HALMAZMŰVELETEK

Halmazműveletek: „benne van-e?” (`isMem`) és „ha új, tedd bele” (`newMem`)

- `isMem` igaz értéket ad eredményül, ha a keresett elem benne van a listában.

```
(* isMem : ''a * ''a list -> bool
   isMem(x, ys) = x eleme-e ys-nek
*)
fun isMem (_, []) = false
  | isMem (x, y::ys) = x = y orelse isMem(x, ys)

infix isMem
```

- `newMem` egy új elemet rak be egy listába, ha még nincs benne.

```
(* newMem : ''a * ''a list -> ''a list
   newMem(x, xs) = [x] és xs listaként ábrázolt uniója
*)
fun newMem (x, xs) = if x isMem xs
                     then xs
                     else x::xs
```

`newMem`, ha a sorrendtől eltekintünk, halmazt hoz létre.

Halmazműveletek: „listából halmaz” (`setof`)

- `setof` halmazt készít egy listából úgy, hogy kiszedi belőle az ismétlődő elemeket. Rossz hatékonyságú.

```
(* setof : ''a list -> ''a list
   setof xs = xs elemeinek listaként ábrázolt halmaza
*)
fun setof []      = []
  | setof (x::xs) = newMem(x, setof xs)
```

- Öt halmazműveletet definiálunk:
  - unió (`union`, $S \cup T$),
  - metszet (`inter`, $S \cap T$),
  - részhalmaza-e (`isSubset`, $T \subseteq S$),
  - egyenlők-e (`isSetEq`, $S = T$),
  - hatványhalmaz (`powerSet`, pS).

Halmazműveletek: „unió” (`union`) és „metszet” (`inter`)

- Listaként kezeljük a halmazokat, később hatékonyabb ábrázolást választhatunk, pl. rendezett listát vagy bináris fát.
- Két halmaz uniója

```
(* union : ''a list * ''a list -> ''a list
   union(xs, ys) = az xs és ys elemeiből álló halmazok uniója
*)
fun union ([], ys)    = ys
  | union (x::xs, ys) = newMem(x, union(xs, ys))
```

- Két halmaz metszete

```
(* inter : ''a list * ''a list -> ''a list
   inter(xs, ys) = az xs és ys elemeiből álló halmazok metszete
*)
fun inter ([], _)     = []
  | inter (x::xs, ys) = let val zs = inter(xs, ys)
                        in
                           if x isMem ys then x::zs else zs
                        end
```

Halmazműveletek: „részhalmaza-e” (`isSubset`) és „egyenlők-e” (`isSetEq`)

- Részhalmaza-e egy halmaz egy másiknak?

```
(* isSubset : ''a list * ''a list -> bool
   isSubset (xs, ys) = az xs elemeiből álló halmaz részhalmaza-e
                       az ys elemeiből álló halmaznak
*)
fun isSubset ([], _)     = true
  | isSubset (x::xs, ys) = (x isMem ys) andalso isSubset(xs, ys)

infix isSubset
```

- Két halmaz egyenlősége (a listák egyenlőségvizsgálata beépített művelet az SML-ben, halmazokra mégsem használható, mert pl. `[3, 4]` és `[4, 3, 4]` listaként ugyan különböznek, de halmazként egyenlők)

```
(* isSetEq : ''a list * ''a list -> bool
   isSetEq(xs, ys) = az xs elemeiből álló halmaz egyenlő-e
                     az ys elemeiből álló halmazzal
*)
fun isSetEq (xs, ys) = (xs isSubset ys) andalso (ys isSubset xs)
```


Halmazműveletek: „halmaz hatványhalmaza” (`powerSet`)

- Az S halmaz hatványhalmaza *összes* részhalmazának a halmaza, az S-t és a $\{\}$-t is beleértve.
- S hatványhalmaza úgy állítható elő, hogy kivesszük S-ből az x elemet, majd *rekurzív módon* előállítjuk az $S - \{x\}$ hatványhalmazát.
- Ha tetszőleges T halmazra $T \subseteq S - \{x\}$, akkor $T \subseteq S$ és $T \cup \{x\} \subseteq S$, így mind T, mind $T \cup \{x\}$ eleme S hatványhalmazának.
- Miközben a fenti elvet rekurzív módon alkalmazzuk, tehát fölsoroltatjuk az $S - \{x\}$ stb. részhalmazait, gyűjtjük a *már kiválasztott* elemeket. Egy-egy rekurzív lépésben a gyűjtő vagy változatlan (T), vagy kiegészül az x elemmel ($T \cup \{x\}$).
- A `pws` függvényben a `base` argumentumban gyűjtjük a halmaz *már kiválasztott* elemeit; kezdetben üres.
- pws(xs, base) = $\{S \cup$ `base` $\mid S \subseteq$ `xs`$\}$, azaz `xs` $\cup$ `base` azon részhalmazainak a listája, amelyek teljes egészében tartalmazzák a base halmazt.

Halmazműveletek: „halmaz hatványhalmaza” (`powerSet`)

- Ezzel a `pws` függvény:

```
(* pws : 'a list * 'a list -> 'a list list
   pws(xs, base) = mindazon halmazok listája, amelyek előállnak xs egy
                   részhalmazának és a base halmaznak az uniójaként
*)
fun pws ([], base) = [base]
  | pws (x::xs, base) = pws(xs, base) @ pws(xs, x::base)
```

- `pws(xs, base)` valósítja meg az $S - \{x\}$ rekurzív hívást (hiszen `x::xs` felel meg S-nek), azaz állítja elő az összes olyan halmazt, amelyekben `x` nincs benne.
- `pws(xs, x::base)` rekurzív módon `base`-ben gyűjti az `x` elemeket, vagyis előállítja az összes olyan halmazt, amelyben `x` benne van.
- `powerSet`-nek már csak megfelelő módon hívnia kell `pws`-t:

```
(* powerSet : 'a list -> 'a list list
   powerSet xs = az xs halmaz hatványhalmaza
*)
fun powerSet xs = pws(xs, [])
```


KIFEJEZÉSEK KIÉRTÉKELÉSE

Mohó kiértékelés: faktoriális kiszámítása rekurzióval

- A faktoriális matematikai definíciója és megvalósítása SML-ben
  $\text{fac } 0 = 1; \text{fac } n = n * \text{fac } (n-1), n > 0$

```
(* fac : int -> int       (-- fontos a klózok sorrendje! --)
   fac n = n!
   PRE n >= 0 *)
fun fac 0 = 1
  | fac n = n * fac(n-1)
```

- `fac` mohó kiértékelése $n = 4$ esetén (egyes triviális lépéseket elhagyunk).

`fac 4` $\rightarrow$ `4 * fac (4-1)` $\rightarrow$ `4 * fac 3` $\rightarrow$ `4 * (3 * fac (3-1))` $\rightarrow$
$\rightarrow$ `4 * (3 * fac (2))` $\rightarrow \cdots \rightarrow$ `4 * (3 * (2 * (1 * 1)))` $\rightarrow \cdots \rightarrow$ `24`

- A rekurzív kiértékelés követi a matematikai definíciót.
- Rontja a hatékonyságot, hogy a rekurzív végrehajtás során minden részeredményt a veremben tárolni kell.
- Ha a szorzás asszociativitását kihasználjuk, nem kell tárolni az összes tényezőt, csak az aktuális részeredményt.

Faktoriális kiszámítása jobbrekurzióval

- Először egy *akkumulátort* (gyűjtőargumentumot) használó *segédfüggvényt* definiálunk. Vegyük észre, hogy a rekurzív hívás *jobbrekurzív:* eredménye közvetlenül, további műveletek elvégzése nélkül adja a végeredményt.

```
(* faci : int -> int -> int  (-- fontos a klózok sorrendje! --)
   faci n p = p * n!          (-- p az akkumulátor --)
*)
fun faci 0 p = p
  | faci n p = faci (n-1) (n*p)
```

- `faci`-t felhasználjuk az egyparaméteres `fac` függvény definiálására. Az akkumulátornak alkalmas *kezdőértéket* adunk.

```
(* fac : int -> int
   fac n = n!
   PRE n >= 0
*)
fun fac n = faci n 1
```

`faci` mint `fac` lokális függvénye:

```
fun fac n =
  let fun faci 0 p = p
        | faci n p = faci (n-1) (n*p)
  in
      faci n 1
  end
```


Faktoriális kiszámítása jobbrekurzióval (folyt.)

- `fac` nem rekurzív, ezért csak `faci` kiértékelését vizsgáljuk (egyes triviális lépéseket összevonunk).

  A függvény:

```
fun faci 0 p = p
  | faci n p = faci (n-1) (n*p)
```

`faci 4 1` $\rightarrow$ `faci (4-1) (4*1)` $\rightarrow$ `faci 3 4` $\rightarrow$ `faci (3-1) (3*4)` $\rightarrow$
$\rightarrow$ `faci 2 12` $\rightarrow \cdots \rightarrow$ `faci 0 24` $\rightarrow$ `24`

- Kiértékelés közben a `p` *akkumulátor* gyűjti a részeredményt, ezért `faci` tárigénye állandó.
- A jobbrekurziót *terminális rekurziónak* is nevezik (angolul: *tail* vagy *terminal* recursion).
- A jó fordítóprogram felismeri a jobbrekurziót, és hatékony tárgykódot állít elő: az argumentumokat frissíthető lokális változókban tárolja, a rekurziót iterációval helyettesíti.
- A jobbrekurzív függvények feldolgozása tehát *iteratívvá* tehető.
- A jobbrekurzív függvényeket ezért *iteratív* függvényeknek is nevezik.

ÖSSZETETT ADATTÍPUSOK

Ennes és típusa

- Két különböző típusú értékből rekordot vagy párt képezhetünk. Pl.

  `{x = 2, y = 1.0} : {x : int, y : real}` és `(2, 1.0) : (int * real)`

- A pár is csak szintaktikus édesítőszer. Pl.

  `(2, 1.0)` ≡ `{1 = 2, 2 = 1.0}` ≡ `{2 = 1.0, 1 = 2}` ≠ `{1 = 1.0, 2 = 2}`.

  Egy párban a tagok sorrendje meghatározó! Az `1` és a `2` is: *mezőnevek*.

- Rekordot kettőnél több értékből is összeállíthatunk. Pl.

  `{nev = "Bea", tel = 3192144, kor = 19} : {kor : int, nev : string, tel : int}`

  Egy hasonló rekord egészszám-mezőnevekkel:

  `{1 = "Bea", 3 = 3192144, 2 = 19}: {1 : string, 2 : int, 3 : int}`

  Az *utóbbi* azonos az alábbi *ennessel* (n-tuple):

  `("Bea", 19, 3192144) : (string * int * int)`

  azaz

  `(string * int * int)` ≡ `{1 = string, 2 = int, 3 = int}`

- Egy rekordban a tagok sorrendje közömbös, a tagokat a mezőnév azonosítja. Egy ennesben a tagok sorrendje nem közömbös, a tagokat a *pozícionális* mezőnév azonosítja.

Példa: Fibonacci-számok

- Ennes lehet függvény argumentuma és eredménye, összetett adat eleme stb.
- A Fibonacci-számok definíciója: $F_0 = 0; F_1 = 1; F_n = F_{n-2} + F_{n-1}, n > 1$.
- Naív megvalósítása:

```
fun fib 0 = 0 | fib 1 = 1 | fib n = fib(n-2) + fib(n-1)
```

- Megvalósítása iterációval:

```
local
  (* iterfib(n, (prev, curr)) = a (prev, curr) Fibonacci-számpárt követő
                                 n-edik Fibonacci-szám (n > 0)
     iterfib : int * (int * int) -> int
  *)
  fun iterfib (1, (prev, curr)) = curr
    | iterfib (n, (prev, curr)) = iterfib(n - 1, (curr, prev + curr))
in
  (* fib n = az n-edik Fibonacci-szám
     fib : int -> int
  *)
  fun fib 0 = 0
    | fib n = iterfib(n, (0, 1))
end
```


HALMAZMŰVELETEK

Halmazműveletek: hatványhalmaz hatékonyabban

- `pws` rossz hatékonyságú, mert kétfelé ágazó rekurziót használ. Pl. egy 19 egész számból álló lista hatványhalmazának előállítását nem lehet kivárni. Írjunk hatékonyabb változatot.
- Az `insAll` segédfüggvény egy elemet szúr be egy listákból álló lista minden eleme elé.

```
(* insAll : 'a * 'a list list * 'a list list -> 'a list list
   insAll(x, yss, zss) = az yss lista ys elemeinek zss elé fűzött
                         listája, amelyben minden ys elem elé x van beszúrva *)
fun insAll (x, [], zss) = zss
  | insAll (x, ys::yss, zss) = insAll(x, yss, x::ys::zss)
```

- `powerSet` `insAll`-t használó rekurzív változata

```
fun powerSet [] = [[]]
  | powerSet (x::xs) = let val pws = powerSet xs
                       in  pws @ insAll(x, pws, [])
                       end
```

- `powerSet` `insAll`-t használó iteratív változata

```
fun powerSet [] = [[]]
  | powerSet (x::xs) = let val pws = powerSet xs
                       in insAll(x, pws, pws)
                       end
```