ABSZTRAKCIÓ, ADATTÍPUSOK

Gyenge és erős absztrakció, adattípusok

- Gyenge absztrakció: a név szinonima, az adatszerkezet részei továbbra is hozzáférhetők.
- Erős absztrakció: a név új dolgot (entitást, objektumot, izét) jelöl, az adatszerkezet részeihez csak korlátok között lehet hozzáférni.
- `type`: gyenge absztrakció
  Pl. `type rat = {num : int, den : int}`
  - ◇ Új nevet ad egy típuskifejezésnek (vö. értékdeklaráció).
  - ◇ Segíti a programszöveg megértését.
- `abstype`: erős absztrakció
  - ◇ Új típust hoz létre: név, műveletek, ábrázolás, jelölés.
  - ◇ Van helyette jobb: `datatype` + modulok
- `datatype`: modulok nélkül gyenge, modulokkal erős absztrakció
  Pl. `datatype 'a option = NONE | SOME of 'a`
  - ◇ Új entitást hoz létre.
  - ◇ Rekurzív és polimorf is lehet.

RACIONÁLIS SZÁMOK

Példa: racionális számok

- A racionális számokat *rekordként* ábrázolhatjuk; az új (gyenge) típus neve `rat`.

```
type rat = {num : int, den : int}
```

- Nevet adunk néhány állandónak.

```
val ratZero = {num = 0, den = 1}; val ratOne   = {num = 1, den = 1};
val ratHalf = {num = 1, den = 2}; val ratThird = {num = 1, den = 3}
```

- A `rat` típusú számokat *normalizált* alakban tároljuk, különben pl. $\frac{1}{2}$ és $\frac{2}{4}$ nem lenne egyenlő. A normalizáláshoz szükségünk van a számláló és a nevező legnagyobb közös osztójára (`gcd`). A közös osztó egyik fontos tulajdonsága, hogy $d|n$ és $d|m \Rightarrow d|n \bmod m$.

```
(* gcd : int -> int -> int
   gcd n m = n és m legnagyobb közös osztója
*)
fun gcd n 0 = abs n
  | gcd n m = gcd (abs m) (abs(n mod m))
```

Példa: racionális számok (folyt.)

- `gcd` ún. *részlegesen alkalmazható* függvény. Ha összes argumentumánál kevesebbre alkalmazzuk, *függvényértéket* ad eredményül.
- Sajnos, a `normalize` függvényben `n` és `m` legnagyobb közös osztóját kétszer is kiszámoljuk: később látni fogjuk, hogyan javíthatunk a hatékonyságán.

```
(* normalize : rat -> rat
   normalize r = r normalizált alakban *)
fun normalize {num = n, den = 0} = raise Domain
  | normalize {num = n, den = d} =
                  {num = n div (gcd n d), den = d div (gcd n d)}
```

- Két egészből *konstruktorfüggvénnyel* (`toRat`) érdemes létrehozni a racionális számot, különben a normalizált tárolás követelménye sérülhet.

```
(* toRat : int -> int -> rat
   toRat n d = n nevezőjű és d számlálójú racionális szám,
               normalizált alakban
*)
fun toRat n d = normalize{num = n, den = d}
```


Példa: racionális számok – a négy alapművelet

```
(* **, //, ++, -- : rat * rat -> rat
   r1 ** r2 = az r1 és r2 racionális számok szorzata
   r1 // r2 = az r1 és r2 racionális számok hányadosa
   r1 ++ r2 = az r1 és r2 racionális számok összege
   r1 -- r2 = az r1 és r2 racionális számok különbsége *)
infix 7 ** //; infix 6 ++ --;

fun (r1 : rat) ** (r2 : rat) =
          toRat (#num r1 * #num r2) (#den r1 * #den r2);

fun (r1 : rat) // (r2 : rat) =
          toRat (#num r1 * #den r2) (#num r2 * #den r1);

fun {num= n1, den= d1} ++ {num= n2, den= d2} =
          toRat (n1*d2 + n2*d1) (d1*d2);

fun {num= n1, den= d1} -- {num= n2, den= d2} =
          toRat (n1*d2 - n2*d1) (d1*d2)
```

Példa: racionális számok – relációs műveletek

- Az = és a <> relációt *készen kapjuk:* két összetett érték strukturálisan összehasonlítható, ha az elemeiken az egyenlőségvizsgálat elvégezhető.

```
(* <<, >>, <<=, >>= : rat * rat -> bool
   r1 << r2 = igaz, ha r1 kisebb r2-nél
   r1 >> r2 = igaz, ha r1 nagyobb r2-nél
   r1 <<= r2 = igaz, ha r1 nem nagyobb r2-nél
   r1 >>= r2 = igaz, ha r2 nem nagyobb r1-nél *)
infix 4 << >> <<= >>=;

fun (r1 : rat) << (r2 : rat) =
          #num r1 * #den r2 < #num r2 * #den r1;

fun (r1 : rat) >> (r2 : rat) =
          #num r1 * #den r2 > #num r2 * #den r1;

fun r1 <<= r2 = not(r1 >> r2);

fun r1 >>= r2 = not(r1 << r2)
```


Példa: racionális számok (folyt.)

- A racionális számokon értelmezett <<= és >>= másképpen:

```
val op<<= = not o op>>;    val op>>= = not o op<<
```

- Egy racionális számot füzérré alakítás után írunk ki a képernyőre.

```
(* toString : rat -> string
   toString r = az r racionális szám füzérként (számláló/nevező
                alakban, ha a nevező = 1, egyébként egészként)
*)
fun toString {num, den = 1} = Int.toString num
  | toString {num, den} = Int.toString num ^ "/" ^ Int.toString den
```

- Példák `rat` típusú értékek használatára

```
normalize (toRat 15 3);       toString(toRat 2 3 ** toRat 5 4);
normalize (toRat 15 ~3);      toString(toRat 2 3 // toRat 5 3);
normalize (toRat ~15 3);      toString(toRat 1 4 ++ toRat 3 10);
normalize (toRat ~15 ~3);     toString(toRat 3 10 -- toRat 1 4)
```

Példa: racionális számok (folyt.)

- Példák `rat` típusú értékek használatára (folyt.)

```
toRat 2 3 << toRat 5 4;          toRat 2 3  >> toRat 5 3;
toRat 1 4 << toRat 3 10;         toRat 3 10 >> toRat 1 4

infix 8 /-/; fun n /-/ d = toRat n d;

toString(2/-/3 **  5/-/4);   2/-/3 <<  5/-/4;  1/-/4  <<  3/-/10;
toString(2/-/3 //  5/-/3);   2/-/3 <<  2/-/3;  3/-/10 >>  1/-/4;
toString(1/-/4 ++  3/-/10);  2/-/3 <<= 2/-/3;
toString(3/-/10 -- 1/-/4);   2/-/3 >>  5/-/3;  3/-/10 >>= 3/-/10
```

- Példák `gcd` részleges alkalmazására

```
(* gcd120 : int -> int                                  gcd120 45;
  gcd m = m legnagyobb közös osztója 120-szal           gcd120 48;
*)                                                      gcd120 ~96;
val gcd120 = gcd 120;                                   gcd120 630;
```


FELHASZNÁLÓI ADATTÍPUSOK

A `datatype` deklaráció

- `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 *adatkonstruktora* (röviden: *konstruktora)* 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 p = p megszólítása
   title : person -> string *)
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 ps = az összes Knight nevének listája
   sirs : person list -> string list *)
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:

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

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 (p, r)= igaz, ha p magasabb rangú r-nél
   superior : person * person -> bool *)
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
```


A felsorolásos 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 *felsorolásos típust* létrehozni a `datatype` deklarációval. Pl.

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

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

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

A felsorolásos típus `datatype` deklarációval (folyt.)

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

```
(* lady p = p főnemes hitvesének rangja
   lady : degree -> string *)
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 deklarálhatnánk, ill. definiálhatnánk:

```
datatype Bool = True | False
(* Not b = b negáltja
   Not : Bool -> Bool *)
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 d = a d diszjunkt unió In1 címkéjű
             elemeinek konkatenációja
   concat : (string, 'a) disun list -> string *)
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.

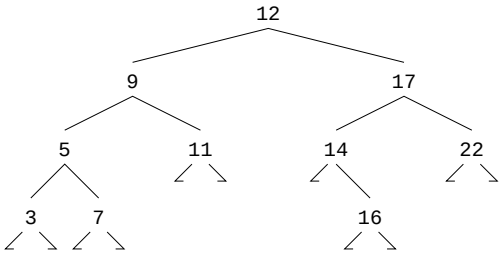
BINÁRIS FÁK

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:

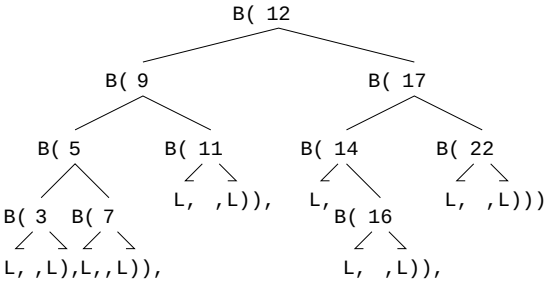


- Az `'a tree` adattípus `L` és `B` adatkonstruktoraival ez a fa pl. a következő lapon látható módon írható le.

Bináris fák `datatype` deklarációval (folyt.)

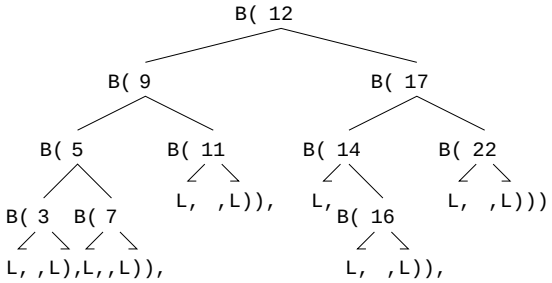
```
B(B(B(B(L,3,L),
        5,
        B(L,7,L)
       ),
      9,
      B(L,11,L)
     ),
   12,
   B(B(L,
       14,
       B(L,16,L)
      ),
     17,
     B(L,22,L)
    )
 )
```

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 tr7  = B(L,7,L);
val tr5  = B(tr3,5,tr7);      val tr11 = B(L,11,L);
val tr9  = B(tr5,9,tr11);     val tr16 = B(L,16,L);
val tr14 = B(L,14,tr16);      val tr22 = B(L,22,L);
val tr17 = B(tr14,17,tr22);   val tr12 = B(tr9,12,tr17)
```


Bináris fák `datatype` deklarációval (folyt.)

- Másféle fastruktúrákat is deklarálhatunk, pl.
  - kezdhetjük 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 ágnak (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 f = az f fa csomópontjainak a száma
   nodes : 'a tree -> int *)
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 f = az f fa mélysége
   depth : 'a tree -> int *)
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
```