repList hatékonyságán:
```

```

fun replist n x =
 let fun rp1 0 xs = xs
 | rp1 n xs = rp1 (n-1) (x :: xs)
 in rp1 n []
 end
```

Összefoglalva, a függvényalkalmazás olyan  $E_1$  alakú összetett kifejezés, amelyben az  $E$  függvényéről érteket eredményező, az  $E_1$  pedig tetszőleges kifejezés. Az  $E E_1 E_2 \dots E_n$  kifejezés nem más, mint a  $(\dots ((E_1) E_2) \dots E_n)$  kifejezés tövидеise.

Mint tudjuk, az SML-értelemezők a kifejezéseket *balról jobbra* haladva értékelik ki. A függvényalkalmazás *erősen köt*, precedenciájá a lehető legmagyarobb.  $A \rightarrow$  típusoperátor (*lambda* kör) ezért peldául a string  $\rightarrow$  (string  $\rightarrow$  string) típuskifejezés ekvivalens a string  $\rightarrow$  string  $\rightarrow$  string típuskifejezéssel (az SML-értelemezők a típuskifejezést redundáns zárójel nélküli írják ki a kepernyőre).

A részlegesen nem alkalmazható függvényt *uncurred* függvénynek is nevezik. Ha egy részlegesen nem alkalmazható függvény típusa ('a \* 'b)  $\rightarrow$  'c, akkor vele ekvivalens, részlegesen alkalmazható változatának a típusa 'a  $\rightarrow$  ('b  $\rightarrow$  'c).

## 6. Magasabb rendű függvények

A magasabb rendű függvények (angolul *higher-order functions* vagy *functional*) több között arra használhatók, hogy előre definiált magasabb rendű függvények alkalmazásával elkerüljük az explicit rekurzótudásokat. Ezáltal könnyebben olvasható és bizonyítható programokat írunk.

### 6.1. secl és secr

Gyakran hasznos, ha egy infix operátor egysik operandusát rögzítjük, a másikat szabadon hagyjuk, például ("Sir"  $\wedge$ ) ekvivalens a knightify függvényvel,

**( / 2.0 )**

olyan függvény, amely 2.0-vel oszt.

Sajnos, ezek a teljelősek így nem használhatók az SML-ben, secr és secr segítségevel azonban építeni függvényeket definíthattunk. A nevet záró 1. ill. r betű arra utal, hogy a bal (*left*), ill. a jobb (*right*) operandust kölcsönösen le.

```
(* secl x f y = f bal oldali argumentumát lekötő függvény
secl : 'a -> ('a * 'b -> 'c) -> 'b -> 'c
*)
fun secl x f y = f(x, y)

(* secr f y x = f jobb oldali argumentumát lekötő függvény
secr : ('a * 'b -> 'c) -> 'b -> 'a -> 'c
*)
fun secr f y x = f(x, y)

sec1 és secr paramtereinek nevét és sorrendjét úgy választottuk meg, hogy egrészt utaljanak a paraméterek szerepére és pozíciójára, másrészt tegyük lehetővé secl és secr részleges alkalmazását. Mielőtt folytatnánk az olvasást, vezessé le önhálóban a két függvény típusát!
```

Végül bemutatunk néhány példát a két függvény alkalmazására.

```
(* knightify n = a "Sir" és n konkatenációja
knightify : string -> string
*)
val knightify = secl "Sir" op^

(* recip r = az r valós szám reciproka
recip : real -> real
*)
val recip = secl 1.0 op/
val halve r = az r valós szám fele
halve : real -> real
*)
val halve = secr op/ 2.0
```

**6.2. Két függvény kompozíciója**

Két függvény kompozícióját általában a operátorral jelölök, maz o betű fogjuk használni.

```
(* f o g = az f és g függvények kompoziciójá
o : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b
*)
infix o;
fun (f o g) x = f(g x)
```

**6.3. map és filter**

map egy paraméterként átadott függvényt egy lista minden elemére, eredménye egy új lista, ter az egy listából összevittű és egy új lista tűzi 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
```

Lásunk néhány példát alkalmazásukról!

```
- 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 ∩ T) például így definíthető filter-rel (ss-ben S, ts-ben T elemeit tároljuk)
(* inter(ss, ts) = az ss és ts halmazzok metszete
inter : "a list * "a list -> "a list
PRE: ∀i, j • i ≠ j, si ∈ ss, sj ∈ ts • si ≠ sj, ∀i, j • i ≠ j, si ∈ ss, sj ∈ ts • si ≠ sj, i ∈ T, tj ∈ T • ti ≠ tj
*)
fun inter (ss, ts) = filter (secr (op isMem) ts) ss
```

**6.4. takeWhile és dropWhile**

Korábban take-kel és drop-pal találkoztunk: take egy lista elejéről vett addott számú elemből, drop pedig lista elejéről adott számú elem elhagyásával képezi listát. Néha olyan függvényekre van szükség, amelyek egy lista elejéről vett, addott predikátumot kielégítő elemekből, ill. egy lista elejéről addott predikátumot kielégítő elemek ellagyásával képeznek listát. Ilyen függvényeket névünk listát, ill. dropwhile néven.

```
(* takewhile p xs = az xs elejéről vett, p-t kielégítő
elemek listája
takeWhile : ('a -> bool) -> 'a list -> 'a list
*)
fun takewhile p [] = []
| takewhile p (x::xs) =
 if p x then x::takewhile p xs else []

(* dropwhile p xs = xs elejéről a p-t kielégítő
elemek elhagyásával
előálló lista
*)
fun dropwhile p [] = []
| dropwhile p (x::xs) =
 if p x then x::dropwhile p xs else []
```

```

dropwhile : ('a -> bool) -> 'a list -> 'a list
*)
fun dropwhile p [] = []
| dropwhile p (x::xs) =
 if p x then dropwhile p xs else x::xs

```

*dropwhile-ban rögzített minősítői* is alkalmazhatunk:

```

fun dropwhile p [] = []
| dropwhile p (xss as x :: xs) =
 if p x then dropwhile p xs else xss

```

### 6.5. exists és forall

exists és forall a logikából jól ismert kantonok ( $\exists, \forall$ ) megvalósítása SML-ben:

Az utóbbit elkövönök néhányszor megtörni. Probáljuk meg együtt! A függvénynek akkor kell igazítani, hogy az xs és az ys által ábrázolt halmazoknak egyetlen közös eleme sincs.

Az fn.  $y \Rightarrow x \Leftarrow y$  függvény akkor ad igaz értéket, ha  $y$  nem egyenlő valamely  $-e$  függvény számára kiköszönhető, és akkor ad igaz eredményt. A különböző fórumokon hivatalosan szerepel, hogy a függvény definícióját és tipusát nem is olyan könnyű megjegyezni. Javasoljuk az olvasónak, hogy csukja be a jezvit, és próbálja meg a specifikáció alapján rekonstruálni a két definíciót és levelezni a típusit!

Számos függvény írható fel foldl és foldr használatával, ha megadjuk a lista szomszédos elemeinek végrehajtandó műveletet és az egyélemeit. Lássunk néhányat:

```
(* sum xs = xs elemeinek összege
sum : int list -> int
*)

```

### 6.6. foldl és foldr

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

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

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

Itt  $e$  az ún. egységelem. Látható, hogy ha elvégzettük a kijelölt műveleteket, mindenkit függvényen egyszerűsítünk az eredeti kifejezést. Kifejezéseket általában – de nem minden! – jobbról balra haladva egyszerűsítünk ezért foldl-néha reduce néven definiáljuk.

*Associatív műveletek* esetén mindenügy, hogy foldl-t vagy foldl-t vagy foldr-t alkalmazzunk, nemasszociatív műveletek esetén azonban az egységelem követésétől függ a dolog: ha  $e \oplus x = x$ , az egységelem bal oldalán, ha  $x \oplus e = x$ , az egységelem jobb oldalán. Ha mindenket egységesen fennáll – és asszociatív műveletek esetén – ez a helyzet, egségenben egységelemeinkből beszélünk.

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

foldl-t és foldr-t nem specifikáltuk arra az esetre, amikor a lista üres, pötoljuk: foldl op $\oplus$  e [] = e és foldr op $\oplus$  e [] = e

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

```

(* foldl f e xs = az xs elemeire balról jobbra haladva
 alkalmazott, kétoperarendsű,
 alkalmazott, kétoperarendsű,
 egységelem 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

```

```

(* foldl f e xs = az xs elemeire jobbról balra haladva
 alkalmazott, kétoperarendsű,
 alkalmazott, kétoperarendsű,
 egységelem 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

```

```

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

```

Mindkét függvénynek ' $a * 'b -> 'b$ ' típusú függvény az első argumentumra. Vagyik őszre, hogy foldl iteratív függvény: második argumentuma, amely kezdetben az egységelem, gyűjtőargumentum-ként viselkedik. Sajnos foldr a veremben gyűjti a részeredményeket, és majd csak a lista kiürülésekor hajtja végre a kijelölt műveleteket.

A két függvény definícióját és tipusát nem is olyan könnyű megjegyezni. Javasoljuk az olvasónak, hogy csukja be a jezvit, és próbálja meg a specifikáció alapján rekonstruálni a két definíciót és levelezni a típusit!

Számos függvény írható fel foldl és foldr alkalmazásával, ha megadjuk a lista szomszédos elemeinek végrehajtandó műveletet és az egyélemeit. 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 xs = az xs részlistáinak konkatenálásával
előálló lista
*)
```

```
flat : 'a list list -> 'a list
```

```
fun flat xs = foldl 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
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 "egységelevennek" azt a listát vesszük, amelyhez a bal oldali lista elemet - jobbról balra haladva - egysével fűzzük hozzá.

```
(* append xs ys = az xs ys előszévével előállító lista
append : 'a list -> 'a list -> 'a list
*)
fun append xs ys = foldr op:: ys xs
```

A beszűró rendezés egy újabb változata foldr-rel:

```
(* ins(x, ys) = az ys rendezett egészlista elemeiből és az
 ys-be a <= reláció szerint beszűrt x-ból
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
insort : int list -> int list
*)
fun inssort xs = foldr ins [] xs
```

## 6.7. curry és uncurry

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

```
(* 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
```

## 6.8. További rekurzív függvények

Egy függvény  $n$ -edik hatványát így specifikálhatjuk:  $f^n = f(\dots f(f(x))\dots)$ , ha  $n \geq 0$  ( $f$   $n$ -szer ismétlődik). Definíálunk SML-függvényt ilyen ismétlések felirására!

```
(* repeat f n x = f n-edik hatvánnya az x helyen
repeat : ('a -> 'a) -> int -> 'a-> 'a
*)
fun repeat f n x =
 if n > 0 then repeat f (n-1) (f x) else x

Sok függvény fejezetőki repeat segítségével. Példák:
(* drop(xs, k) = xs első k elemének elhagyásával
előálló lista
*)
drop : 'a list * int -> 'a list
fun drop (xs, k) = repeat tl k xs

(* repelist k = füzér, amelyben "Ha!" k-szor ismétlődik
repelist : int -> string list
*)
fun repelist k = repeat (secl "Ha!" op::) k []
```