ABSZTRAKCIÓ FÜGGVÉNYEKKEL (ELJÁRÁSOKKAL)

Függvény mint visszatérési érték

- A függvényekről mint absztrakciós eszközökről szólva eddig olyan magasabbrendű függvényeket használtunk, amelyeknek más függvények voltak a paraméterei.
- Most olyan magasabbrendű függvényeket mutatunk be, amelyek *függvényt* (pontosabban *függvényértéket)* adnak eredményül.
- A korábban bemutatott *átlagcsillapítás* sokszor használható módszer, ezért érdemes önálló függvényként megírni: ha adott az f függvény, elő kell állítani x és fx átlagát.

```
(* averageDamp f = f valamely x értékre alkalmazva
                   előállítja x és f x átlagát     *)
fun averageDamp f = fn x => (x + f x) / 2.0
```

- Jól látható, hogy `averageDamp`, ha csak egyetlen paraméterre alkalmazzuk, függvényértéket ad eredményül. `averageDamp` részlegesen alkalmazható függvény.
- Példa `averageDamp` alkalmazására:

```
(averageDamp (fn x => x*x)) 10.0; (* 10.0 és 100.0 átlaga *)
```

- A kiértékelés sorrendje miatt a külső zárójelpár el is hagyható:

```
averageDamp (fn x => x*x) 10.0
```


Függvény mint visszatérési érték (folyt.)

- `averageDamp` definíciója felírható (*szintaktikai édesítőszerrel).*

```
fun averageDamp f x = (x + f x) / 2.0;
```

- `sqrt averageDamp`-pel felírt változata explicitté teszi a *fixpontmeghatározás* és az *átlagcsillapítás* módszerét, továbbá az $y = x/y$ *egyenlet használatát.*

```
fun sqrt x = fixedPoint(averageDamp (fn y => x/y), 1.0);
sqrt 4.0;
```

- Tanulság: egy folyamatot sokféle eljárással leírhatunk, de a *lényeget* sokkal könnyebb megérteni, ha *megfelelően megválasztott absztrakciókat* vezetünk be.
- Még egy példa a bemutatottak alkalmazására: az x köbgyöke az $y \mapsto x/y^2$ – SML-jelöléssel az `fn y => x/(y*y)` – függvény fixpontja. A megoldás már kész is van!

```
fun cubeRoot x  = fixedPoint(averageDamp (fn y => x/y/y), 1.0);
cubeRoot 8.0;
```


Függvény mint visszatérési érték (folyt.): az általános Newton-módszer

- Ha $x \mapsto g(x)$ egy differenciálható függvény, akkor a $g(x) = 0$ egyenlet az $x \mapsto f(x)$ függvény egy fixpontja, ahol $f(x) = x - g(x)/g'(x)$ és $g'(x)$ a g x szerinti deriváltja.
- Az *általános Newton-módszer* a fixpontmódszer egy alkalmazása az f függvény egy fixpontjának megtalálására. Számos g függvényre és megfelelően megválasztott x értékre a Newton-módszer gyorsan konvergál.
- Először is azt a `deriv` függvényt kell definiálnunk, amelynek (az `averageDamp` függvényhez hasonlóan) függvény a paramétere, és függvényt ad eredményül.
- Ha g függvény és dx egy kis szám, akkor a g függvény g' deriváltja az a függvény, amelynek értéke bármely x számra a következő: $g'(x) = (g(x + dx) - g(x))/dx$.

```
(* deriv g = g deriváltja
*)
val dx = 0.00001;
fun deriv g = fn x => (g(x+dx) - g x) / dx
```

- Például az $x \mapsto x^3$ függvény deriváltja $x = 5$-re (pontos értéke 75):

```
let fun cube x = x*x*x in deriv cube 5.0 end
```

Függvény mint visszatérési érték (folyt.): a Newton-módszer fixpont-folyamatként

- `deriv` felhasználásával az általános Newton-módszer definiálható *fixpont-folyamatként*:

```
fun newtonTransform g x = x - (g x / deriv g x)
and newtonsMethod g guess = fixedPoint(newtonTransform g, guess)
```

- Példa `newtonsMethod` használatára:

```
fun sqrt x = newtonsMethod (fn y => y*y-x) 1.0;
sqrt 16.0
```

- Két általános módszer egy-egy alkalmazását láttuk egy szám négyzetgyökének kiszámítására: az egyik a fixpont-, a másik a Newton-módszer.
- Mivel az utóbbi is a fixpontmódszeren alapul, valójában a fixpontmódszer kétféle alkalmazását láttuk.
- Mindkét esetben egy függvényből indulunk ki, és kiszámítjuk valamely transzformáltjának egy fixpontját.
- Ezt az általános módszert is definiálhatjuk eljárásként (függvényként), ezt mutatjuk be a következő diákon.

ABSZTRAKCIÓ ADATOKKAL

Függvény mint visszatérési érték (folyt.): a fixpontmódszer kétféle alkalmazása

-
```
(* fixedPointOfTransform g transform guess =
     a fixed point of (transform g) with the initial guess guess
*)
fun fixedPointOfTransform (g, transform, guess) =
                                  fixedPoint(transform g, guess)
```

- Ez volt `sqrt` fixpontkeresésen alapuló első változata:

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

- Átírva az általános módszert megvalósító függvénnyel:

```
fun sqrt x = fixedPointOfTransform (fn y => x/y,
                                    averageDamp, 1.0)
```

- Ez volt `sqrt` Newton általános módszerét használó második változata:

```
fun sqrt x = newtonsMethod (fn y => y*y-x) 1.0;
```

- Átírva az általános módszert megvalósító függvénnyel:

```
fun sqrt x = fixedPointOfTransform (fn y => y*y-x,
                                    newtonTransform, 1.0)
```


Adatabsztrakció: racionális számok

- A következő előadásokon összetett adatokkal és adatabsztrakcióval foglalkozunk.
- Az adatabsztrakció lényege: összetett adatokkal dolgozó programjainkat úgy építjük föl, hogy
  - az adatokat felhasználó programrészek az adatok szerkezetéről ne tételezzenek fel semmit, csak az előre definiált műveleteket használják,
  - az adatokat definiáló programrészek az adatokat felhasználó programrészektől függetlenek legyenek.
  - A program e két része közötti interfész *konstrukturokból* és *szelektorokból* áll.
- Az összetett adatok közül eddig ennesekkel és listákkal találkoztunk.
- Első nagyobb példánkban a racionális számok és a rajtuk végezhető műveletek megvalósítását mutatjuk be.
- A racionális számot ábrázolhatjuk egy olyan párral, amelynek az első tagja a számláló *(numerator)* és a második a nevező *(denominator)*.
- Megvalósítjuk a négy aritmetikai alapműveletet: `addRat`, `subRat`, `mulRat`, `divRat`, továbbá az egyenlőségvizsgálatot: `equRat`.

Adatabsztrakció: racionális számok (folyt.)

- Tegyük föl, hogy
  - van olyan *konstruktorműveletünk,* amely egy `n` számlálóból és egy `d` nevezőből létrehozza a racionális számot: `makeRat(n,d)`, továbbá
  - van egy-egy olyan *szelektorműveletünk*, amelyek egy `q` racionális szám számlálóját, ill. nevezőjét előállítják: `num q`, `den q`.
- A jól ismert műveleteket írjuk át SML-programmá:
  $n_1/d_1 + n_2/d_2 = (n_1d_2 + n_2d_1)/(d_1d_2)$, $n_1/d_1 - n_2/d_2 = (n_1d_2 - n_2d_1)/(d_1d_2)$,
  $(n_1/d_1)(n_2/d_2) = (n_1n_2)/(d_1d_2)$, $(n_1/d_1)/(n_2/d_2) = (n_1d_2)/(d_1n_2)$,
  $n_1/d_1 = n_2/d_2$ akkor és csak akkor, ha $n_1d_2 = n_2d_1$.

```
fun addRat(x, y) =
   makeRat(num x * den y + num y * den x, den x * den y)
fun subRat(x, y) =
   makeRat(num x * den y - num y * den x, den x * den y)
fun mulRat(x, y) = makeRat(num x * num y, den x * den y)
fun divRat(x, y) = makeRat(num x * den y, den x * num y)
fun equRat(x, y) = num x * den y = den x * num y
```


Adatabsztrakció: racionális számok (folyt.)

- Az SML-ben az *ennes* létrehozására van *konstruktorműveletünk:* a tagokat kerek zárójelek között, vesszővel elválasztva felsoroljuk, és
- van az *ennes* egy-egy tagját kiválasztó *szelektorműveletünk:* `# i`, ahol `i` az i-edik tag *pozicionális címkéje,* 1-től kezdve.
- Példák: `(3, 4); #1(3, 4); #2(3, 4);`
- Az *ennes* tagjai *mintaillesztéssel* is köthetők névhez, pl. `val (n, d) = (3, 4).`
- *Gyenge* absztrakcióval valósítjuk meg a racionális szám típusát, a konstruktort és szelektorokat:

```
type rat = int * int;
fun makeRat (n, d) = (n, d) : rat;
fun num (q : rat) = #1 q;
fun den q = #2 (q : rat);
```

- A *gyenge absztrakció* nevet ad egy objektumnak, de *nem rejti el* a megvalósítás részleteit.
- Szükségünk lesz kiíróműveletre is az `n/d` alakú racionális szám kiírásához.

```
fun printRat q =
      print(makestring(num q) ^ "/" ^ makestring(den q) ^ "\n");
```


Adatabsztrakció: racionális számok (folyt.)

- Ezzel racionális számokat megvalósító programunk első változata kész is van.
- Néhány példa a program használatára:

```
val oneHalf  = makeRat(1,2);
val oneThird = makeRat(1,3);
val twoThird = makeRat(2,3);

printRat oneHalf;
printRat(addRat(oneHalf, oneThird));
printRat(mulRat(oneHalf, oneThird));
printRat(addRat(oneThird, oneThird));

equRat(addRat(oneThird, oneThird), twoThird);

oneThird = oneThird;
addRat(oneThird, oneThird) = twoThird;
```


Adatabsztrakció: racionális számok (folyt.)

- Az utolsó példából, ha kipróbáljuk, láthatjuk, hogy programunk nem *normalizálja*, azaz nem a lehető legegyszerűbb alakban tárolja, ill. írja ki a racionális számokat.
- Segíthetünk a dolgon, ha a konstruktorműveletben a számlálót és a nevezőt a legnagyobb közös osztójukkal elosztjuk:

```
fun makeRat (n, d) =
      let val g = gcd(n, d) in (n div g, d div g) : rat end;
```

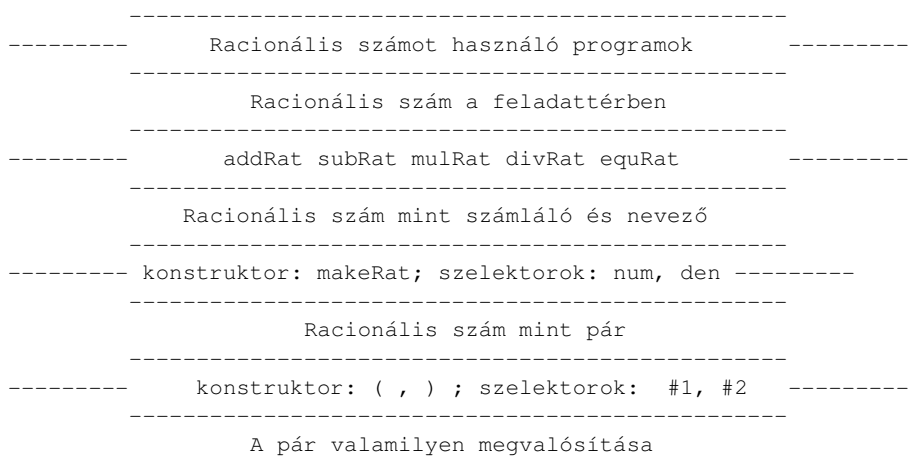
  A szelektorműveleteken nem változatunk.
- A racionális számokat normalizált alakjukban tároljuk, ezért nemcsak a kiírás, hanem az egyenlőségvizsgálat is helyes eredményt ad:

```
printRat(addRat(oneThird, oneThird));
addRat(oneThird, oneThird) = twoThird;
```

- A normalizáláshoz csak egyetlen helyen kellett változtatni a programon!

Adatabsztrakció: racionális számok (folyt.)

Adatabsztrakciós korlátok a racionális számok csomagban



Adatabsztrakció: racionális számok (folyt.)

- Az absztrakciós korlátok elszigetelik egymástól a program egyes részeit.
- Előnye, hogy a programokat egyszerűbb karbantartani és módosítani, pl. az adatok ábrázolását megváltoztatni.
- Pl. a racionális szám normalizálható a létrehozása helyett akkor, amikor a számlálójára vagy a nevezőjére van szükségünk. Ha gyakran hozunk létre racionális számokat, de csak ritkán használjuk a számlálóját vagy a nevezőjét, akkor az utóbbi megoldás a hatékonyabb.

```
fun num (q : rat) =
      let val (n, d) = q; val g = gcd(n, d) in n div g end
fun den (q : rat) =
      let val (n, d) = q; val g = gcd(n, d) in d div g end
```

- A `makeRat` függvény nem normalizáló változatát használjuk; a program többi része nem változik.

```
printRat(addRat(oneThird, oneThird));
addRat(oneThird, oneThird) = twoThird;
equRat(addRat(oneThird, oneThird), twoThird) = true;
```


Adatabsztrakció: racionális számok (folyt.)

- *Adatokról* szólva nem elég annyit mondanunk, hogy „adat az, amit az adott konstruktorok és szelektorok megvalósítanak”.
- Nyilvánvaló, hogy konstruktorok és szelektorok csak bizonyos halmaza alkalmas pl. a racionális számok megvalósítására.
- Racionális számok esetén a konstruktornak és a szelektoroknak garantálniuk kell az alábbi feltételek (axiómák) teljesülését:

```
(* PRE : d > 0 *
 x = makeRat(n, d);
 n = num x
 d = den x
```

- Eggyel alacsonyabb absztrakciós szinten a pár-ábrázolásnak is ki kell elégítenie a következő feltételeket:

```
q = (x, y)
x = #1 q
y = #2 q
```


Adatabsztrakció: racionális számok (folyt.)

- Bármely megvalósítás, amely ezeket a feltételeket kielégíti, megfelel, például a következő is:

```
exception Cons of string;
fun cons (x, y) =
      let fun dispatch 0 = x
            | dispatch 1 = y
            | dispatch _ = raise Cons "argument not 0 or 1"
      in  dispatch
      end;
fun fst z = z 0;
fun snd z = z 1;
```

- A tulajdonságleíró egyenletek

```
q = cons(n, d)
n = fst q
d = snd q
```

- Vegyük észre, hogy a racionális számot megvalósító `cons` objektum: *függvény!* `fst` és `snd` *üzenetet küld* az objektumnak. Ennek a programozási stílusnak ezért *üzenetküldés* a neve.

Adatabsztrakció: racionális számok (folyt.)

- Példa:

```
val q = cons(1, 2);
fst q = 1; snd q = 2;
```

- A konstruktor és a szelektorok megvalósítása üzenetküldéssel:

```
fun makeRat (n, d) =
      let val g = gcd(n, d) in cons(n div g, d div g) end;
fun num q = fst q;
fun den q = snd q;
```

- Racionális számokat megvalósító csomagunk nagy hibája, hogy *gyenge absztrakciót* valósít meg, azaz nem rejti el a megvalósítás részleteit; a programozóra bízza, hogy az absztrakciós korlátokat milyen mértékben tartja be. Ez hibák forrása.
- A megvalósítás részleteit *erős absztrakcióval,* modulok segítségével rejthetjük el a külvilág elől. Az „implementációs” modul neve az SML-ben: `structure`, az (opcionális) „interfészmodul” neve pedig: `signature`.

```
structure name = struct ... end
signature name = sig ... end
```


Adatabsztrakció modulokkal: racionális számok

```
(* compile "Gcd.sml" *)
load "Gcd";

structure Rat =
struct
  type rat = int * int;
  fun makeRat (n, d) = let val g = Gcd.gcd(n, d) in (n div g, d div g) : rat end
  fun num (q : rat) = #1 q
  fun den q = #2 (q : rat)
  fun addRat(x, y) = makeRat(num x * den y + num y * den x, den x * den y)
  fun subRat(x, y) = makeRat(num x * den y - num y * den x, den x * den y)
  fun mulRat(x, y) = makeRat(num x * num y, den x * den y)
  fun divRat(x, y) = makeRat(num x * den y, den x * num y)
  fun equRat(x, y) = num x * den y = den x * num y
  fun printRat q = print(makestring(num q) ^ "/" ^ makestring(den q) ^ "\n");
  val one      = makeRat(1,1)
  val zero     = makeRat(0,1)
  val oneHalf  = makeRat(1,2)
  val oneThird = makeRat(1,3)
  val twoThird = makeRat(2,3)
end;
```

Az absztrakció még nem elég erős: a részletek nincsenek eléggé elrejtve!

Adatabsztrakció modulokkal: racionális számok (folyt.)

Ez a megvalósított `Rat` struktúra tényleges szignatúrája:

```
> structure Rat :
  {type rat = int * int,
   val addRat : (int * int) * (int * int) -> int * int,
   val den : int * int -> int,
   val divRat : (int * int) * (int * int) -> int * int,
   val equRat : (int * int) * (int * int) -> bool,
   val makeRat : int * int -> int * int,
   val mulRat : (int * int) * (int * int) -> int * int,
   val num : int * int -> int,
   val one : int * int,
   val oneHalf : int * int,
   val oneThird : int * int,
   val printRat : int * int -> unit,
   val subRat : (int * int) * (int * int) -> int * int,
   val twoThird : int * int,
   val zero : int * int}
```

Kilátszik a `rat` típus két komponensének `int` típusa.

Adatabsztrakció modulokkal: racionális számok (folyt.)

A szignatúra létrehozása és a struktúrához kötése *korlátozza* a megvalósított értékek láthatóságát:

```
signature Rat = sig
                  type rat
                  val makeRat : int * int -> rat
                  val num : rat -> int
                  val den : rat -> int
                  val addRat : rat * rat -> rat
                  val subRat : rat * rat -> rat
                  val mulRat : rat * rat -> rat
                  val divRat : rat * rat -> rat
                  val equRat : rat * rat -> bool
                  val printRat : rat -> unit
                  val one : rat
                  val oneHalf : rat
                  val oneThird : rat
                  val twoThird : rat
                  val zero : rat
                end;
```

Adatabsztrakció modulokkal: racionális számok (folyt.)

```
structure Rat :> Rat = (* ez ún. áttetsző szignatúrakötés *)
struct
  type rat = int * int;
  fun makeRat (n, d) = let val g = Gcd.gcd(n, d)
                       in
                           (n div g, d div g) : rat
                       end
  fun num (q : rat) = #1 q
  fun den q = #2 (q : rat)

  fun addRat(x, y) = makeRat(num x * den y + num y * den x, den x * den y)
  fun subRat(x, y) = makeRat(num x * den y - num y * den x, den x * den y)
  fun mulRat(x, y) = makeRat(num x * num y, den x * den y)
  fun divRat(x, y) = makeRat(num x * den y, den x * num y)
  fun equRat(x, y) = num x * den y = den x * num y
  fun printRat q = print(makestring(num q) ^ "/" ^ makestring(den q) ^ "\n");

  val one      = makeRat(1,1)
  val zero     = makeRat(0,1)
  val oneHalf  = makeRat(1,2)
  val oneThird = makeRat(1,3)
  val twoThird = makeRat(2,3)
end;
```


Adatabsztrakció modulokkal: racionális számok (folyt.)

Ez a `Rat` szignatúrához *(áttetsző szignatúrakötéssel)* kötött `Rat` struktúra tényleges szignatúrája:

```
> New type names: rat
  structure Rat :
  {type rat = rat,
   val addRat : rat * rat -> rat,
   val den : rat -> int,
   val divRat : rat * rat -> rat,
   val equRat : rat * rat -> bool,
   val makeRat : int * int -> rat,
   val mulRat : rat * rat -> rat,
   val num : rat -> int,
   val one : rat,
   val oneHalf : rat,
   val oneThird : rat,
   val printRat : rat -> unit,
   val subRat : rat * rat -> rat,
   val twoThird : rat,
   val zero : rat}
```


Adatabsztrakció modulokkal: racionális számok (folyt.)

- Példák a `Rat` struktúra használatára:

```
open Rat;
printRat oneHalf;
printRat(addRat(oneHalf, oneThird));
printRat(mulRat(oneHalf, oneThird));
printRat(addRat(oneThird, oneThird));

equRat(addRat(oneThird, oneThird), twoThird);
addRat(oneThird, oneThird) = twoThird;

! addRat(oneThird, oneThird) = twoThird;
! ^^^^^^^^^^^^^^^^^^^^^^^^^
! Type clash: expression of type
!   rat
! cannot have equality type ''a
```

- Hopp! Az = reláció nem használható!
- Ha akarjuk, az `eqtype` deklarációval meg kell mondanunk az mosml-értelmezőnek, hogy `rat` típusú értékek egyenlőségvizsgálatát engedélyezzük, azaz a `rat` ún. *egyenlőségi típus*.

Adatabsztrakció modulokkal: racionális számok (folyt.)

```
signature Rat =
sig
   eqtype rat
   val makeRat : int * int -> rat
   val num : rat -> int
   val den : rat -> int
   ...
   val zero : rat
end;
```

```
> signature Rat =
  /\=rat.
    {type rat = rat,
     val makeRat : int * int -> rat,
     val num : rat -> int,
     val den : rat -> int,
     ...
     val zero : rat}
```