

ABSZTRAKCIÓ FÜGGVÉNYEKEL (ELJÁRÁSOKKAL)

Absztrakció függvényekkel (eljárásokkal) FP-8.10-3

Prímteszt

- A prime predikátum egy n szám prím voltát teszteli. A findDivisor függvény 2-től kezdve megkeresi az n szám legkisebb osztóját. Az n szám prím, ha a legkisebb osztó az n szám maga.
- Az n osztóit 2-től \sqrt{n} -ig kell keresni, így a lépések száma $O(\sqrt{n})$.

```

fun prime n =
  let
    infix divides
    fun smallestDivisor n = findDivisor(n, 2)
    and findDivisor (n, testDivisor) =
      if square testDivisor > n
      then n
      else if testDivisor divides n
      then testDivisor
      else findDivisor(n, testDivisor+1)
    and square x = x * x
    and a divides b = b mod a = 0
  in
    n = smallestDivisor n
  end

```

Gyakorló feladat.

prime egyesével lépésként keresi meg az n legkisebb osztóját. Írjon gyorsabb megoldást!

Legnagyobb közös osztó

- Következő példának a és b legnagyobb közös osztóját számolja ki az euklideszi algoritmussal.
- Az alapgondolat az, hogy ha a -t b -vel osztva r a maradék, akkor a és b közös osztói azonosak b és r közös osztóival.
- A matematikai definíciót most is pontosan követi az SML-függvény.

$$\begin{array}{l} \text{gcd}(a, 0) = a \\ \text{gcd}(a, b) = \text{gcd}(b, a \bmod b) \end{array} \quad \left| \begin{array}{l} \text{fun gcd (a, 0) = a} \\ \text{gcd (a, b) = gcd (b, a mod b)} \end{array} \right.$$

- A *folyamat* iteratív. A lépések száma logaritmikusan nő.
Pontosabban – a *Lamé-tétel* szerint – ha az euklideszi algoritmus egy számpár legnagyobb közös osztóját k lépésben számítja ki, akkor a számpár kisebbik tagja nem lehet kisebb a k -adik Fibonacci-számnál. (Ld. SICP, 1.2.5. szakasz.)
Legyen n az algoritmus kisebbik paramétere. Ha a legnagyobb közös osztó kiszámításához k lépésre van szükség, akkor $n \geq F(k) \approx \Phi^k / \sqrt{5}$. Azaz a k lépésszám valóban az n (Φ alapú) logaritmusával arányos.

Deklaratív programozás. BME VIK, 2004. őszi félév

(Funkcionális programozás)

Absztrakció függvényekkel (eljárásokkal) FP-8.10-4

Prímteszt (folyt.)

- A következő SML-predikátum egy szám prím voltát *valószínűségi módszerrel* teszteli. A lépések száma $O(\lg n)$.
- Az algoritmus a kis Fermat-tételre alapul, amely azt mondja ki, hogy:
ha n prím és $0 < a < n$, akkor a^n modulo n szerint *kongruens* a -val, azaz $a^n \bmod n = a$.
 - Két szám akkor *kongruens* modulo n szerint, ha n -nel osztva mindkettőnek ugyanaz a maradéka. Egy a szám n -nel való osztásának maradékát a modulo n szerinti maradékának, vagy röviden a modulo n -nek is nevezik.
- Ha n nem prím, akkor az $a < n$ számok nagy hányadára nem teljesül a fenti reláció.
- A prímteszt algoritmusai ezek után a következők:
 - Adott n -re véletlenszerűen válasszunk egy $a < n$ számot: ha $a^n \bmod n \neq a$, akkor n nem prím. Ellenkező esetben nagy a valószínűsége, hogy n prím.
 - Válasszunk véletlenszerűen egy másik $a < n$ számot: ha $a^n \bmod n = a$, akkor növekedett annak a valószínűsége, hogy az n prím. Újabb és újabb a értékeket választva egyre biztosabbak lehetünk abban, hogy az n prím.

Prímteszt (folyt.)

- Az `expmod` segédfüggvény a `base` szám `exp`-edik hatványának modulo `m` szerinti maradékát adja eredményül.

```
(* expmod (base, exp, m) = base exp-edik hatványa modulo m
*)
fun expmod (_, 0, _) = 1
  | expmod (b, e, m) =
    if even e
    then square(expmod(b, e div 2, m)) mod m
    else b * expmod(b, e-1, m) mod m
and even n = n mod 2 = 0
and square x = x * x
```

- Nagyon hasonló felépítésű `exptFast`-hoz. A lépések száma a kitevő logaritmusával arányos.
- Szükségünk van véletlenszámok előállítására. Részletek az SML alapkönyvtárból:

```
Random.range (min, max) gen = an integral random number in the
  range [min, max). Raises Fail if min > max.
Random.newgen () = a random number generator, taking the seed
  from the system clock.
```

Függvények mint általános számítási módszerek

- 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 függetlenné 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.

Prímteszt (folyt.)

- Betöltjük a Random könyvtárat:

```
loadOne "Random";
```

- `fermatTest` generál egy álvéletlen-számot, és egyszer elvégzi a vizsgálatot:

```
(* fermatTest n = false if n is not prime
*)
fun fermatTest n =
  let fun tryIt a = expmod(a, n, n) = a
      in tryIt(Random.range (1, n) (Random.newgen()))
  end
```

- `fastPrime times`-szor megismétli a vizsgálatot:

```
(* fastPrime (n, times) = true if n passes the prime test
  times times
*)
fun fastPrime (n, 0) = true
  | fastPrime (n, t) = fermatTest n andalso fastPrime(n, t-1)
```

- Ez a megoldás csak nagy valószínűséggel, de nem teljes bizonyossággal ad választ a kérdésre.

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

```

Deklaratív programozás. BME VIK, 2004. őszi félév

(Funkcionális programozás)

Absztrakció függvényekkel (eljárásokkal) FP-8.10-11

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

```

- A `makestring` függvény (típusa `numtxt -> string`) tetszőleges numerikus (`int`, `real`, `word`, `word8`), `char` és `string` típusú értéket `string` típusúvá alakít.
- 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.

Deklaratív programozás. BME VIK, 2004. őszi félév

(Funkcionális programozás)

Egyenlet gyökeinek meghatározása intervallumfelezéssel (folyt.)

- Az előfeltételek betartását célszerű `search` alkalmazásakor 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.

Deklaratív programozás. BME VIK, 2004. őszi félév

(Funkcionális programozás)

Absztrakció függvényekkel (eljárásokkal) FP-8.10-12

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

```

Deklaratív programozás. BME VIK, 2004. őszi félév

(Funkcionális programozás)

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);
```

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 *átlagsillapítás* sokszor használható módszer, ezért érdemes önálló függvényként megírni: ha adott az f függvény, elő kell állítani x és $f x$ 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 édesítőszerral).

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

- `sqrt averageDamp`-pel felírt változata explicitté teszi a *fixpontmeghatározás* és az *átlagszillapítás* 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 *lényeg* 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.): 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.

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 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` fixpontkeresésén 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)
```

## ABSZTRAKCIÓ ADATOKKAL

### 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:

$$n_1/d_1 + n_2/d_2 = (n_1d_2 + n_2d_1)/(d_1d_2), \quad n_1/d_1 - n_2/d_2 = (n_1d_2 - n_2d_1)/(d_1d_2),$$

$$(n_1/d_1)(n_2/d_2) = (n_1n_2)/(d_1d_2), \quad (n_1/d_1)/(n_2/d_2) = (n_1d_2)/(d_1n_2),$$

$$n_1/d_1 = n_2/d_2 \text{ akkor és csak akkor, ha } n_1d_2 = n_2d_1.$$

```

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

- 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 tétélezzenek 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 *konstruktorokból* 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.)

- 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 címkeje*, 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.)

### Adatabsztrakciós korlátok a racionális számok csomagban

```

Racionális számot használó programok

Racionális szám a feladattérben

addRat subRat mulRat divRat equRat

Racionális szám mint számláló és nevező

konstruktor: makeRat; szelektorok: num, den

Racionális szám mint pár

konstruktor: (,) ; szelektorok: #1, #2

A pár megvalósítása SML-ben
```

## 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 változtatunk.

- 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.)

- 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.)

- 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 bízta, 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: 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 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.)

```
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.)

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.)

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.)

- A Rat struktúrában definiált értékekre teljes nevükkel kell hivatkozni:

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

- open-nel – a szignatúra által korlátozott mértékben – láthatóvá tehetjük a struktúra tartalmát:

```
open Rat;
equRat (addRat (oneThird, oneThird), twoThird);
addRat (oneThird, oneThird) = twoThird;
```

- A láthatóvá tétel lehet lokális (deklaráció, ill. kifejezés lokális deklarációval):

```
local open Rat
 val q1 = addRat (oneThird, oneThird); val q2 = twoThird
in val ratPair = (q1, q2)
end;

let open Rat
in printRat (addRat (oneThird, oneThird));
end;
```

## 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}
```

## Adatabsztrakció modulokkal: racionális számok (folyt.)

- Válasszunk a matematikában megszokotthoz közelebb álló neveket a függvényeknek:

```
signature Rat = sig
 eqtype rat
 val rat : int * int -> rat
 val num : rat -> int
 val den : rat -> int
 val ++ : rat * rat -> rat
 val -- : rat * rat -> rat
 val ** : rat * rat -> rat
 val // : rat * rat -> rat
 val == : rat * rat -> bool
 val toString : rat -> string
 val one : rat
 val oneHalf : rat
 val oneThird : rat
 val twoThird : rat
 val zero : rat
end;
```

## Adatabsztrakció modulokkal: racionális számok (folyt.)

```

structure Rat :> Rat =
struct
 type rat = int * int;
 fun rat (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 op++(x, y) = rat(num x * den y + num y * den x, den x * den y)
 fun op--(x, y) = rat(num x * den y - num y * den x, den x * den y)
 fun op**(x, y) = rat(num x * num y, den x * den y)
 fun op//(x, y) = rat(num x * den y, den x * num y)
 fun op==(x, y) = num x * den y = den x * num y
 fun toString r = makestring(num r) ^ "/" ^ makestring(den r)
 val one = rat(1,1)
 val zero = rat(0,1)
 val oneHalf = rat(1,2)
 val oneThird = rat(1,3)
 val twoThird = rat(2,3) end;

```

## Adatabsztrakció modulokkal: racionális számok (folyt.)

- A szokásos alpműveleti jeleket is újradefiniálhatjuk.
- Eredeti jelentésük nem vész el, de a műveletek teljes nevét kell használnunk *prefix* pozícióban:

```

load "Int";
let open Rat
 val op+ = ++
 val op- = --
 val op* = **
 val op/ = //
in
 print(toString oneHalf ^ "\n");
 print(toString(oneHalf + oneThird) ^ "\n");
 print(toString(oneHalf * oneThird) ^ "\n");
 print(toString(oneThird - oneThird) ^ "\n");
 print(toString(twoThird / oneThird) ^ "\n");
 oneThird + oneThird = twoThird;
 Int.(1,2);
 1 Int.+ 2 (* hibás! *)
end;

```

## Adatabsztrakció modulokkal: racionális számok (folyt.)

- Az új műveleti jelek prefix pozícióban használhatók:

```

let open Rat
in
 print(toString(++(**(oneThird, oneHalf),oneThird) ^ "\n");
 ++(oneThird, oneThird) = twoThird
end;

```

A ( és a \*\* közé legalább egy szóköz kell, különben az mosml *megjegyzés* kezdetének veszi!

- Vagy akár infix pozíciójává alakíthatók:

```

let open Rat
 infix 6 ++ --
 infix 7 ** //
in
 print(toString(oneThird ** oneHalf ++ oneThird) ^ "\n");
 oneThird ++ oneThird = twoThird
end;

```

## Adatabsztrakció modulokkal: racionális számok (folyt.)

- Új típust és konstruktorokat hozhatunk létre a datatype deklarációval:

```

structure Rat :> Rat =
struct
 datatype rat = Rat of int * int
 fun rat (n, d) = let val g = Gcd.gcd(n, d) in Rat(n div g, d div g)
 fun num (Rat q) = #1 q
 fun den (Rat q) = #2 q
 fun op++(x, y) = rat(num x * den y + num y * den x, den x * den y)
 fun op--(x, y) = rat(num x * den y - num y * den x, den x * den y)
 fun op**(x, y) = rat(num x * num y, den x * den y)
 fun op//(x, y) = rat(num x * den y, den x * num y)
 fun op==(x, y) = num x * den y = den x * num y
 fun toString r = makestring(num r) ^ "/" ^ makestring(den r);
 val one = rat(1,1)
 val zero = rat(0,1)
 val oneHalf = rat(1,2)
 val oneThird = rat(1,3)
 val twoThird = rat(2,3) end;

```

## Adatabsztrakció modulokkal: racionális számok (folyt.)

- Az adatkonstruktor mintaillesztésre *szelektorként* is felhasználható (és használni is kell):

```
structure Rat :> Rat =
struct
 datatype rat = Rat of int * int;
 fun rat (n, d) = let val g = Gcd.gcd(n, d) in Rat(n div g, d div g)
 fun num (Rat(n, _)) = n
 fun den (Rat(_, d)) = d
 fun op++(x, y) = rat(num x * den y + num y * den x, den x * den y)
 fun op--(x, y) = rat(num x * den y - num y * den x, den x * den y)
 fun op**(x, y) = rat(num x * num y, den x * den y)
 fun op//(x, y) = rat(num x * den y, den x * num y)
 fun op==(x, y) = num x * den y = den x * num y
 fun toString r = makestring(num r) ^ "/" ^ makestring(den r);
 val one = rat(1,1)
 val zero = rat(0,1)
 val oneHalf = rat(1,2)
 val oneThird = rat(1,3)
 val twoThird = rat(2,3) end;
```

## Adatabsztrakció modulokkal: racionális számok (folyt.)

- Az adatkonstruktorfüggvény *valóban* használható új érték létrehozására:

```
structure Rat :> Rat =
struct
 datatype rat = Rat of int * int;
 val rat = Rat;
 fun num (Rat(n, _)) = n
 fun den (Rat(_, d)) = d
 fun op++(x, y) = rat(num x * den y + num y * den x, den x * den y)
 fun op--(x, y) = rat(num x * den y - num y * den x, den x * den y)
 fun op**(x, y) = rat(num x * num y, den x * den y)
 fun op//(x, y) = rat(num x * den y, den x * num y)
 fun op==(x, y) = num x * den y = den x * num y
 fun toString r = makestring(num r) ^ "/" ^ makestring(den r);
 val one = rat(1,1)
 val zero = rat(0,1)
 val oneHalf = rat(1,2)
 val oneThird = rat(1,3)
 val twoThird = rat(2,3) end;
```

## Összefoglalás: gyenge és erős adatabsztrakció

- Gyenge absztrakció: a név szinonima, az adatszerkezet részei továbbra is hozzáférhetők.
- Erős absztrakció: a név új dolgot (entitást, objektumot) 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.
  - Túlhaladott, van helyette jobb: datatype + modulok
- datatype: modulok nélkül gyenge, modulokkal erős absztrakció;
  - pl. datatype 'a esetleg = Semmi | Valami of 'a
  - Belső változata az SML-ben: datatype 'a option = NONE | SOME of 'a
  - Új entitást hoz létre.
  - Rekurzív és polimorf is lehet.

## GYENGE ÉS ERŐS ABSZTRAKCIÓN

## Összefoglalás: gyenge és erős adatabsztrakció (folyt.)

- `datatype logi = Igen | Nem` Felsorolásos típus.  
`datatype logi3 = igen | nem | talan` Felsorolásos típus.  
`datatype 'a esetleg = Semmi | Valami of 'a` Polimorf típus.
- **Adatkonstruktor**nak nevezzük a létrehozott Igen, Nem, igen, nem, talan, Semmi és Valami értékeket. Valami ún. *adatkonstruktorfüggvény*, az összes többi ún. *adatkonstruktorállandó*. Az adatkonstruktorok a többi értékével azonos névtérben vannak.
- **Típuskonstruktor**nak nevezzük a létrehozott `logi`, `logi3` és `esetleg` neveket; `esetleg` ún. postfix *típuskonstruktorfüggvény* (vagy típusoperátor), a másik kettő ún. *típuskonstruktorállandó* (röviden típusállandó). Típusnévként használható a típusállandó (pl. `logi`), valamint a típusállandóra vagy típusváltozóra alkalmazott típuskonstruktorfüggvény (pl. `int list` vagy `'a esetleg`). A típuskonstruktorok más névtérben vannak, mint az értékek.
- Természetesen az adatkonstruktoroknak is van típusuk, pl.

|      |        |        |                    |
|------|--------|--------|--------------------|
| Igen | : logi | Semmi  | : 'a esetleg       |
| Nem  | : logi | Valami | : 'a -> 'a esetleg |
- Példa `datatype` deklarációval létrehozott adattípust kezelő függvényre

```
fun inverz Nem = Igen | Igen = Nem
```

## 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* (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
```

## 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
```

## 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.)

- 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
```

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 felsorolása 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 egyenletek* tekintjük:

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

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
```

Felsorolós 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 deklarálhatnánk, ill. definiálhatnánk:

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

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.

Polimorf adattípusok

- Láttuk, hogy a list postfix pozíciójú *típusoperátor*, nem típus: a datatype deklaráció az adatkonstruktorok mellett *típuskonstruktor* 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
```

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 lekötöni.

Esetsztérválasztás (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 Pi mintához tartozó Ei kifejezés lesz.

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

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

Például a lady függvényt így is definiálhattuk volna:

```
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"

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

ESETSZÉTVÁLASZTÁS, OPCIONÁLIS ÉRTÉK

Opcionális érték kezelése ('a option)

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

Függvények az Option könyvtárból:

```
val getOpt      : '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)
```

getOpt (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"
```

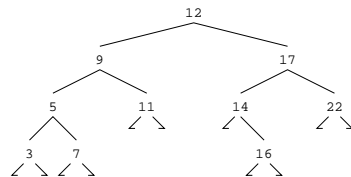
BINÁRIS FÁK

Bináris fák datatype deklarációval

- A listához hasonlóan rekurzív adatszerkezetet 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:

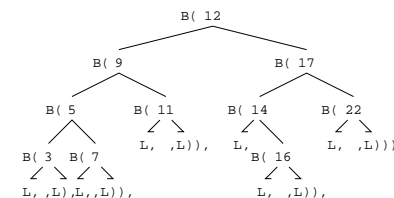


- Az 'a tree' adattípus L és B adatkonstruktoraival ez a fa pl. a következő lapon látható módon írható le.

Bináris fák datatype deklarációval (folyt.)

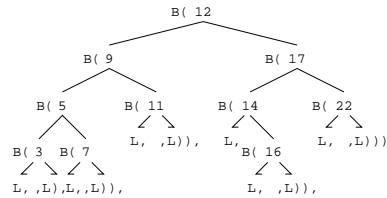
```
B(B(B(B(L, 3, L),
5,
B(L, 7, L)
),
9,
B(L, 11, L)
),
12,
B(B(L,
14,
B(L, 16, L)
),
17,
B(L, 22, L)
)
)
```

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



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)
  
```

Egyszerű műveletek bináris fákön

- 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
  
```

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 ágoknak (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 fákön (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
  
```