

LISTÁK

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 : 'a list * int -> 'a list
```

```
  take (xs, i) = xs, ha i < 0; az xs első i db eleméből
                álló lista, ha i >= 0
```

```
*)
```

```
fun take (_, 0)      = []
  | take ([], _)    = []
  | take (x::xs, i) = x :: take(xs, i-1);
```

```
(* drop : 'a list * int -> 'a list
```

```
  drop(xs, i) = xs, ha i < 0; az xs első i db elemének
                eldobásával előálló lista, ha i >= 0
```

```
*)
```

```
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



Lista redukciója kétoperandusú művelettel (foldr, foldl)

- Vissza-visszatérő feladat egy lista redukciója kétoperandusú művelettel. Közös, hogy n db értékből egyetlen értéket kell előállítani (vö. *redukció*).
- `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⊕ e [x1, x2, ..., xn] = (x1 ⊕ (x2 ⊕ ... ⊕ (xn ⊕ e) ...))
foldr op⊕ e [] = e
foldl op⊕ e [x1, x2, ..., xn] = (xn ⊕ ... ⊕ (x2 ⊕ (x1 ⊕ e)) ...)
foldl op⊕ e [] = e
```

- Asszociatív műveleteknél `foldr` és `foldl` eredménye azonos.

Példák foldr és foldl alkalmazására

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

- 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 további megvalósítása

```
datatype 'a esetleg = Semmi | Valami of 'a
```

```
(* maxl10 : ('a * 'a -> 'a) -> 'a list -> 'a esetleg
   maxl10 max ns = az ns lista max szerinti legnagyobb eleme
*)
```

```
fun maxl10 max []          = Semmi
  | maxl10 max (n::ns)    = Valami(foldl max n ns)
```

```
(* maxl11 : ('a * 'a -> 'a) -> 'a list -> 'a esetleg
   maxl11 max ns = az ns lista max szerinti legnagyobb eleme
*)
```

```
fun maxl11 max []          = Semmi
  | maxl11 max (n::ns)    = Valami(foldr max n ns)
```

```
idoiro "maxl10, ns200k, foldl-rel: " maxl10 Int.max ns200k;
idoiro "maxl11, ns200k, foldr-rel: " maxl11 Int.max ns200k;
idoiro "maxl10, ns2M, foldl-rel: "   maxl10 Int.max ns2M;
```


Lista redukciója bal oldali egységelemű függvénnyel (foldL)

● A kivonás művelete balra köt: $x_1 - x_2 - x_3 - x_4 = ((x_1 - x_2) - x_3) - x_4$.

● Nem feleltethető meg sem foldr-nek, sem foldl-nek.

`foldr op⊕ e [x1, x2, ..., xn] = (x1 ⊕ (x2 ⊕ ... ⊕ (xn ⊕ e) ...))`

`foldl op⊕ e [x1, x2, ..., xn] = (xn ⊕ ... ⊕ (x2 ⊕ (x1 ⊕ e)) ...)`

● Nevezzük foldL-nek a listában *balról jobbra* haladó, alábbi specifikációjú függvényt. Vegyük észre, hogy \oplus *bal oldali* egységelemet vár.

`foldL op⊕ e [x1, x2, ..., xn] =`

`(... ((e ⊕ x1) ⊕ x2) ⊕ ... ⊕ xn)`

● foldL olyan kétargumentumú függvényt vár, amelynek az „egységelem” (valójában: a részeredmény) az *első* argumentuma: $f : 'a * 'b \rightarrow 'a$.

`(* foldL : ('a * 'b -> 'a) -> 'a -> 'b list -> 'a`

`foldL f e xs = az xs elemeire balról jobbra haladva`

`alkalmazott, kétoperandusú, e egységelemű`

`f művelet eredménye *)`

`fun foldL f e (x::xs) = foldL f (f(e, x)) xs`

`| foldL f e [] = e;`

Példák listaelemek különbségének és hányadosának képzésére

- Az `e` argumentum aktuális értéke a sorozat *első* eleme – a *kisebbítendő*, ill. az *osztandó*.

```
foldL op- 20 [] = 20;
```

```
foldL op- 20 [5, 6, 7] = (((20 - 5) - 6) - 7);
```

```
foldL (op div) 180 [] = 180;
```

```
foldL (op div) 180 [2, 3, 5] = (((180 div 2) div 3) div 5);
```

- Ha többször használjuk `e` műveleteket, érdemes nekik nevet adni. A *kisebbítendő*, ill. az *osztandó* speciális kezelését elrejtjük.

```
fun subtract ns = foldL op- (hd ns) (tl ns);
```

```
subtract [20, 5, 6, 7] = (((20 - 5) - 6) - 7);
```

```
fun divide ns = foldL op div (hd ns) (tl ns);
```

```
divide [180, 2, 3, 5] = (((180 div 2) div 3) div 5);
```

Listaelemek különbsége és hányadosa foldl-lel és foldr-rel

- Igazság szerint foldL felesleges: a feladat jól megoldható foldl-lel vagy foldr-rel is.

```
fun subtract1 ns = hd ns - foldl op+ 0 (tl ns);
subtract1 [20, 5, 6, 7] = (((20 - 5) - 6) - 7);
```

```
fun divide1 ns = hd ns div foldl op* 1 (tl ns);
divide1 [180, 2, 3, 5] = (((180 div 2) div 3) div 5);
```

- foldr és foldl típusa, ha egyparaméteres függvénynek tekintjük őket (a \rightarrow jobbra köt!):

```
foldr, foldl : ('a * 'b -> 'b) -> ('b -> 'a list -> 'b)
```

Azaz ha foldr-t vagy foldl-t egy $'a \rightarrow * 'b \rightarrow 'b$ típusú függvényre alkalmazzuk, akkor olyan függvényt ad eredményül, amelyet egy $'b$ típusú egységelemre és egy $'a \text{ list}$ típusú listára alkalmazva $'b$ típusú (redukált) értéket kapunk.

ÖSSZETETT ADATTÍPUSOK



Felhasználói adattípusok: ismét a `datatype` deklarációról

- `person` néven új összetett típust hozunk létre:

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

- Az új típusnak négy *adatkonstruktor*a (röviden: *konstruktor*) van: `King`, `Peer`, `Knight` és `Peasant`.
- `King` ún. *adatkonstruktorállandó*, a többi ún. *adatkonstruktorfüggvény*.
- Az adatkonstruktoroknak is van típusuk:

```
King :      person
Peer  :      string * string * int -> person
Knight :      string -> person
Peasant :      string -> person
```

A datatype deklaráció (folyt.)

```
King :    person
Peer  :    string * string * int -> person
Knight :    string -> person
Peasant :    string -> person
```

- King (király) csak egy van, ezért definiálhattuk konstruktorállandóként.
- A Peer-t (főnemest) nemesi címe (string), birtokának neve (string) és sorszáma (int) azonosítja.
- A Knight-ot (lovagot) és a Peasant-ot (parasztot) csupán a neve (string) azonosítja.
- Példa a person adattípus alkalmazására:
 - val persons = [King, Peasant "Jack Cade", Knight "Gawain",
Peer("Duke", "Norfolk", 9)];
 - > val persons = [King, Peasant "Jack Cade", ...] : person list
- Az egyes esetek mintaillesztéssel választhatók szét.
- Minden esetet le kell fedni mintával; ha nem, figyelmeztetést kapunk.
- A minták tetszőlegesen összetettek lehetnek.

A datatype deklaráció (folyt.)

- Az alábbi példában a négy közül az egyik a `Peasant` name *minta*, és benne name a *mintaazonosító*.

```
(* title : person -> string
   title p = p megszólítása *)
fun title King = "His Majesty the King "
  | title (Peer (deg, ter, _)) = "The " ^ deg ^ " of " ^ ter
  | title (Knight name) = "Sir " ^ name
  | title (Peasant name) = name
```

- A `sirs` függvény az összes `Knight` nevét összegyűjti a `person` típusú személyek egy listájából (a változatok sorrendje *fontos* az `_` miatt!):

```
(* sirs : person list -> string list
   sirs ps = az összes Knight nevének listája *)
fun sirs [] = []
  | sirs ((Knight s)::ps) = s::sirs ps
  | sirs (_::ps) = sirs ps
```

A datatype deklaráció (folyt.)

- Ha más lenne a változatok sorrendje, a `_ :: ps` minta nemcsak a `King-re`, a `Peer-re` és a `Peasant-ra` illeszkedne (ti. ezek helyett áll a példában), hanem a `Knight-ra` is.
- Az összes diszjunkt eset fölsorolása segíti az algoritmus helyességének belátását, bizonyítását.
- Azért vontunk össze három esetet egyetlen változatban, mert a részletezésük hosszabbá tenné a program szövegét is, végrehajtását is.
- A bizonyítás nem okoz gondot, ha a függvény harmadik sorát (`sirs (_ :: ps) = sirs ps`) *feltételes egyenletnek* tekintjük:

$$\text{sirs}(p :: ps) = \text{sirs } ps \text{ if } \forall s. p \neq \text{Knight } s.$$

A datatype deklaráció (folyt.)

- A sorrend még fontosabb a következő példában, amelyben személyek hierarchiáját vizsgáljuk. Itt 16 helyett csak 7 esetet kell megkülönböztetnünk: azokat, amelyek *igaz* eredményt adnak.

```
(* superior : person * person -> bool
   superior (p, r)= igaz, ha p magasabb rangú r-nél *)
fun superior (King, Peer _) = true
  | superior (King, Knight _) = true
  | superior (King, Peasant _) = true
  | superior (Peer _, Knight _) = true
  | superior (Peer _, Peasant _) = true
  | superior (Knight _, Peasant _) = true
  | superior _ = false
```

Felsorolós típus `datatype` deklarációval

- Gyakori, hogy egy név csak néhány különböző értéket vehet fel (azaz a név által felvehető értékek halmaza kis számosságú), ilyen esetben érdemes *felsorolós típust* létrehozni a `datatype` deklarációval. Pl.

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

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

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

Felsorolásos típus datatype deklarációval (folyt.)

- A degree típusú adatok feldolgozásakor külön-külön elemezzük az előforduló eseteket, pl.

```
(* lady : degree -> string
   lady p = p főnemes hitvesének rangja *)
fun lady Duke      = "Duchess "
  | lady Marquis   = "Marchioness"
  | lady Earl      = "Countess"
  | lady Viscount  = "Viscountess"
  | lady Baron     = "Baroness"
```

- A belső bool típushoz hasonló Bool típust és hozzá a Not függvényt például így is deklarálnánk, ill. definiálnánk:

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

Polimorf adattípusok

- Láttuk, hogy a `list postfix` pozíciójú *típusoperátor*, nem típus: a `datatype` deklaráció az adatkonstruktorok mellett *típuskonstruktort* is létrehoz.
- A belső `'a list` típushoz hasonló `'a List` listát és vele együtt a `Nil` és a `Cons` *adatkonstruktorokat* például így definiálhatjuk:

```
datatype 'a List = Nil | Cons of 'a * 'a List
```

- A `Cons` *adatkonstruktorfüggvény* alkalmazásával elég körülményes a listák létrehozása. Az 1, 2, 3, 4 sorozatot például így kell megadni:

```
Cons(1, Cons(2, Cons(3, Cons(4, Nil))))
```

- Bevezethetjük az *infix* pozíciójú `:::` *adatkonstruktoroperátort*:

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

- A *hatospontot* közvetlenül a típusdeklarációban is definiálhatjuk:

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

Polimorf adattípusok: megkülönböztetett egyesítés

- Következő példánk két típus *megkülönböztetett egyesítése*, más néven diszjunkt uniója:

```
datatype ('a, 'b) disun = In1 of 'a | In2 of 'b
```

- Itt három dolgot definiáltunk:

1. a kétargumentumú `disun` típusoperátort,
2. az `In1` : `'a -> ('a, 'b) disun` és
3. az `In2` : `'b -> ('a, 'b) disun` adatkonstruktorfüggvényeket.

- `('a, 'b) disun` az `'a` és `'b` típusok megkülönböztetett egyesítése. *Megkülönböztetettnek* nevezzük az egyesítést, mert később is bármikor meg tudjuk mondani, hogy egy `('a, 'b) disun` típusú pár egyik vagy másik eleme melyik alaptípusból származik. Az új típusba tartozó értékek `In1 x` alakúak, ha `x 'a` típusú, és `In2 y` alakúak, ha `y 'b` típusú.
- Az `In1` és `In2` konstruktorfüggvények olyan *címkének* tekinthetők, amelyek az `'a` típust megkülönböztetik a `'b` típustól.

Megkülönböztetett egyesítés (folyt.)

- A megkülönböztetett egyesítés lehetővé teszi, hogy különböző típusokat használjunk ott, ahol egyébként csak egyetlen típust használhatnánk (vö. objektum-orientált programozás, ahol pl. egy *alakzat* osztálynak *téglalap*, *háromszög* vagy *kör* nevű leszármazottai lehetnek).
- Az SML-ben megkülönböztetett egyesítéssel tudunk létrehozni *különböző típusú elemekből* álló listát:

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

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

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

Megkülönböztetett egyesítés (folyt.)

- Egy példa concat alkalmazására:

```
- concat [In1 "Ó! ", In2 King, In1 "Skócia"];  
> val it = "Ó! Skócia : string
```

- Az In1 konstruktorfüggvény típusa 'a -> ('a, 'b) disun, ezért a string típusú "Ó! " argumentumra alkalmazva (string, 'b) disun típusú érték az eredmény.
- Az In2 konstruktorfüggvény típusa 'b -> ('a, 'b) disun, ezért a person típusú King kifejezésre alkalmazva ('a, person) disun típusú érték az eredmény.
- Az [In1 "Ó!", In2 King, In1 "Skócia"] kifejezésben mindkét alaptípust lekötjük, ezért ennek a listának a típusa: ((string, person) disun) list.
- Az [In2 "Ó", In2 King, In1 "Skócia"] kifejezés kiértékelése hibajelzést eredményez, mert a 'b típusváltozót nem lehet ugyanabban a kifejezésben egyszer így, másszor úgy lekötni.

ESETSZÉTVÁLASZTÁS, OPCIONÁLIS ÉRTÉK

Esetszétválasztás (case)

```
case E of P1 => E1 | P2 => E2 | ... | Pn => En
```

Az SML-értelmező – balról jobbra és fölülről lefelé haladva – megpróbálja E-t P1-re illeszteni, ha nem sikerül, P2-re s.í.t. A case-kifejezés eredménye az E kifejezésre illeszkedő első P_i mintához tartozó E_i kifejezés lesz.

A case is csak szintaktikus édesítőszert, ui. helyettesíthető fn-jelöléssel:

```
(fn P1 => E1 | P2 => E2 | ... | Pn => En) E
```

Például a lady függvényt így is definiálhattuk volna:

<pre>datatype degree = Duke Marquis Earl Viscount Baron (* lady : degree -> string lady p = p főnemes hitvesének rangja *) fun lady p = case p of Duke => "Duchess " Marquis => "Marchioness" Earl => "Countess" Viscount => "Viscountess" Baron => "Baroness"</pre>	<pre>(* lady : degree -> string lady p = p főnemes hitvesének rangja *) fun lady p = (fn Duke => "Duchess " Marquis => "Marchioness" Earl => "Countess" Viscount => "Viscountess" Baron => "Baroness") p</pre>
---	--

Opcionális érték kezelése ('a option)

```
datatype 'a option = NONE | SOME of 'a
```

Függvények az Option könyvtárból:

```
val getOption      : 'a option * 'a -> 'a
val isSome        : 'a option -> bool
val valOf         : 'a option -> 'a
val filter        : ('a -> bool) -> 'a -> 'a option
val map           : ('a -> 'b) -> 'a option -> 'b option
val mapPartial    : ('a -> 'b option) -> ('a option -> 'b option)
```

getOption (*xopt*, *d*) = *x* if *xopt* is SOME *x*, *d* otherwise.

isSome *xopt* = true if *xopt* is SOME *x*, false otherwise.

valOf *xopt* = *x* if *xopt* is SOME *x*, raises Option otherwise.

filter *p* *x* = SOME *x* if *p* *x* is true, NONE otherwise.

map *f* *xopt* = SOME(*f* *x*) if *xopt* is SOME *x*, NONE otherwise.

mapPartial *f* *xopt* = *f* *x* if *xopt* is SOME *x*, NONE otherwise.

Példák opcionális értékek kezelésére

- Egészlista legnagyobb elemének kiválasztása

Üres listának nincs legnagyobb eleme; egyelemű lista egyetlen eleme a „legnagyobb”; legalább kételemű lista legnagyobb eleme az első elem és a maradéklista elemei közül a legnagyobb.

```
(* maxl : int list -> int option
   maxl ns = az ns egészlista legnagyobb eleme *)
fun maxl []      = NONE      (* üres *)
  | maxl [n]     = SOME n    (* egyelemű *)
  | maxl (n::ns) =           (* legalább kételemű *)
    SOME(Int.max(n, valOf(maxl ns)))
```

- Füzer elején álló karaktersorozat átalakítása egész számmá

```
val Int.fromString : string -> int option (* Overflow *)
```

```
Int.fromString s = SOME i if a decimal integer numeral can be scanned
  from a prefix of string s, ignoring any initial whitespace;
  NONE otherwise. A decimal integer numeral, after any initial
  whitespace, must have the form: [+~-]?[0-9]+
```

```
Int.fromString "1234"; Int.fromString "-1234"; Int.fromString "~1234";
Int.fromString "+1234"; Int.fromString "+007"; Int.fromString "alma"
```

LOKÁLIS ÉRVÉNYŰ DEKLARÁCIÓK

Deklaráció lokális érvényű deklarációval: `local`-deklaráció

- Ún. `local`-deklarációt használunk, ha egyes deklarációkat fel akarunk használni más deklarációkban, miközben *el akarjuk rejtetni* őket a program többi része előtt.

- Szintaxisa:


```
local d1      ahol d1 egy nemüres deklarációsorozat,
in d2                d2 egy másik nemüres deklarációsorozat.
end
```

- Példa:

```
(* length : 'a list -> int
   length zs = a zs lista hossza
*)
local
  (* len : 'a list * int -> int
     len (zs, n) = az n és a zs lista hosszának összege
  *)
  fun len ([], n)      = n
    | len (_::zs, n) = len(zs, n+1)
in
  fun length zs = len(zs, 0)
end
```

BINÁRIS FÁK

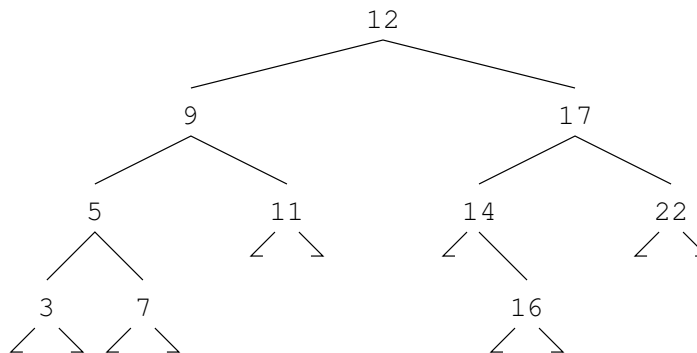


Bináris fák datatype deklarációval

- A listához hasonlóan rekurzív adatszerkezet a *fa*.
- Először olyan bináris fát deklarálunk, amelynek a levelei üresek, a csomópontjaiban pedig előbb a bal részfát, majd az 'a típusú értéket, és végül a jobb részfát adjuk meg:

```
datatype 'a tree = L | B of 'a tree * 'a * 'a tree
```

- Tekintsük például az alábbi fát:



- Az 'a tree adattípus L és B adatkonstruktoraival ez a fa pl. a következő lapon látható módon írható le.

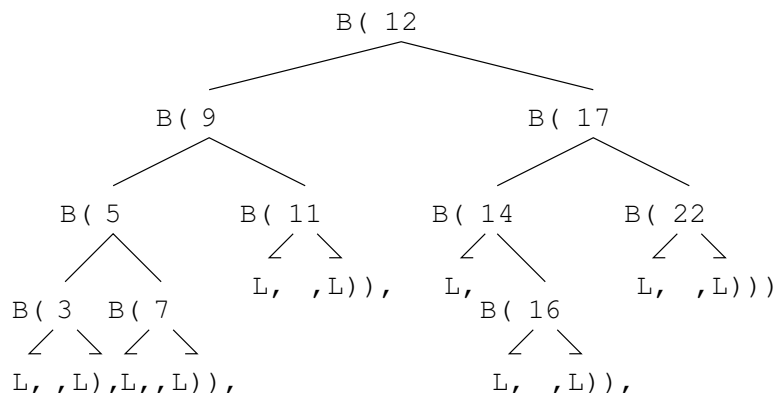
Bináris fák datatype deklarációval (folyt.)

```

B ( B ( B ( B ( L , 3 , L ) ,
          5 ,
          B ( L , 7 , L )
        ) ,
        9 ,
        B ( L , 11 , L )
      ) ,
      12 ,
      B ( B ( L ,
              14 ,
              B ( L , 16 , L )
            ) ,
            17 ,
            B ( L , 22 , L )
          )
    )
  )

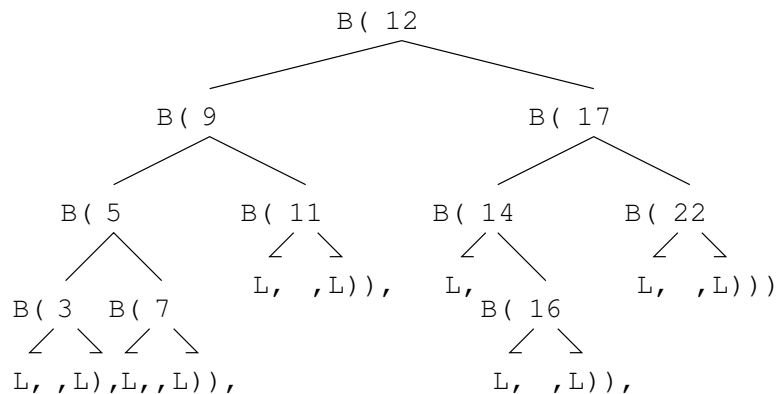
```

A bal oldali kifejezést elég nehéz átlátni. A fastruktúra szöveges leírását megkönnyíti, ha az ábrába beírjuk a megfelelő adatkonstruktorokat.



Bináris fák datatype deklarációval (folyt.)

- A fastruktúra szöveges leírása átláthatóbb, ha az egyes részfáknak nevet adunk, és a részfákból építjük fel a teljes fát:



```

val tr3  = B(L, 3, L);
val tr5  = B(tr3, 5, tr7);
val tr9  = B(tr5, 9, tr11);
val tr14 = B(L, 14, tr16);
val tr17 = B(tr14, 17, tr22);

val tr7  = B(L, 7, L);
val tr11 = B(L, 11, L);
val tr16 = B(L, 16, L);
val tr22 = B(L, 22, L);
val tr12 = B(tr9, 12, tr17)
  
```

Bináris fák datatype deklarációval (folyt.)

- Másféle fastruktúrákat is deklarálhatunk, pl.
 - kezdhethetjük az 'a típusú értékkel, majd folytathatjuk előbb a bal, azután a jobb részfa megadásával,
 - felhasználhatjuk a levelet is értékek tárolására,
 - az értéket nem tároló üres csonkokat pedig E-vel jelölhetjük.

- A leírtak szerinti bináris fát hoz létre a következő deklaráció:

```
datatype 'a tree = E | L of 'a | B of 'a * 'a tree * 'a tree
```

- A rekurzív függvényekhez hasonlóan a rekurzív adattípusok deklarációjában is kell lennie nemrekurzív ágak (ún. triviális esetnek).
- A nemrekurzív ág hiánya miatt az alábbi, szintaktikailag helyes deklarációk használhatatlanok:

```
datatype 'a badtree = B of 'a badtree * 'a * 'a badtree
```

```
datatype 'a badtree = L of 'a badtree
```

```
      | B of 'a badtree * 'a * 'a badtree
```

Egyszerű műveletek bináris fákon

- nodes egy fa csomópontjait számlálja meg. Legyen

```
datatype 'a tree = L | N of 'a * 'a tree * 'a tree

(* nodes : 'a tree -> int
   nodes f = az f fa csomópontjainak a száma *)
fun nodes (N(_, t1, t2)) = 1 + nodes t2 + nodes t1
  | nodes L = 0
```

- nodes akkumulátort használó változata (nodesa):

```
fun nodesa f =
  let (* nodes0(f, n) = n + a csomópontok száma f-ben
       nodes0 : 'a tree * int -> int *)
      fun nodes0 (N(_, t1, t2), n) =
          nodes0(t1, nodes0(t2, n+1))
        | nodes0 (L, n) = n
      in nodes0(f, 0)
      end
```

Egyszerű műveletek bináris fákon (folyt.)

- A fa gyökeréből a leveléhez vezető úton az élek számát (az út hosszát) az adott levél szintjének is nevezzük. A szintek közül a legnagyobbat a fa *mélységének* hívjuk.
- `depth` egy fa mélységét határozza meg.

```
(* depth : 'a tree -> int
   depth f = az f fa mélysége *)
fun depth (N(_, t1, t2)) = 1 + Int.max(depth t2, depth t1)
  | depth L = 0
```

- `depth` akkumulátort használó változata (`deptha`):

```
fun deptha f = let fun depth0 (N(_, t1, t2), d) =
                    Int.max(depth0(t1, d+1), depth0(t2, d+1))
                    | depth0 (L, d) = d
                in
                    depth0(f, 0)
                end
```

Egyszerű műveletek bináris fákon (folyt.)

- `fulltree` n mélységű teljes bináris fát épít, és a fa csomópontjait 1-től $2^n - 1$ -ig beszámozza. Egy teljes bináris fában minden csomópontból pontosan két él indul ki, és minden levelének ugyanaz a szintje.

```
(* fulltree : int -> 'a tree
   fulltree n = n mélységű teljes fa *)
fun fulltree n =
  let fun ftree (_, 0) = L
        | ftree (k, n) = N(k, ftree(2*k, n-1), ftree(2*k+1, n-1))
      in
    ftree(1, n)
  end
```

- `reflect` a fát a függőleges tengelye mentén tükrözi.

```
(* reflect : 'a tree -> 'a tree
   reflect t = a függőleges tengelye mentén tükrözött t fa *)
fun reflect L = L
  | reflect (N(v,t1,t2)) = N(v, reflect t2, reflect t1)
```

Lista előállítás bináris fa elemeiből

- Mindhárom függvény *bináris fából listát* állít elő. Abban különböznek egymástól, hogy a csomópontokban tárolt értékeket mikor veszik ki, és milyen sorrendben járják be a részfákat:
 - `preorder` először az értéket veszi ki, majd bejárja a bal, és azután a jobb részfát;
 - `inorder` először bejárja a bal részfát, majd kiveszi az értéket, végül bejárja a jobb részfát;
 - `postorder` először bejárja a bal, majd a jobb részfát, és utoljára veszi ki az értéket.
- Az akkumulátort nem használó változatok egyszerűek, érthetőek, de nem elég hatékonyak a @ operátor használata miatt.

```
(* preorder : 'a tree -> 'a list
   preorder f = az f fa elemeinek preorder sorrendű listája *)
fun preorder L = []
  | preorder (N(v,t1,t2)) = v :: preorder t1 @ preorder t2
(* inorder : 'a tree -> 'a list
   inorder f = az f fa elemeinek inorder sorrendű listája *)
fun inorder L = []
  | inorder (N(v,t1,t2)) = inorder t1 @ (v :: inorder t2)
(* postorder : 'a tree -> 'a list
   postorder f = az f fa elemeinek postorder sorrendű listája *)
fun postorder L = []
  | postorder (N(v,t1,t2)) = postorder t1 @ (postorder t2 @ [v])
```

Lista előállítás bináris fa elemeiből (folyt.)

- Ha `inorder` előző változatában az `inorder t1 @ (v :: inorder t2)` kifejezésben a `v :: inorder t2` részkifejezést nem tesszük zárójelbe, a fordító hibát jelez, mivel `::` és `@` azonos precedenciájú, és ezért zárójelek nélkül a nyilvánvalóan hibás `inorder t1 @ v` részkifejezést akarná kiértékelni.
- `inorder` előző megvalósításával kb. egyenértékű a következő változata, amelyben a `v` elem helyett az egyelemű `[v]` listát fűzzük `inorder t2` elé:

```
fun inorder L = []
  | inorder (N(v,t1,t2)) = inorder t1 @ ([v] @ inorder t2)
```

Ez a változat azonban *roppant sérülékeny*, ugyanis a hatékonysága függ a zárójelek kirakásától. Ha a `[v] @ inorder t2` részkifejezést nem tesszük zárójelbe, akkor a fordító először a `inorder t1 @ [v]` részkifejezést fogja kiértékelni, azaz egy egyelemű listához fűz egy (általában) jóval hosszabbat!

- Az elmondottakhoz hasonló okból `postorder` bemutatott változata is *rendkívül sérülékeny!* Ha ugyanis a `postorder t1 @ (postorder t2 @ [v])` kifejezésben az amúgyis rossz hatékonyságú `postorder t2 @ [v]` részkifejezést nem tesszük zárójelbe, akkor a fordító először a `postorder t1 @ postorder t2` részkifejezést értékeli ki, azaz a két, feltehetően hosszú listát fűzi egybe, majd a létrehozott eredménylistát fűzi az egyelemű listához!

Lista előállítása bináris fa elemeiből (folyt.)

Az akkumulátort használó változatok nehezebben érthetőek meg, de *hatékonyabbak*, elsősorban a veremhasználat szempontjából.

```
(* preord : 'a tree * 'a list -> 'a list
   preord(f, vs) = az f fa elemeinek a vs lista elé fűzött,
                   preorder sorrendű listája *)
fun preord (L, vs) = vs
  | preord (N(v,t1,t2), vs) = v::preord(t1, preord(t2,vs))

(* inord : 'a tree * 'a list -> 'a list
   inord(f, vs) = az f fa elemeinek a vs lista elé fűzött,
                   inorder sorrendű listája *)
fun inord (N(v,t1,t2), vs) = inord(t1, v::inord(t2,vs))
  | inord (L, vs) = vs

(* postord : 'a tree * 'a list -> 'a list
   postord(f, vs) = az f fa elemeinek a vs lista elé fűzött,
                    postorder sorrendű listája *)
fun postord (N(v,t1,t2), vs) = postord(t1, postord(t2, v::vs))
  | postord (L, vs) = vs
```


Bináris fa előállítás a lista elemeiből: `balPreorder`

- Listát *kiegyensúlyozott* (**balanced**) *bináris fává* alakítanak a következő függvények: `balPreorder`, `balInorder` és `balPostorder`; a különbség közöttük most is a bejárási sorrendben van.
- (* `balPreorder: 'a list -> 'a tree`
`balPreorder xs = az xs lista elemeiből álló, preorder`
`bejárású, kiegyensúlyozott fa`
 *)

```

fun balPreorder [] = L
  | balPreorder (x::xs) =
    let val k = length xs div 2
    in
      N(x, balPreorder(List.take(xs, k)),
        balPreorder(List.drop(xs, k)))
    end
  
```
- A hatékonyságot kisebb mértékben rontja, hogy `List.take` és `List.drop` egymástól függetlenül *kétszer* mennek végig a lista első felén.

take és drop egyetlen függvénnyel: take'ndrop

- Írjunk take'ndrop néven olyan függvényt, amelynek egy xs listából és egy k egészből álló pár az argumentuma, és egy olyan pár az eredménye, amelynek első tagja a lista első k db eleme, második tagja pedig a lista többi eleme.

```
(* take'ndrop : 'a list * int -> 'a list * 'a list
   take'ndrop(xs, k) = olyan pár, amelynek
                       első tagja xs első k db eleme,
                       második tagja pedig xs maradéka
```

*)

```
fun take'ndrop (xs, k) =
    let fun td (xs, 0, ts) = (rev ts, xs)
        | td ([], _, ts) = (rev ts, [])
        | td (x::xs, k, ts) = td(xs, k-1, x::ts)
    in
        td(xs, k, [])
    end
```

- take'ndrop felhasználása, nevezetesen az eredményül átadott pár miatt módosítani kell balpreorder felépítésén.

Bináris fa előállítás a lista elemeiből: `balPreorder`, újra

- Ez volt:

```
fun balPreorder [] = L
  | balPreorder (x::xs) =
    let val k = length xs div 2
    in  N(x, balPreorder(List.take(xs, k)),
        balPreorder(List.drop(xs, k)))
    end
```

- Ez lett:

```
(* balPreorder: 'a list -> 'a tree
   balPreorder xs = az xs lista elemeiből álló, preorder ... *)
fun balPreorder [] = L
  | balPreorder (x::xs) =
    let val k = length xs div 2
        val (ts, ds) = take'ndrop(xs, k)
    in  N(x, balPreorder ts, balPreorder ds)
    end
```

Bináris fa előállítás a lista elemeiből

- (* balInorder: 'a list -> 'a tree
 balInorder xs = az xs lista elemeiből álló, inorder bejárású,
 kiegyensúlyozott fa
 *)

```

fun balInorder [] = L
  | balInorder (x::xs) =
    let val k = length xs div 2
        val ys = List.drop(x, k)
    in
      N(hd xs, balInorder(List.take(x, k)),
        balInorder(ys))
    end

```
- (* balPostorder: 'a list -> 'a tree
 balPostorder xs = az xs lista elemeiből álló, postorder
 bejárású, kiegyensúlyozott fa
 *)

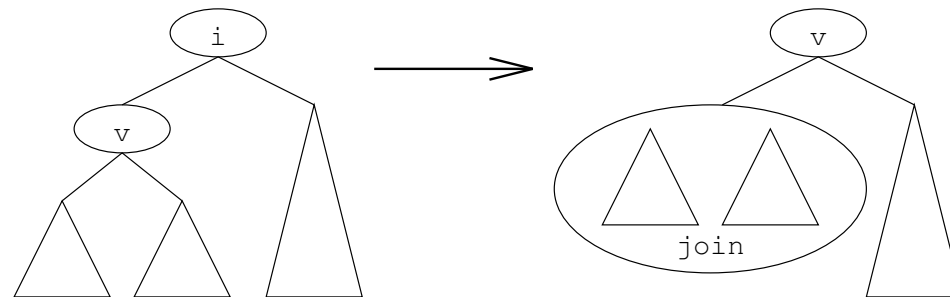
```

fun balPostorder xs = balPreorder(rev xs)

```
- balInorder take'ndrop-pal való definiálását meghagyjuk gyakorló feladatnak.

Elem törlése bináris fából

- Adott értékű *elemet* rekurzív módszerrel *megkeresni* egyszerű feladat.
- Új *elemet beszúrni* sem nehéz: rekurzív módszerrel keresünk egy levelet, és ennek a helyére berakjuk az új értéket. Ha a fa rendezve van, ügyelnünk kell arra, hogy a rendezettség megmaradjon.
- Adott értékű *elemet* vagy *elemeket* rekurzív módszerrel *kitörölni* valamivel nehezebb: ha a törlendő érték az éppen vizsgált részfa gyökerében van, a két részre szétcsúszó fa részfáit *egyesíteni* kell, miután a törlést a két részfán már végrehajtottuk.



- Megtehetjük, hogy előbb egyesítjük a két részfát, majd az eredményül kapott fából töröljük az adott értékű elemet.

Elem rekurzív törlése bináris fából (folyt.)

- A `join`-nal egyesítjük a törlés hatására létrejövő két részfat: a bal részfat lebontja, és közben az elemeit egyesével berakja a jobb részfatba.

```
(* join : 'a tree * 'a tree -> 'a tree
   join(b, j) = a b és a j fák egyesítésével létrehozott fa *)
fun join (L, tr) = tr
  | join (N(v, lt, rt), tr) = N(v, join(lt, rt), tr)
```

- A `remove` rendezetlen bináris fából törli az `i` értékű elem összes előfordulását.

```
(* remove : 'a * 'a tree -> 'a tree
   remove(i, f) = i összes előfordulását törli f-ből *)
fun remove (i, L) = L
  | remove (i, N(v, lt, rt)) =
    if i<>v
    then N(v, remove(i, lt), remove(i, rt))
    else join(remove(i, lt), remove(i, rt))
```

Bináris keresőfák: `blookup`, `binsert`

- Rendszerint adott kulcsú elemet keresünk egy rendezett bináris fában, ehhez értékeket kell összehasonlítanunk egymással, ehhez a keresett kulcsnak *egyenlőségi típusúnak* kell lennie (a példában a `string` típust használjuk).
- A függvények *kivételt* jeleznek, ha a keresett kulcsú elem nincs a keresőfában: `exception Bsearch of string`.
- A `blookup` függvény adott kulcshoz tartozó értéket ad vissza:

```
(* blookup : (string * 'a) tree * string -> 'a
   blookup(f, b) = az f fában a b kulcshoz tartozó érték
*)
fun blookup (L, b) = raise Bsearch("LOOKUP: " ^ b)
  | blookup (N((a,x), t1, t2), b) =
    if b < a      then blookup(t1,b)
    else if a < b then blookup(t2, b)
    else x
```

Bináris keresőfák: `bupdate`

- A `binsert` függvény egy új kulcsú elemet rak be egy rendezett bináris fába, ha még nincs benne:

```
(* binsert : (string * 'a) tree * (string * 'a) -> (string * 'a) tree
   binsert(f, (b,y)) = az új (b,y) kulcs-érték párral bővített f fa *)
fun binsert (L, (b,y)) = N((b,y), L, L)
  | binsert (N((a, x), t1, t2), (b,y)) =
    if b < a      then N((a, x), binsert(t1, (b,y)), t2)
    else if a < b then N((a, x), t1, binsert(t2, (b,y)))
    else (* a=b *) raise Bsearch("INSERT: " ^ b)
```

- A `bupdate` függvény meglévő kulcsú elembe új értéket ír be egy rendezett bináris fában:

```
(* bupdate : (string * 'a) tree * (string * 'a) -> (string * 'a) tree
   bupdate(f, (b,y)) = az f fa, a b kulcshoz tartozó érték helyén
                       az y értékkel *)
fun bupdate (L, (b,y)) = raise Bsearch("UPDATE: " ^ b)
  | bupdate (N((a,x), t1, t2), (b,y)) =
    if b < a      then N((a,x), bupdate(t1, (b,y)), t2)
    else if a < b then N((a,x), t1, bupdate(t2, (b,y)))
    else (* a=b *) N((b,y), t1, t2)
```

- A függvények *generikussá* tételét meghagyjuk gyakorló feladatnak.