

LISTÁK

További listakezelő függvények: foldr és foldl FP-5-3

Lista redukciója kétoperandusú művelettel (foldr, foldl)

- foldr jobbról balra, foldl balról jobbra haladva egy kétoperandusú műveletet (pontosabban egy *pár* 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.

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*.

Deklaratív programozás. BME VIK, 2004. tavaszi félév

(Funkcionális programozás)

További listakezelő függvények: foldr és foldl FP-5-4

Példák foldr és foldl alkalmazására

- isum egy egészlista elemeinek összegét, rprod egy valószínűségi lista 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-lel vagy foldr-rel. Kétoperandusú 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, 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;
```
- $\text{foldl } \text{op} \oplus e [x_1, x_2, \dots, x_n] = (x_n \oplus \dots \oplus (x_2 \oplus (x_1 \oplus e)) \dots)$
 $\text{foldl } \text{op} \oplus e [] = e$

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

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
```

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)
```

LISTÁK

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]

(* maradék : ('a * 'a -> bool) -> ('a * 'a list) -> 'a list
   maradék p (x, ys) = az x::ys p-t kielégítő futama utáni maradéka *)
fun maradék p (x, []) = []
  | maradék p (x, y::ys) = if p(x, y) then maradék 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 = maradék p (x, xs)
    in
      if null ms then [fs] else fs :: futamok1 p ms
    end
```

Deklaratív programozás. BME VIK, 2004. tavaszi félév

(Funkcionális programozás)

Listák: futamok előállítása FP-5-11

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, y::ys) zs = if p(x, y)
            then futam (y, ys) (x::zs)
            else (rev(x::zs), y::ys);
        val (fs, ms) = futam (x, xs) []
    in
      fs :: futamok2 p ms
    end
```

Deklaratív programozás. BME VIK, 2004. tavaszi félév

(Funkcionális programozás)

Példák listák használatára: futamok előállítása (folyt.)

• Példák:

```
futam op<= (1, [9,19,3,4,24,34,4,11,45,66,13,45,66,99]);
maradék 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 maradék,
2. p-t paraméterként adjuk át futam-nak és maradék-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.

Deklaratív programozás. BME VIK, 2004. tavaszi félév

(Funkcionális programozás)

Listák: futamok előállítása FP-5-12

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, 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
```

Deklaratív programozás. BME VIK, 2004. tavaszi félév

(Funkcionális programozás)

Függvények mint általános számítási módszerek

ABSZTRAKCIÓN FÜGGVÉNYEKKEL (ELJÁRÁSOKKAL)

- Láttuk, hogy a függvény (ill. általában az eljárás) olyan *absztrakció*, amely – a paraméterként átadott adatok konkrét értékétől függetlenül – összetett műveleteket ír le.
- Az olyan magasabbrendű függvény, amelynek függvény a paramétere, még *magasabb szintű* absztrakció, hiszen az általa megvalósított összetett műveletet nemcsak egyes konkrét adatoktól, hanem egyes konkrét műveletektől is függetlenül tesszük.
- A magasabbrendű függvény (eljárás) tehát valamilyen *általános számítási módszert* fejez ki.
- A következő lapokon két nagyobb példát ismertetünk: általános számítási módszert függvények *zérushelyeinek és fixpontjának* a megtalálására.

Egyenlet gyökeinek meghatározása intervallumfelezéssel

- Az intervallumfelezés módszere hatékony eljárás az $f(x) = 0$ egyenlet gyökeinek megtalálására, ahol f folytonos függvény.
- A közismert alapötlet a következő:
 - Megfelelően megválasztott a -ra és b -re, amelyekre $f(a) < 0 < f(b)$, f -nek legalább egy zérushelye van a és b között.
 - A zérushely megtalálásához legyen $x = a + b/2$. Ha $f(x) > 0$, akkor f zérushelyét a és x között, ha $f(x) < 0$, akkor x és b között kell keresnünk.
 - A keresést – a rekurziót – akkor hagyjuk abba, amikor két egymás utáni közelítő érték *eltérése* egy előre meghatározott értéknél kisebb lesz.
- Mivel az eltérés minden lépésben a felére csökken, az f zérushelyének megtalálásához szükséges lépések száma $O(L/T)$, ahol L az intervallum hossza kezdetben, és T a megengedett eltérés.
- A leírt algoritmust valósítja meg a `search` függvény (ld. a következő lapon):


```
(* search (f, negPoint, posPoint) = root of f x in the
      negPoint <= x <= posPoint interval
      PRE: f negPoint <= 0 and f posPoint >= 0
      *)
```

Egyenlet gyökeinek meghatározása intervallumfelezéssel (folyt.)

```
fun search (f, negPoint, posPoint) =
  let val midPoint = average(negPoint, posPoint)
  in
    if closeEnough(negPoint, posPoint)
    then midPoint
    else let val testValue = f midPoint
         in
            if positive(testValue)
            then search(f, negPoint, midPoint)
            else if negative(testValue)
            then search(f, midPoint, posPoint)
            else midPoint
          end
        end
  end
and average (x, y) = (x+y)/2.0
and closeEnough (x, y) = abs(x-y) < 0.001
and positive x = x >= 0.0
and negative x = x < 0.0
```

Egyenlet gyökeinek meghatározása intervallumfelezéssel (folyt.)

- Az előfeltételek betartását célszerű `search` alkalmazásokor ellenőrizni, nehogy rossz választ kapjunk az SML értelmezőtől.
 - `search(Math.sin, 4.0, 2.0) (* Helyes az eredménye *)`
`> val it = 3.14111328125 : real`
 - `search(Math.sin, 2.0, 4.0) (* Hibás az eredménye *)`
`> val it = 2.00048828125 : real`
- A `halfIntervalMethod` függvény elvégzi az ellenőrzést, és jelzi, ha `negPoint` vagy `posPoint` kezdeti értéke nem jó.


```
(* halfIntervalMethod (f, a, b) = root of f x in the
      a <= x <= b interval
*)
```
- Figyeljük meg az *ügyek szétválasztása* elv alkalmazását: `search` a gyökkeresési stratégiát valósítja meg, `halfIntervalMethod` pedig az előfeltételeket meglétét ellenőrzi.

A `print` és a `makestring` függvény; a `unit` típus; szekvenciális kifejezések

- A `print` függvény (típusa `string -> unit`), ha egy füzérre alkalmazzuk, a `nullas`-t adja eredményül, *mellékhatásként* pedig kiírja a a füzér értékét.
- A `()` vagy `{}` jelet *nullasnak* nevezzük, típusa: `unit`.
- A `unit` típus egyetlen eleme a `nullas`. A `unit` a típusműveletek ún. *egységelme*.
- A `makestring` függvény (típusa `numtxt -> string`) tetszőleges numerikus (`int`, `real`, `word`, `word8`), `char` és `string` típusú értéket `string` típusvá alakít.
- Az (`e1`; `e2`; `e3`) szekvenciális kifejezés *eredménye* azonos az `e3` kifejezés eredményével. Ha az `e1` és `e2` kifejezéseknek van mellékhatásuk, az érvényesül. (`e1`; `e2`; `e3`) egyenértékű a következő `let`-kifejezéssel:


```
let val _ = e1 val _ = e2 in e3 end
```
- Az `e1 before e2 before e3` kifejezés *eredménye* azonos az `e1` kifejezés eredményével. Ha az `e2` és `e3` kifejezéseknek van mellékhatása, az érvényesül. `e1 before e2 before e3` egyenértékű a következő `let`-kifejezéssel:


```
let val e = e1 val _ = e2 val _ = e3 in e end
```

Egyenlet gyökeinek meghatározása intervallumfelezéssel (folyt.)

- ```
fun halfIntervalMethod(f, a, b) =
 let val aValue = f a
 val bValue = f b
 in
 if negative aValue andalso positive bValue
 then search(f, a, b)
 else if negative bValue andalso positive aValue
 then search(f, b, a)
 else print ("Values " ^ makestring a ^ " and " ^
 makestring b ^ " are not of opposite sign.\n")
 end
```

Megjegyzés: `print`-ről és `makestring`-ről részletek a következő lapon.

- A függvénynek ez a változata hibás, mert az `if-then-else` feltételes kifejezés összes ágának *ugyanolyan típusú* eredményt *kell* adnia, márpedig `print` eredménye nem `int` típusú.
- Megoldás az (`e`; `f`) alakú ún. *szekvenciális kifejezés* használata: az értelmező kiértékeli `e`-t és `f`-et a felírt sorrendben, eredményül pedig `f` értékét adja.

## Egyenlet gyökeinek meghatározása intervallumfelezéssel (folyt.)

- ```
fun halfIntervalMethod(f, a, b) =
  let val (aValue, bValue) = (f a, f b)
  in
    if negative aValue andalso positive bValue
    then search(f, a, b)
    else if negative bValue andalso positive aValue
    then search(f, b, a)
    else (print ("Values " ^ makestring a ^ " and " ^
               makestring b ^ " are not of opposite sign.\n");
         0.0)
  end;

- halfIntervalMethod(Math.sin, 2.0, 4.0);
> val it = 3.14111328125 : real
- halfIntervalMethod(fn x => x*x*x-2.0*x-3.0, 1.0, 2.0);
> val it = 1.89306640625 : real
- halfIntervalMethod(Math.sin, 2.0, 2.5);
Values 2.0 and 2.5 are not of opposite signs
> val it = 0.0 : real
```

Függvény fixpontjának meghatározása

- Az $f(x) = x$ egyenletet kielégítő x az f függvény *fixpontja*.
- Egy f függvény valamely fixpontját megfelelő kezdőértékből kiindulva f rekurzív alkalmazásával határozhatjuk meg:

$f x, f(f x), f(f(f x)), f(f(f(f x))), \dots$

A rekurzió akkor fejezhető be, amikor már elhanyagolható mértékű a változás.

- A `fixedPoint` függvény paramétere egy pár; ennek első tagja egy függvény, amelynek a fixpontját keressük, a második tagja pedig a fixpont egy első közelítése.

```
(* fixedPoint (f, firstGuess) = fixpoint of f in the proximity
   of firstGuess with tolerance tolerance
*)
```

- Szükségünk van még a közelítés megkívánt pontosságára:

```
val tolerance = 0.00001;
```

Függvény fixpontjának meghatározása (folyt.)

- A fixpontszámítás hasonlít a négyzetgyökvonás korábban megbeszélt folyamatára: mindkettő azon alapul, hogy addig finomítjuk a közelítést, amíg valamilyen feltétel nem teljesül.
- A négyzetgyökvonás könnyedén megfogalmazható fixpontszámításként: ha x négyzetgyöke y , akkor $y * y = x$, azaz $y = x/y$. Az $f y = x/y$ függvény fixpontja tehát az x négyzetgyöke.

```
fun sqrt x = fixedPoint (fn y => x/y, 1.0);
```

- A megoldásunk rossz, ugyanis nem konvergál! Könnyen belátható: Legyen x négyzetgyökének első közelítése y_1 , a második $y_2 = x/y_1$, a harmadik $y_3 = x/y_2 = x/(x/y_1) = y_1$. Látható, hogy a folyamat sohasem ér véget.
- Az oszcillációt pl. úgy gátolhatjuk meg, hogy *korlátozzuk* két közelítő érték között a változás mértékét.
- Mivel a helyes válasz mindig az y közelítő érték és x/y között van, y -hoz x/y -nál *közelebb eső* új közelítő értéként y és x/y átlagát választhatjuk: $y \leftarrow (y + x/y)/2$.

```
fun sqrt x = fixedPoint (fn y => (y+x/y)/2.0, 1.0);
```

- Ezt a gyakran használható módszert *átlagsillapításnak* (angolul *average damping*) nevezik.

Függvény fixpontjának meghatározása (folyt.)

```
fun fixedPoint (f, firstGuess) =
  let
    fun closeEnough (v1, v2) = abs(v1-v2) < tolerance
    fun try guess =
      let
        val next = f guess
      in
        if closeEnough(guess, next)
        then next
        else try next
      end
  in
    try firstGuess
  end;
```

```
loadOne "Math";
fixedPoint(Math.cos, 1.0);
fixedPoint(fn y => Math.sin y + Math.cos y, 1.0);
```