# 1 Példaprogramok: fák kezelése SML-ben

## 1.1 Fa adott tulajdonságának ellenõrzése (ugyanannyi)

Tekintsük az alábbi adattípus-deklarációt:

```
datatype 'a fa = A | B of 'a fa * 'a fa
                   | C of 'a fa * 'a fa * 'a fa
```

Írjon `ugyanannyi` néven olyan SML-függvényt, amely egy `'a fa` típusú fáról eldönti, hogy `B` és `C` csomópontjainak ugyanannyi `A` gyermeke (saját levele) van-e!
A függvény specifikációja:

```
ugyanannyi f = igaz, ha f B és C csomópontjainak ugyanannyi
               A gyermeke (saját levele) van
ugyanannyi :  'a fa -> bool
```

Példák:

```
ugyanannyi A = true;
ugyanannyi(B(B(A,A), C(A,A,A))) = false;
ugyanannyi(B(C(B(A,A), B(A,A),B(A,A)),
             B(C(A,A,A), C(A,A,A)))) = true;
```

**1. megoldás**

Összeszámoljuk, hogy a `B` és a `C` csomópontoknak hány `A` gyermeke van külön-külön, majd megnézzük, hogy a két szám egyenlõ-e.

```
(* ua :  'a fa -> int * int
   ua f = hármas, amelynek az 1. tagja a B, a 2. tagja a C
          csomópontok A leveleinek a száma, a 3. tagja pedig
          1, ha az aktuális csomópont, A, egyébként 0
*)
fun ua (B(b1, b2))=
      let val (b11, c11, a11) = ua b1
          val (b21, c21, a21) = ua b2
      in
         (b11 + b21 + a11 + a21, c11 + c21, 0)
      end
  | ua (C(c1, c2, c3)) =
      let val (b11, c11, a11) = ua c1
          val (b21, c21, a21) = ua c2
          val (b31, c31, a31) = ua c3
      in (b11 + b21 + b31,
          c11 + c21 + c31 + a11 + a21 + a31, 0)
      end
  | ua A = (0, 0, 1)
```



A rekurzió befejezõdését korábban *teljes indukcióval* igazoltuk, rekurzív adattípusok, pl. listák és fák esetén *strukturális indukcióval* bizonyítjuk.
*A teljes indukciót* az egész számok halmazán értelmezzük, és azon alapul, hogy minden egész után egy nála eggyel nagyobb egész *következik*. Az `INT` típust így is lehetne deklarálni: `datatype INT = 0 | Succ of INT`. A strukturális indukció a teljes indukció általánosítása rekurzív adattípusokra; szembetûnõ a hasonlóság az `INT` típus deklarációja és például a `datatype 'a List = Nil | Cons of 'a List` deklaráció között.
Az adott esetben a kiértékelés biztosan véget ér, mert `ua`-t vagy rekurzív módon alkalmazzuk az aktuális `B` vagy `C` fa egy részfájára, amely biztosan rövidebb az aktuális fánál, vagy befejezõdik a hívás, mert az aktuális fa `A`.

```
fun ugyanannyi f = let val (b, c, _) = ua f in b = c end
```

**2. megoldás**

A második paraméterként átadott számlálót eggyel növeljük, ha `B` csomópontnak van `A` gyermeke , és csökkentjük, ha `C` csomópontnak van `A` gyermeke. (`ua` és `ba`, ill. `ua` és `ca` kölcsönösen rekurzív függvények.)

```
(* ua :  'a fa * int -> int
   ua(f, num) = num, ha f = A; vagy num + az f-beli B-k A
      gyermekeinek száma - az f-beli C-k A gyermekeinek száma
*)
fun ua(A, num) = num
  | ua(B(x, y), num) = ba(x, ba(y, num))
  | ua(C(x, y, z), num) = ca(x, ca(y, ca(z, num)))

(* ba :  'a fa * int -> int
   ba(f, num) = num + a B-k A leveleinek száma
*)
and ba(A, num) = num + 1
  | ba(x, num) = ua(x, num)

(* ca :  'a fa * int -> int
   ca(f, num) = num - a C-k A leveleinek száma
*)
and ca(A, num) = num - 1
  | ca(x, num) = ua(x, num)

fun ugyanannyi f = ua(f, 0) = 0
```

**3. megoldás**

A 2. megoldás egyszerűsített változata egy újabb paramétert használ a *növekmény* átadására. Ennek értéke `B` csomópont esetén +1, `C` csomópont esetén pedig -1. (Szeredi Péter megoldása.)

```
local
  (* ua :  'a fa * int * int -> int
     ua (f, num, incr) = num + incr, ha f = A; vagy
              num + incr + az f-beli B-k A gyermekeinek száma
                         - az f-beli C-k A gyermekeinek száma
  *)
  fun ua (C(c1, c2, c3), num, incr) =
                ua(c1, ua(c2, ua(c3, num, ~1), ~1), ~1)
    | ua (B(b1, b2), num, incr) = ua(b1, ua(b2, num, 1), 1)
    | ua (A, num, incr) = num + incr
in
   fun ugyanannyi f = ua(f, 0, 0) = 0
end
```

## 1.2 Fa adott tulajdonságú részfáinak száma (bea)

Tekintsük az alábbi adattípus-deklarációt:

```
datatype fa = A | B of fa * fa | C of fa * fa * fa
```

Írjon `bea` néven olyan SML-függvényt, amely megszámlálja egy fa típusú fában azokat a `B` csomópontokat, amelyeknek minden részfája `B` vagy `A` (de nem `C`), és ezeknek a számát adja eredményül! Segédfüggvényt definiálhat.

```
bea f = azoknak az f-beli B-knek a száma, amelyeknek csak B
        vagy A részfájuk van
bea :  fa -> int
```

Példák:

```
bea A = 0;
bea(B(B(A,A),C(A,A,A))) = 1;
bea(B(C(B(A,A),C(A,A,A),B(A,A)),B(B(A,A),C(A,B(A,A),A)))) = 4;
```

**1. megoldás**

A `ba` segédfüggvény az olyan `B`-ket számlálja meg , amelyeknek egyetlen utódja sem `C`.

```
(* ba f = olyan (b, c) pár, ahol b a jó B-k száma f-ben,
          és c = true, ha f-ben van C
ba :  fa -> int * bool
*)
```



```
fun ba A = (0, false)
  | ba (C(bf, kf, jf)) =
       let val (bb, _) = ba bf
           val (kb, _) = ba kf
           val (jb, _) = ba jf
       in
          (bb+kb+jb, true)
       end
  | ba (B(bf, jf)) =
       let val (bb, bc) = ba bf
           val (jb, jc) = ba jf
           val b = bc orelse jc
       in
          (bb + jb + (if b then 0 else 1), b)
       end
```

Ha az aktuális fa `A`, a jó `B`-k száma nem változik, az õsei között pedig lehetnek jó `B`-k (ezért `false` az eredménypár második tagja). Ha az aktuális fa `C`, a részfái és ezek utódai között lehetnek jó `B`-k, de az õsei között egyetlen `B` sem lehet jó (ezért `true` az eredménypár második tagja). Ha az aktuális fa `B` és az utódai között nincs `C`, akkor 1-gyel megnöveljük a jó `B`-k számát, egyébként nem módosítjuk; az utódokra vonatkozó információt minden esetben változatlanul adjuk tovább.
A `bea` függvény a `ba` segédfüggvény által elõállított eredménypár elsõ tagját adja eredményül.

```
fun bea f = #1 (ba f)
```

**2. megoldás**

Ez a megoldás rosszabb hatékonyságú, mert a részfákat többször is bejárja, a már megszerzett információt nem használja fel újra.

```
fun bea f =
      let (* csupaAvB f = igaz, ha f-nek nincs C részfája
             csupaAvB : fa -> bool
          *)
          fun csupaAvB (B(A, A)) = true
            | csupaAvB (B(b1, A)) = csupaAvB b1
            | csupaAvB (B(A, b2)) = csupaAvB b2
            | csupaAvB (B(b1, b2)) =
                         csupaAvB b1 andalso csupaAvB b2
            | csupaAvB _ = false

          (* szamol f = f jó B csomópontjainak száma
             szamol :  fa -> int
          *)
```

```
    fun szamol A = 0
      | szamol (B(A, A)) = 1
      | szamol (b as B(f1, f2)) =
            szamol f1 + szamol f2 +
              (if csupaAvB b then 1 else 0)
      | szamol (c as C(f1, f2, f3)) =
            szamol f1 + szamol f2 + szamol f3
in
   szamol f
end
```

## 1.3 Fa adott tulajdonságú részfáinak száma (testverE)

Írjon `testverE` néven olyan SML-függvényt, amely a

```
datatype 'a fa = E | N of 'a fa * 'a fa * 'a fa
```

deklarációval megadott fában meghatározza azoknak az `E` leveleknek a számát, amelyeknek legalább egy testvérük van! Egy `E` levél testvérének az ugyanahhoz az `N` csomóponthoz tartozó másik `E` levelet nevezzük.
A függvény specifikációja:

```
testverE f = az E testvérek száma az f fában
testverE : 'a fa -> int
```

Példák:

```
testverE E = 0;
testverE (N(E,E,E)) = 3;
testverE (N(E,N(E,E,E),N(N(E,E,E),E,E))) = 8;
testverE (N(E,N(E,E,E),N(E,E,E))) = 6;
```

### Megoldás

A feladat és a megoldása nagyon egyszerû. Ügyeltünk arra, hogy csak a valóban megkülönböztetendõ esetekre írjunk fel változatokat. Figyelje meg, hogy az `N(E,f2,E)` és az `N(f1,E,E)` eseteket visszavezettük az `N(E,E,f3)` esetre. Ezzel ugyan egy lépéssel mélyítettük a rekurziót, de ha késõbb a program adott ágát javítani, módosítani kell, csökkent a hibák elkövetésének lehetõsége.

```
fun testverE (N(E,E,E)) = 3
  | testverE (N(E,E,f3)) = 2 + testverE f3
  | testverE (N(E,f2,E)) = testverE(N(E,E,f2))
  | testverE (N(f1,E,E)) = testverE(N(E,E,f1))
  | testverE (N(f1,f2,f3)) =
               testverE f1 + testverE f2 + testverE f3
  |testverE E = 0
```



## 1.4 Fa adott elemeinek összegzése (szintOssz)

Írjon `szintOssz` néven olyan SML-függvényt, amely egy bináris fában tárolt értékek szintenkénti összegébõl alkotott listát ad eredményül! A lista elsõ eleme az elsõ szinten lévõ gyökérelem értéke, második eleme a második szinten tárolt, legfeljebb két elem összege s.í.t. A fa típusa:

```
datatype itree = L of int | N of itree * int * itree
```

A függvény specifikációja:

```
szintOssz t = a t-beli elemek szintenkénti összegének listája
szintOssz :  itree -> int list
```

Példák:

```
szintOssz(L 1999) = [1999];
szintOssz(N(N(L 4, 2, L 5), 1, N(N(L 8, 6, L 9), 3, L 7)))
                                            = [1, 5, 22, 17];
```

A második példában használt fa és a szintenkénti összegek ábrázolása:

```
        1          1
       /  \
      2    3       5
     / \  / \
    4  5 6   7     22
   / \
  8   9            17
```

### 1. megoldás

A `listaOsszeg` segédfüggvénynek két, esetleg különbözõ hosszúságú lista elemeinek páronkénti összegébõl álló lista az eredménye. (A rövidebb listából hiányzó elemeket 0-kal pótolja.) Jobbrekurzív változata az elemek sorrendjét megfordítaná, ezért az eredeti sorrendet `rev`-vel helyre kellene állítani.

```
local
  (* listaOsszeg (xs, ys) = az xs és ys elemeibõl páronként
                             képzett összegek listája
     listaOsszeg :  int list * int list -> int list
  *)
  fun listaOsszeg (x::xs, y::ys) = x+y::listaOsszeg(xs, ys)
    | listaOsszeg ([], ys) = ys
    | listaOsszeg (xs, []) = xs
in
   fun szintOssz(N(left, x, right)) =
         x ::  listaOsszeg(szintOssz left, szintOssz right)
     | szintOssz (L x) = [x]
end
```

A `szintOssz` függvény az `N` csomópontban elõállítja a bal, ill. a jobb részfa szintenkénti összegeinek listáját, majd a két lista elemeit páronként összeadja. Az `L` levél egyetlen elemébõl egyelemû listát képez.

**2. megoldás**

Az `s0` segédfüggvény a `t` fa azonos szintjein lévõ elemeket hozzáadja az `xs` lista megfelelõ eleméhez, és ezt a listát adja eredményül. A fa gyökere a lista jobb szélsõ elemének felel meg: ahogy egyre mélyebbre haladunk a fában, úgy építjük a listát, ill. haladunk jobbról balra a már felépült listában.

```
local
  (* s0(t, xs) = az egyes szinteken lévõ t-beli és a megfelelõ
                 xs-beli elemek összegének a listája
     s0 : itree * int list -> int list
  *)
  fun s0 (L v, []) = [v]
    | s0 (L v, x::xs) = x+v::xs
    | s0 (N(l, v, r), []) = v::s0(l, s0(r, []))
    | s0 (N(l, v, r), x::xs) = x+v::s0(l, s0(r, xs))
in
   fun szintOssz t = s0(t, [])
end
```

**3. megoldás**

Vegyük észre, hogy a 2. megoldásban az `s0` segédfüggvény két-két klóza alig különbözik egymástól. A hasonlóságot még jobban kiemelhetjük:

```
  fun s0 (L v, xs as []) = 0+v::xs
    | s0 (L v, x::xs) = x+v::xs
    | s0 (N(l, v, r), xs as []) = 0+v::s0(l, s0(r, xs))
    | s0 (N(l, v, r), x::xs) = x+v::s0(l, s0(r, xs))
```

Az egymáshoz hasonló klózokat összevonhatjuk, ezzel (Szeredi Péter megoldása):

```
local
(* feje :  int list -> int
   feje xs = hd xs vagy 0, ha xs = []
*)
  fun feje [] = 0 | feje (x::_) = x

(* farka :  'a list -> 'a list
   farka xs = tl xs vagy [], ha xs = []
*)
  fun farka [] = [] | farka (_::xs) = xs
```



```
  (* s0(t, xs) = az egyes szinteken lévõ t-beli és a megfelelõ
                 xs-beli elemek összegének a listája
     s0 : itree * int list -> int list
  *)
  fun s0 (L v, xs) =
        feje x + v ::  farka xs
    | s0 (N(l, v, r), xs) =
        feje x + v ::  s0(l, s0(r, farka xs))
in
   fun szintOssz t = s0(t, [])
end
```

## 1.5 Kifejezésfa egyszerûsítése (`egyszerusit`)

Az alábbi adattípus-definíciók olyan kifejezésfát írnak le, amelynek a levelei egész számok, a gyökérelemei pedig a `++`, `--`, `**` és `//` mûveleti jelek:

```
datatype oper = ++ | -- | ** | //
datatype ExTr = Lf of int | Br of oper * ExTr * ExTr
```

Írjon olyan SML-függvényt `egyszerusit` néven, amely egy kifejezésfában az `m++n` alakú részkifejezések összes elõfordulását az összegükre, az `m**n` alakú részkifejezések összes elõfordulását a szorzatukra cseréli, a többi részkifejezést pedig változatlanul hagyja (`m` és `n` egész számok)!
Gondoljon a redukció során keletkezõ, hasonló alakú részkifejezések helyettesítésére, de kerülje el a végtelen rekurziót!
A függvény specifikációja:

```
egyszerusit kf = kf egyszerûsített változata, amelyben m++n,
                 ill. m**n összes elõfordulása helyén m és n
                 összege, ill. szorzata van
egyszerusit :  ExTr -> ExTr
```

Példák:

```
egyszerusit(Br(++,Lf 1,Lf 2)) = Lf 3;
egyszerusit(Br(//,
               Br(++,Br( **,Lf 3,Lf 4),
               Br(++,Lf 5,Lf 6)),Lf 7)) =
                                       Br(//,Lf 23,Lf 7);
egyszerusit(Br(//,Br(//,Lf 3,Lf 4),Lf 5)) =
                             Br(//,Br(//,Lf 3,Lf 4),Lf 5);
egyszerusit(Br(//,Br(--,Br( **,Lf 3,Lf 4),
                        Br(++,Lf 5,Lf 6)),Lf 7)) =
                     Br(//, Br(--, Lf 12, Lf 11), Lf 7);
```

**1. megoldás**

A ++ és a `**` mûveleti jeleket tartalmazó részkifejezések kezelésére két-két változatot kell írni: egyet-egyet a ++, ill. `**` mûveleti jelbõl és pontosan két levélbõl (jelöljük `Lf b` -vel és `Lf j`-vel) álló, és egyet-egyet a ++, ill. `**` mûveleti jelbõl és egyéb részfákból álló csomópontok kezelésére. Az elõbbi két esetben a kijelölt mûvelet elvégezhetõ, az eredmény az `Lf(b+j`, ill. az `Lf(b*j)` levél. Az utóbbi két esetben `egyszerusit` rekurzív hívásával elõször is a bal és a jobb részfát egyszerûsítjük. Elõfordulhat, hogy mindkettõ levéllé válik , ezért meg kell próbálni, hátha további rekurzív hívással lehet egyszerûsíteni az adott mûveleti jelbõl és a redukált részfákból összerakott fát is.
Ha az aktuális fa gyökerében más mûveleti jel van, `egyszerusit` rekurzív hívásával csak a bal és a jobb részfát egyszerûsítjük, további redukcióra nincs lehetõség. Nem lehet egyszerûsíteni a kifejezést akkor sem, ha az aktuális fa levél.

```
fun egyszerusit (Br( ++, Lf b, Lf j)) = Lf(b+j)
  | egyszerusit (Br( **, Lf b, Lf j)) = Lf(b*j)
  | egyszerusit (Br( ++, bf, jf)) =
      egyszerusit(Br( ++, egyszerusit bf, egyszerusit jf))
  | egyszerusit (Br( **, bf, jf)) =
      egyszerusit(Br( **, egyszerusit bf, egyszerusit jf))
  | egyszerusit (Br(mj, bf, jf)) =
                     Br(mj, egyszerusit bf, egyszerusit jf)
  | egyszerusit (kf as Lf v) = kf
```

**2. megoldás**

Három változat (klóz) összevonásával, a közös részek kiemelésével a megoldás rövidebbé tehetõ.

```
fun egyszerusit (Br( ++, Lf b, Lf j)) = Lf(b+j)
  | egyszerusit (Br( **, Lf b, Lf j)) = Lf(b*j)
  | egyszerusit (Br(mj, bf, jf)) =
      let val f = Br(mj, egyszerusit bf, egyszerusit jf)
      in
        if mj = ++ orelse mj = ** then egyszerusit f else f
      end
  | egyszerusit (kf as Lf v) = kf
```

## 1.6 Kifejezésfa egyszerûsítése (`coeff`)

Tekintse az alábbi típust és adattípust:

```
type term = int * char
datatype expr = ++ of expr * term | Z
infix 6 ++
```



Egy `term` típusú párt egy egész együttható és egy `char` típusú változónév szorzatának, egy `expr` típusú kifejezést `term` típusú tagok és `Z` (zérus) állandók összegének tekintünk. Írjon SML-függvényt `coeff` néven, amelynek `expr` típusú kifejezésbõl és `char` típusú változónévbõl álló pár az argumentuma, és az eredménye az adott változó együtthatóinak az összege az adott kifejezésben! Hatékony, jobbrekurzív programot írjon! Segédfüggvényt definiálhat.
A függvény specifikációja:

```
coeff (e, v) = v együtthatóinak az összege e-ben
coeff :  expr * char -> int
```

Példák:

```
coeff(Z ++ (2,#"a") ++ (3,#"b") ++ (~5,#"a") ++
                               (4,#"c"), #"a") = ~3;
coeff(Z ++ (2,#"a") ++ (3,#"b") ++ (~5,#"a") ++
                               (4,#"c"), #"x") = 0;
```

**Megoldás**

Figyeljük meg, hogy a `Z` az `expr` típusú kifejezések *baloldali egységeleme:* `Z` maga is `expr` típusú kifejezés, az `expr` típusú kifejezésekben pedig csak a bal oldalon állhat `expr` típusú kifejezés.
A `cf` segédfüggvény az `n` argumentumban gyûjti az `e`-beli `v`-k együtthatóinak az összegét. `v` `coeff`-ben lokális, a `cf` szempontjából azonban globális név.

```
fun coeff (e, v) =
          (* cf(e, n) = n + a v együtthatóinak az összege
                        e-ben
             cf :  expr -> int -> int
          *)
     let fun cf (e ++ (c, v0)) n =
                 cf e (n + (if v = v0 then c else 0))
           | cf Z n = n
     in
        cf e 0
     end
```

## 1.7 Szövegfeldolgozás (`parPairs`)

Írjon SML-függvényt `parPairs` néven, amely az argumentumként kapott füzérben található, egymáshoz tartozó kerek nyitó és csukó zárójelek pozícióiból alkotott párok listáját adja eredményül, tetszõleges sorrendben! A füzér karaktereit 1-tõl számozzuk. Segédfüggvényt definiálhat.
A függvény specifikációja:

```
parPairs s = az s füzérbeli, egymáshoz tartozó, kerek nyitó és
             csukó zárójelek pozícióiból alkotott párok listája
```

```
parPairs :  string -> (int * int) list
```

Példák:

```
parPairs "Zárójelmentes." = [];
parPairs ")" = [];
parPairs "(" = [];
parPairs "real(3*4) + (sin(0.5) - (11.4+3.4)) * 1.2" =
                 [(5, 11), (19, 23), (27, 38), (15, 39)];
```

**Megoldás**

Az alábbi megoldásban kihasználjuk, hogy a lista veremként, azaz LIFO-tárként használható: amit legutoljára rakunk bele, azt vesszük ki belõle legközelebb.
A füzért az `explode` függvény karakterlistává alakítja. A `pp` segédfüggvény, ha kerek nyitózárójelet talál, az indexét (azaz helyének sorszámát az eredeti füzérben), berakja a `bs` verembe. Ha csukózárójelet talál, és a `bs` verem nem üres, kiveszi a `bs`-bõl a megfelelõ nyitózárójel indexét, és a `(b, i)` párt berakja `ps`-be. Ha egyéb karaktert talál, egyszerûen továbblép a listában. `i` értékét minden egyes lépésben 1-gyel megnöveli. Ha a lista elfogy, a indexpárok `ps`-ben összegyûjtött listáját az eredeti sorrendbe rakva (azaz `rev`-et alkalmazva ) adja eredményül.

```
fun parPairs s =
      let (* pp (cs, i, bs, ps) =
               cs = a feldolgozandó karakterek listája;
               i  = a cs elsõ karakterének az indexe
                    ( = helye az eredeti füzérben);
               bs = a még le nem zárt nyitózárójelek indexének
                    fordított sorrendû listája;
               ps = az egymáshoz tartozó nyitó- és csukó-
                    zárójelek indexeibõl álló párok listája
             pp :  char list * int * int list *
                    (int * int) list -> (int * int) list
          *)
         fun pp (#"("::cs, i, bs, ps) =
                                 pp(cs, i+1, i::bs, ps)
         | pp (#")"::cs, i, b::bs, ps) =
                             pp(cs, i+1, bs, (b,i)::ps)
         | pp (_::cs, i, bs, ps) = pp(cs, i+1, bs, ps)
         | pp ([], _, _, ps) = rev ps
      in
         pp(explode s, 1, [], [])
      end
```