

# ABSZTRAKCIÓ FÜGGVÉNYEKKEL (ELJÁRÁSOKKAL)

Absztrakció függvényekkel (eljárásokkal) FP-5..7-2

## Elágazó rekurzió

- Korábban lineáris-rekurzív, ill. lineáris-iteratív folyamatokra láttunk példákat (faktoriális kiszámítása kétféleképpen).
- Most *elágazó rekurzióra* nézzünk példát: állítsuk elő a Fibonacci-számok sorozatát.
- Egy Fibonacci-számot az előző két Fibonacci-szám összege adja:

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

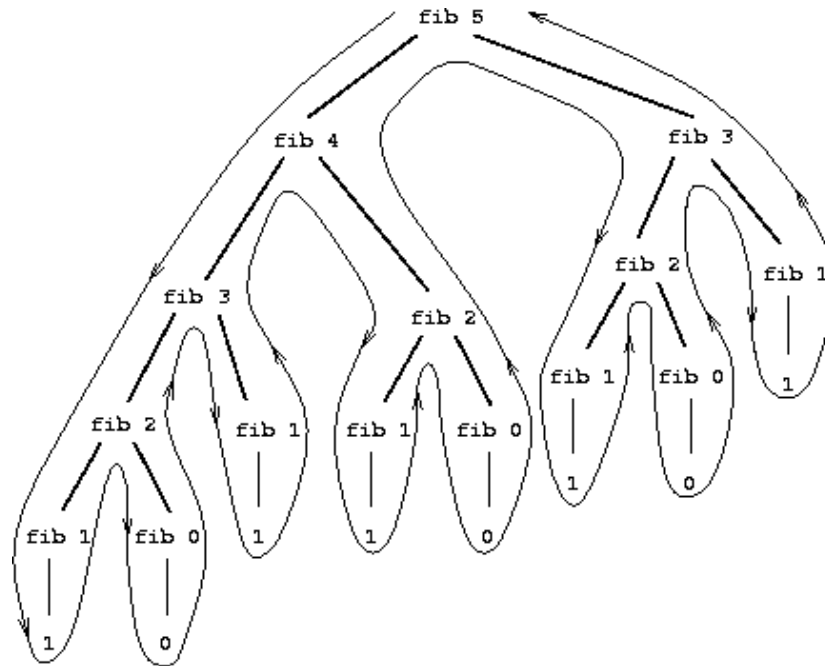
- A Fibonacci-számok matematikai definíciója könnyen átírható SML-függvénnyé:

$F(0) = 0$		fun fib 0 = 0
$F(1) = 1$		fib 1 = 1
$F(n) = F(n - 1) + F(n - 2)$ , ha $n > 1$		fib n = fib(n-1) + fib(n-2)

Emlékeztetőül: a `fib` függvény definíciójában a 3. klóznak az utolsónak kell lennie, mert az `n` minta minden argumentumra illeszkedik.

- A következő lapon látható ábra illusztrálja az elágazóan rekurzív folyamatot `fib 5` kiszámítása esetén.

## Elágazó rekurzió (folyt.)



- fib 5-öt fib 4 és fib 3, fib 4-et fib 3 és fib 2 kiszámításával stb. kapjuk.

## Elágazó rekurzió (folyt.)

- Az előző program alkalmas az elágazó rekurzió lényegének bemutatására, de szinte alkalmatlan a Fibonacci-számok előállítására!
- Vegyük észre, hogy pl. fib 3-at kétszer is kiszámítjuk, azaz a munkának ezt a részét kb. a harmadát) feleslegesen végezzük el.
- Belátható, hogy  $F(n)$  meghatározásához pontosan  $F(n+1)$  levélből álló fát kell bejárni, azaz ennyiszor kell meghatározni  $F(0)$ -t vagy  $F(1)$ -et.
- $F(n)$  exponenciálisan nő  $n$ -nel.  
Pontosabban,  $F(n)$  a  $\Phi^n / \sqrt{5}$ -höz közel eső egész, ahol  $\Phi = (1 + \sqrt{5})/2 \approx 1.61803$ , az ún. *arany metszés arányszáma*.  $\Phi$  kielégíti a  $\Phi^2 = \Phi + 1$  egyenletet.
- A megteendő lépések száma tehát  $F(n)$ -nel együtt exponenciálisan nő  $n$ -nel. Ugyanakkor a tárigény csak lineárisan nő  $n$ -nel, mert csak azt kell nyilvántartani, hogy hányadik szinten járunk a fában.
- Általában is igaz, hogy elágazó rekurzió esetén a lépések száma a fa csomópontjainak a számával, a tárigény viszont a fa maximális mélységével arányos.

## Elágazó rekurzió (folyt.)

---

- A Fibonacci-számok lineáris-iteratív folyamattal is előállíthatók.

Ha az  $a$  és  $b$  változók kezdőértéke rendre  $F(1) = 1$  és  $F(0) = 0$ , és ismétlődően alkalmazzuk az  $a \leftarrow a + b, b \leftarrow a$  transzformációkat, akkor  $n$  lépés után  $a = F(n + 1)$  és  $b = F(n)$  lesz. Az iteratív folyamatot létrehozó SML-függvény egy változata:

```
fun fib n = let fun fibIter (i, b, a) =
                if i = n then b
                else fibIter(i+1, a, a+b)
            in
                fibIter(0, 0, 1)
            end
```

- *Mintaillesztést* használhatunk, ha  $i$ -t nem növeljük, hanem  $n$ -től 0-ig csökkentjük.

Figyelem: a klózek sorrendje, mivel nem egymást kizáróak a minták, lényeges!

```
fun fib n = let fun fibIter (0, b, a) = b
                | fibIter (i, b, a) = fibIter(i-1, a, a+b)
            in
                fibIter(n, 0, 1)
            end
```

## Elágazó rekurzió (folyt.)

---

- A Fibonacci-példában a lépések száma elágazó rekurzióval tehát  $n$ -nel exponenciálisan, lineáris rekurzióval  $n$ -nel arányosan nőtt, kis  $n$ -ekre is hatalmas a nyereség!
- Téves lenne azonban azt a következtetést levonni, hogy az elágazó rekurzió használhatatlan. Amikor hierarchikusan strukturált adatokon kell műveleteket végezni, pl. egy fát kell bejárni, akkor az elágazó rekurzió (angolul: *tree recursion*) nagyon is természetes és hasznos eszköz.
- Az elágazó rekurzió numerikus számításoknál az algoritmus első megfogalmazásakor is hasznos lehet: gondoljunk csak arra, hogy milyen könnyű volt átírni a Fibonacci-számok matematikai definícióját programmá.
- Ha már értjük a feladatot, az első, rossz hatékonyságú változatot könnyebb átírni jó, hatékony programmá. Az elágazó rekurzió segíthet a feladat megértésében.

Az iteratív Fibonacci-algoritmushoz csak egy aprócska ötlet kellett. A következő feladatra azonban nem lenne könnyű iteratív algoritmust írni.

- Hányféleképpen lehet felváltani egy dollárt 50, 25, 10, 5 és 1 centesekre?
- Általánosabban: adott összeget adott érmékkel hányféleképpen lehet felváltani?

## Elágazó rekurzió (folyt.): pénzváltás

Tegyük föl, hogy  $n$  darab érme áll a rendelkezésünkre valamilyen (pl. nagyság szerint csökkenő) sorrendben. Ekkor az  $a$  összeg lehetséges felváltásainak számát úgy kapjuk meg, hogy

- kiszámoljuk, hogy az  $a$  összeg hányféleképpen váltható fel az első ( $d$  értékű) érmét kivéve a többi érmével, és ehhez
- hozzáadjuk, hogy az  $a - d$  összeg hányféleképpen váltható fel az összes érmével, az elsőt is beleértve – más szóval azt, hogy az  $a$  összeget hányféleképpen tudjuk úgy felváltani, hogy a  $d$  érmét legalább egyszer felhasználjuk.

A feladat tehát rekurzióval megoldható, hiszen redukálható úgy, hogy kisebb összegeket kevesebb érmével kell felváltanunk. A következő alapeseteket különböztessük meg:

- Ha  $a = 0$ , a felváltások száma 1.  
(Ha az összeg 0, csak egyféleképpen, 0 db érmével lehet „felváltani”.)
- Ha  $a < 0$ , a felváltások száma 0.
- Ha  $n = 0$ , a felváltások száma 0.

A példában a `firstDenomination` (magyarul *első címlet*) függvényt felsorolással valósítottuk meg. Tömörebb és rugalmasabb lenne a megvalósítása lista alkalmazásával.

## Elágazó rekurzió (folyt.): pénzváltás

```
fun countChange amount =
  let (* cC amount kindsOfCoins = az amount összes felváltásainak száma
      kindsOfCoins db érmével *)
    fun cC (amount, kindsOfCoins) =
      if amount < 0 orelse kindsOfCoins = 0 then 0
      else if amount = 0 then 1
      else cC (amount, kindsOfCoins - 1) +
          cC (amount - firstDenomination kindsOfCoins, kindsOfCoins)
    and firstDenomination 1 = 1
      | firstDenomination 2 = 5
      | firstDenomination 3 = 10
      | firstDenomination 4 = 25
      | firstDenomination 5 = 50
  in
    cC(amount, 5)
  end;
```

```
countChange 10 = 4; countChange 100 = 292;
```

*Gyakorló feladatok.*

- Írja át a `firstDenomination` függvényt úgy, hogy a címleteket egy lista tartalmazza.
- Írja meg a `cC` függvény mintaillesztést használó változatát!

## Hatványozás

- Az eddig látott folyamatokban a kiértékelési (végrehajtsági) lépések száma az adatok  $n$  számával lineárisan, ill. exponenciálisan nőtt. Most olyan példa következik, amelyben a lépések száma az  $n$  logaritmusával arányos.

- A  $b$  szám  $n$ -edik hatványának definícióját ugyancsak könnyű átrakni SML-be.

$$b^0 = 1 \quad \left| \begin{array}{l} \text{fun expt (b, 0) = 1} \\ b^n = b \cdot b^{n-1} \quad \left| \begin{array}{l} \text{expt (b, n) = b * expt(b, n-1)} \end{array} \right. \end{array} \right.$$

- A létrejövő folyamat lineáris-rekurzív.  $O(n)$  lépés és  $O(n)$  méretű tár kell a végrehajtásához.
- A faktoriálisszámításhoz hasonlóan könnyű felírni lineáris-iteratív változatát.

```
fun expt (b, n) =
  let fun exptIter (0, product) = product
      | exptIter (counter, product) =
          exptIter (counter-1, b * product)
  in
    exptIter(n, 1)
  end
```

- $O(n)$  lépés és  $O(1)$  – azaz konstans – méretű tár kell a végrehajtásához.

## Hatványozás (folyt.)

- Kevesebb lépés is elég, ha kihasználjuk az alábbi egyenlőségeket:

$$b^0 = 1$$

$$b^n = (b^{n/2})^2, \text{ ha } n \text{ páros}$$

$$b^n = b \cdot b^{n-1}, \text{ ha } n \text{ páratlan}$$

```
fun expt (b, n) =
  let fun exptFast 0 = 1
      | exptFast n =
          if even n
          then square(exptFast(n div 2))
          else b * exptFast(n-1)
      and even i = i mod 2 = 0
      and square x = x * x
  in exptFast n end
```

- A lépések száma és a tár mérete  $O(\lg n)$ -nel arányos. Konstans tárigényű iteratív változata:

```
fun expt (b, 0) = 1 (* Nem hagyható el! Miért nem? *)
  | expt (b, n) = let fun exptFast (1, r) = r
                    | exptFast (n, r) =
                        if even n then exptFast(n div 2, r*r)
                        else exptFast(n-1, r*b)
                    and even i = i mod 2 = 0
  in exptFast(n, b) end
```

# LISTÁK

További listakezelő függvények FP-5..7-12

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

- Legyen  $xs = [x_0, x_1, \dots, x_{i-1}, x_i, x_{i+1}, \dots, x_{n-1}]$ , akkor  
 $\text{take}(xs, i) = [x_0, x_1, \dots, x_{i-1}]$  és  $\text{drop}(xs, i) = [x_i, x_{i+1}, \dots, x_{n-1}]$ .
- take egy megvalósítása (jobbrekurzív-e? jobbrekurzívvá tehető-e? robusztus-e?)

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

- drop egy megvalósítása (jobbrekurzív-e? jobbrekurzívvá tehető-e? robusztus-e?)

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

- Könyvtári változatuk – `List.take`, ill. `List.drop` – ha az `xs` listára alkalmazzuk,  $i < 0$  vagy  $i > \text{length } xs$  esetén `Subscript` néven kivételt jelez.

## Lista redukciója kétoperandusú művelettel

Idézzük föl az egészlista maximális értékét megkereső `maxl` függvény két változatát:

- `maxl` jobbról balra egyszerűsítő (nem jobbrekurzív) változata

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

- `maxl` balról jobbra egyszerűsítő (jobbrekurzív) változata:

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

- Amint ez a példa is mutatja, vissza-visszatérő feladat egy lista redukciója kétoperandusú művelettel.
- Közös bennük, hogy  $n$  db értékből egyetlen értéket kell előállítani, ezért is beszélünk *redukcióról*.

## Lista redukciója kétoperandusú művelettel (`foldr`, `foldl`)

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

- A  $\oplus$  művelet  $e$  operandusa néhány gyakori műveletben – összeadás, szorzás, konjunkció (logikai „és”), alternáció (logikai „vagy”) – a (jobb oldali) *egységelem* szerepét tölti be.
- Asszociatív műveleteknél `foldr` és `foldl` eredménye azonos.

## Példák foldr és foldl alkalmazására

- `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`-el vagy `foldr`-rel. Kétooperandusú műveletként olyan segédfüggvényt (`inc`) alkalmazunk, amelyik *nem használja* az első paraméterét.

```
(* inc : 'a * int -> int
   inc (_, n) = n + 1 *)
fun inc (_, n) = n + 1;

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

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

## Lista: foldr és foldl definíciója

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

```
(* foldr f e xs = az xs elemeire jobbról balra haladva
   alkalmazott, kétooperandusú, 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;
```

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

```
(* foldl f e xs = az xs elemeire balról jobbra haladva
   alkalmazott, kétooperandusú, 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;
```



## További példák `foldr` és `foldl` alkalmazására

---

- 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 [x1, x2, x3] = (x1 :: (x2 :: (x3 :: ys)))
```

```
foldl op :: ys [x1, x2, x3] = (x3 :: (x2 :: (x1 :: 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)
```

## További példák `foldr` és `foldl` alkalmazására

---

- `maxl` két megvalósítása

```
(* maxl : ('a * 'a -> 'a) -> 'a list -> 'a
   maxl max ns = az ns lista max szerinti legnagyobb eleme
   *)
```

```
(* nem jobbrekurzív *)
```

```
fun maxl max []          = raise Empty
  | maxl max (n::ns)    = foldr max n ns
```

```
(* jobbrekurzív *)
```

```
fun maxl' max []        = raise Empty
  | maxl' max (n::ns)  = foldl max n ns
```

## Példák listák használatára: futamok előállítása

A futam olyan elemekből álló lista, amelynek szomszédos elemei kielégítenek egy predikátumot. Írjon olyan SML függvényt futamok néven, amelynek egy lista futamaiból álló lista az eredménye.

### • Első változat: futam és maradék előállítása két függvénnyel

```
(* futam : ('a * 'a -> bool) -> ('a * 'a list) -> 'a list
   futam p (x, ys) = az x::ys p-t kielégítő első futama (prefixuma *)
fun futam p (x, []) = [x]
  | futam p (x, y::ys) = if p(x, y) then x :: futam p (y, ys) else [x]

(* maradek : ('a * 'a -> bool) -> ('a * 'a list) -> 'a list
   maradek p (x, ys) = az x::ys p-t kielégítő futama utáni maradéka *)
fun maradek p (x, []) = []
  | maradek p (x, y::ys) = if p(x, y) then maradek p (y, ys) else y::ys

(* futamok1 : ('a * 'a -> bool) -> 'a list -> 'a list list
   futamok1 p xs = az xs p-t kielégítő futamaiból álló lista *)
fun futamok1 p [] = []
  | futamok1 p (x::xs) =
    let val fs = futam p (x, xs)
        val ms = maradek p (x, xs)
    in
      if null ms then [fs] else fs :: futamok1 p ms
    end
```

## Példák listák használatára: futamok előállítása (folyt.)

### • Példák:

```
futam op<= (1, [9,19,3,4,24,34,4,11,45,66,13,45,66,99]);
maradek op<= (1, [9,19,3,4,24,34,4,11,45,66,13,45,66,99]);
futamok1 op<= [1,9,19,3,4,24,34,4,11,45,66,13,45,66,99];
futamok1 op<= [99,1];
futamok1 op<= [99];
futamok1 op<= [];
```

### • Hatékonyságot rontó tényezők

1. futamok1 kétszer megy végig a listán: először futam, azután maradek,
2. p-t paraméterként adjuk át futam-nak és maradek-nak,
3. egyik függvény sem használ akkumulátort.

### • Javítási lehetőség

1. futam egy párt adjon eredményül, ennek első tagja legyen a futam, második tagja pedig a maradék; a futam elemeinek gyűjtésére használjunk akkumulátort,
2. futam legyen lokális futamok2-n belül,
3. az if null ms then [fs] else szövegrész törölhető: a rekurzió egy hívással később mindenképpen leáll.

## Példák listák használatára: futamok előállítása (folyt.)

---

### • Második változat: futam és maradék előállítása egy lokális függvénnyel

```
(* futamok2 : ('a * 'a -> bool) -> 'a list -> 'a list list
   futamok2 p xs = az xs p-t kielégítő futamaiból álló lista
*)
fun futamok2 p []          = []
  | futamok2 p (x::xs) =
    let (* futam : ('a * 'a list) -> 'a list * 'a list
        futam (x, ys) zs = olyan pár, amelynek első tagja az x::ys p-t
                           kielégítő első futama (prefixuma) a zs elé
                           fűzve, második tagja pedig az x::ys maradéka
        *)
        fun futam (x, []) zs          = (rev(x::zs), [])
          | futam (x, yys as y::ys) zs = if p(x, y)
                                          then futam (y, ys) (x::zs)
                                          else (rev(x::zs), yys);

        val (fs, ms) = futam (x, xs) []
    in
        fs :: futamok2 p ms
    end
```

## Példák listák használatára: futamok előállítása (folyt.)

---

### • Harmadik változat: az egyes futamokat és a futamok listáját is gyűjtjük

```
(* futamok3 : ('a * 'a -> bool) -> 'a list -> 'a list list
   futamok3 p xs = az xs p-t kielégítő futamaiból álló lista
*)
fun futamok3 p []          = []
  | futamok3 p (x::xs) =
    let (* futamok : ('a * 'a list) -> 'a list -> 'a list * 'a list
        futamok (x, ys) zs zss = az x::ys p-t kielégítő futamaiból álló
                                   lista zss elé fűzve
        *)
        fun futamok (x, []) zs zss          = rev(rev(x::zs)::zss)
          | futamok (x, yys as y::ys) zs zss =
            if p(x, y)
            then futamok (y, ys) (x::zs) zss
            else futamok (y, ys) [] (rev(x::zs)::zss)
    in
        futamok (x, xs) [] []
    end
```

## Nagyzárthelyi feladatsorainak megbeszélése

---

A 7. előadáson 2004. november 2-án az október 28-ai nagyzárthelyi feladatainak megoldását beszéltük meg.

A javítási útmutató a tárgy honlapján <<http://dp.iit.bme.hu/dp04a>> található meg.