SICP

Forrás

Hanák Péter SML-átiratának visszaírásával

Irodalom: [SICP] Abelson, Sussman & Sussman: *Structure and Interpretation of Computer Programs,* The MIT Press, 1996

ABSZTRAKCIÓ FÜGGVÉNYEKKEL (ELJÁRÁSOKKAL)

Négyzetgyökvonás Newton-módszerrel

- A funkcionális nyelvi függvények sokban hasonlítanak a matematikai függvényekhez: egy vagy több argumentumtól függő értéket adnak eredményül. Egy dologban azonban mindenképpen különböznek: a funkcionális nyelvi függvényeknek *hatékonyaknak* is kell lenniük.
- Nézzük pl. a négyzetgyök következő definícióját: $\sqrt{x} = y$, ahol $y \geq 0$ és $y^2 = x$.
- Ez az egyenletrendszer alkalmas pl. annak ellenőrzésére, hogy egy szám egy másiknak a négyzetgyöke-e, de nem alkalmas a négyzetgyök előállítására.
- A matematikai függvénnyel egy bizonyos tulajdonságot *deklarálunk,* a funkcionális nyelvi függvénnyel (eljárással) azt is megmondjuk, *hogyan kell kiszámítani* az adott értéket. (A deklaratív programozás tehát csak az imperatív programozáshoz *képest* tekinthető deklaratívnak. Vö. MIT és HOGYAN.)
- A négyzetgyökszámítás legismertebb módszere a *szukcesszív approximáció:* ha az y az x négyzetgyökének egy közelítése, akkor az y és az x/y átlaga a négyzetgyök egy jobb közelítése lesz. A lépéssorozat akkor fejeződik be, amikor a közelítő értéket már elég jónak tartjuk.
- Írjuk le ezt az algoritmust Scheme-ben (l. következő dia)!

Négyzetgyökvonás Newton-módszerrel (folyt.)

```
(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x)
                 x)))
```

- A megoldási stratégiánkat jól tükrözi a fenti programrészlet. Ezt a stílust *fölülről lefelé haladó* (top down) módszernek nevezik. Kezdetben nem foglalkozunk a részletekkel, feltesszük, hogy minden megvan, amire szükségünk van, legfeljebb később megírjuk.
- Ezután definiálnunk kell azokat az eljárásokat, amiket felhasználunk (lásd forrás).

Végül meg kell hívnunk az `sqrt-iter` függvényt a négyzetgyök első közelítő értékével.

```
(define (sqrt_newton x) (sqrt-iter 1.0 x))
```

Eljárások (függvények) és folyamatok

- Az eljárások (függvények) olyan *minták*, amelyek megszabják a számítási folyamatok (processzek) menetét, *lokális* viselkedését.
- Egy számítási folyamat *globális* viselkedését (pl. a szükséges lépések számát, a végrehajtási időt) általában nehéz megbecsülni, de törekednünk kell rá.

Lineáris rekurzió és iteráció

- A faktoriális matematikai definíciójának hű tükörképe az alábbi Scheme-program:

$0! = 1$

$n! = n(n-1)!$

```
(define (factorial n)
  (if (= n 1)
      1 (* n (factorial (- n 1)))))
```

- Ha a helyettesítési modellünket alkalmazzuk, láthatjuk, hogy a program által létrehozott folyamat az összes tényezőt `n`-től 1-ig eltárolja, mielőtt az első szorzást végrehajtaná („késlelteti” a szorzásokat) – a folyamat *lineáris-rekurzív folyamat.*
- Ha ehelyett 1-et szoroznánk 2-vel, majd a részszorzatokat 3-mal, 4-gyel s.í.t., akkor az `n` érték meghaladásakor az utolsó részszorzat éppen `n` faktoriálisa lenne! Ehhez a programban szükségünk van egy olyan *formális paraméterre* (tkp. lokális változóra), amely tárolja a részszorzat aktuális értékét, és egy másikra, amely 1-től `n`-ig számlál. A létrehozott folyamat *lineáris-iteratív folyamat.*

Lineáris rekurzió és iteráció (folyt.)

```
(define (faci n) (fact-iter 1 1 n))

(define (fact-iter product counter max-count)
  (if (> counter max-count)
      product
      (fact-iter (* counter product) (+ counter 1) max-count)))
```

Fact-iter jobban érthető változatát kapjuk, ha a számlálót lefele számláltatjuk.

```
(define (faci n) (fact-iter 1 n))

(define (fact-iter product counter)
  (if (< counter 1)
      product
      (fact-iter (* counter product) (- counter 1))))
```

Eljárások (függvények) és folyamatok

- Ne tévesszük össze egymással a rekurzív (számítási) folyamatot és a rekurzív függvényt (eljárást)!
- Egy rekurzív függvény esetén csupán a szintaxisról van szó, arról, hogy hivatkozik-e a függvény (eljárás) önmagára.
- A folyamat esetében viszont a folyamat menetéről, lefolyásáról beszélünk.
- Ha egy függvény *jobbrekurzív (tail-recursive),* a megfelelő folyamat – az értelmező/fordító jóságától függően – még lehet iteratív.

Még visszatérünk az „absztrakció függvényekkel” témakörhöz, de most témát váltunk: megismerkedünk a *paraméteres polimorfizmus* fogalmával, majd egy nagyon funkcionális adatszerkezettel, a listával foglalkozunk.

Elágazó rekurzió

- Korábban lineáris-rekurzív, ill. lineáris-iteratív folyamatokra láttunk példákat (faktoriális kiszámítása kétféleképpen).
- Most *elágazó rekurzióra* nézzünk példát: állítsuk elő a Fibonacci-számok sorozatát.
- Egy Fibonacci-számot az előző két Fibonacci-szám összege adja:

```
0, 1, 1, 2, 3, 5, 8, 13, 21, ...
```

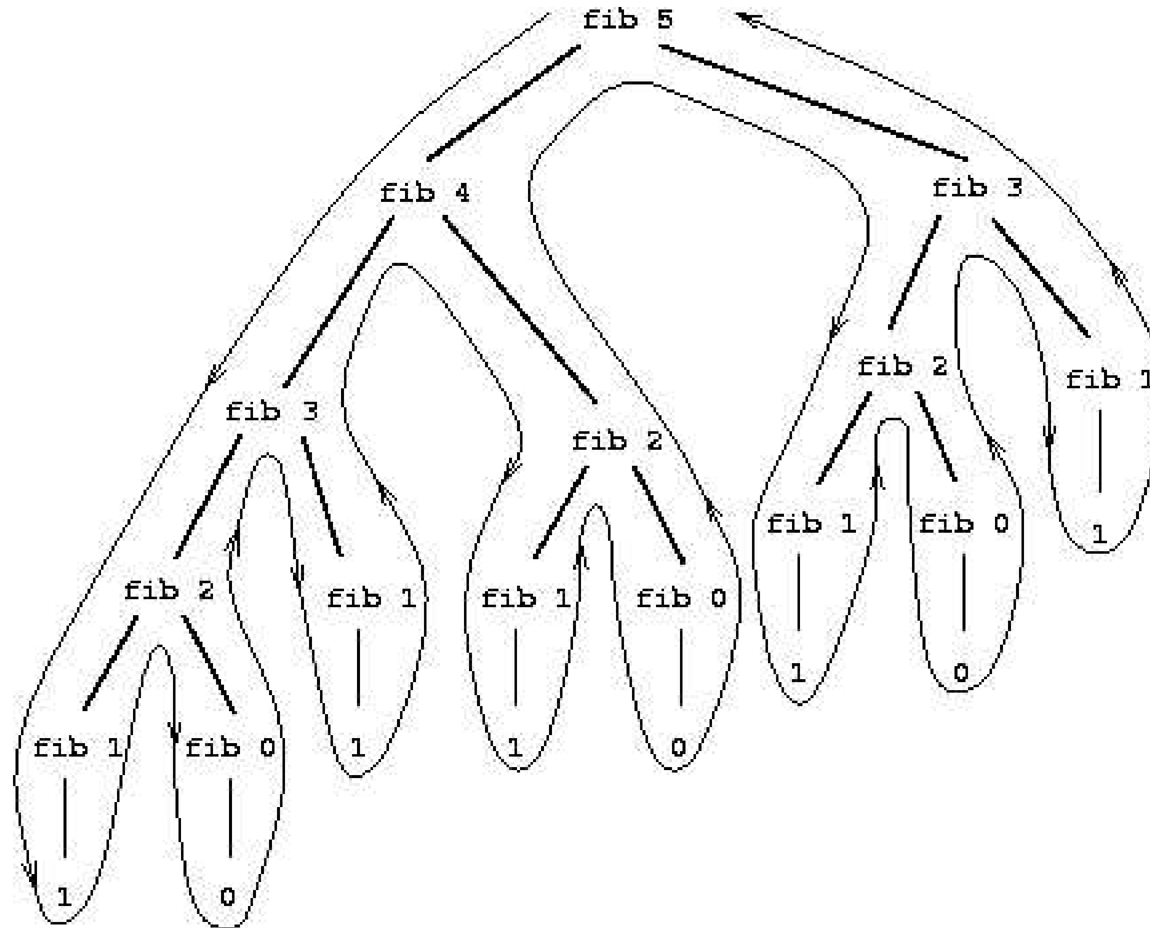
- A Fibonacci-számok matematikai definíciója könnyen átírható Scheme-eljárássá:

$$F(0) = 0$$
$$F(1) = 1$$
$$F(n) = F(n-1) + F(n-2), \text{ ha } n > 1$$

```
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
   (else (+ (fib (- n 1))
            (fib (- n 2))))))
```

- A következő lapon látható ábra illusztrálja az elágazóan rekurzív folyamatot `fib 5` kiszámítása esetén.

Elágazó rekurzió (folyt.)



- `fib 5`-öt `fib 4` és `fib 3`, `fib 4`-et `fib 3` és `fib 2` kiszámításával stb. kapjuk.

Elágazó rekurzió (folyt.)

- Az előző program alkalmas az elágazó rekurzió lényegének bemutatására, de szinte alkalmatlan a Fibonacci-számok előállítására!
- Vegyük észre, hogy pl. `fib 3`-at kétszer is kiszámítjuk, azaz a munkának ezt a részét kb. a harmadát) feleslegesen végezzük el.
- Belátható, hogy $F(n)$ meghatározásához pontosan $F(n+1)$ levélből álló fát kell bejárni, azaz ennyiszer kell meghatározni $F(0)$-t vagy $F(1)$-et.
- $F(n)$ exponenciálisan nő n-nel.

  Pontosabban, $F(n)$ a $\Phi^n/\sqrt{5}$-höz közel eső egész, ahol $\Phi = (1-\sqrt{5})/2 \approx 1.61803$, az ún. *aranymetszés* arányszáma. Φ kielégíti a $\Phi^2 = \Phi + 1$ egyenletet.
- A megteendő lépések száma tehát $F(n)$-nel együtt exponenciálisan nő n-nel. Ugyanakkor a tárigény csak lineárisan nő n-nel, mert csak azt kell nyilvántartani, hogy hányadik szinten járunk a fában.
- Általában is igaz, hogy elágazó rekurzió esetén a lépések száma a fa csomópontjainak a számával, a tárigény viszont a fa maximális mélységével arányos.

Elágazó rekurzió (folyt.)

- A Fibonacci-számok lineáris-iteratív folyamattal is előállíthatók.

  Ha az a és b változók kezdőértéke rendre $F(1) = 1$ és $F(0) = 0$, és ismétlődően alkalmazzuk az $a \leftarrow a + b$, $b \leftarrow a$ transzformációkat, akkor n lépés után $a = F(n+1)$ és $b = F(n)$ lesz. Az iteratív folyamatot létrehozó Scheme-eljárás egy változata:

```
(define (fibi n) (fib-iter 1 0 n))

(define (fib-iter a b count)
  (if (= count 0)
      b
      (fib-iter (+ a b) a (- count 1))))
```

Elágazó rekurzió (folyt.)

- A Fibonacci-példában a lépések száma elágazó rekurziónál tehát n-nel exponenciálisan, lineáris rekurziónál n-nel arányosan nőtt, kis n-ekre is hatalmas a nyereség!
- Téves lenne azonban azt a következtetést levonni, hogy az elágazó rekurzió használhatatlan. Amikor hierarchikusan strukturált adatokon kell műveleteket végezni, pl. egy fát kell bejárni, akkor az elágazó rekurzió (angolul: *tree recursion)* nagyon is természetes és hasznos eszköz.
- Az elágazó rekurzió numerikus számításoknál az algoritmus első megfogalmazásakor is hasznos lehet: gondoljunk csak arra, hogy milyen könnyű volt átírni a Fibonacci-számok matematikai definícióját programmá.
- Ha már értjük a feladatot, az első, rossz hatékonyságú változatot könnyebb átírni jó, hatékony programmá. Az elágazó rekurzió segíthet a feladat megértésében.

Az iteratív Fibonacci-algoritmushoz csak egy aprócska ötlet kellett. A következő feladatra azonban nem lenne könnyű iteratív algoritmust írni.

- Hányféleképpen lehet felváltani *egy* dollárt 50, 25, 10, 5 és 1 centesekre?
- Általánosabban: adott összeget adott érmékkel hányféleképpen lehet felváltani?

Elágazó rekurzió (folyt.): pénzváltás

Tegyük föl, hogy n darab érme áll a rendelkezésünkre valamilyen (pl. nagyság szerint csökkenő) sorrendben. Ekkor az a összeg lehetséges felváltásainak számát úgy kapjuk meg, hogy

- kiszámoljuk, hogy az a összeg hányféleképpen váltható fel az első (d értékű) érmét kivéve a többi érmével, és ehhez
- hozzáadjuk, hogy az $a - d$ összeg hányféleképpen váltható fel az összes érmével, az elsőt is beleértve – más szóval azt, hogy az a összeget hányféleképpen tudjuk úgy felváltani, hogy a d érmét legalább egyszer felhasználjuk.

A feladat tehát rekurzióval megoldható, hiszen redukálható úgy, hogy kisebb összegeket kevesebb érmével kell felváltanunk. A következő alapeseteket különböztessük meg:

- Ha $a = 0$, a felváltások száma 1.

  (Ha az összeg 0, csak egyféleképpen, 0 db érmével lehet „felváltani”.)
- Ha $a < 0$, a felváltások száma 0.
- Ha $n = 0$, a felváltások száma 0.

A példában a `first-denomination` (magyarul *első címlet)* eljárást felsorolással valósítottuk meg. Tömörebb és rugalmasabb lenne a megvalósítása lista alkalmazásával.

Elágazó rekurzió (folyt.): pénzváltás

```
(define (count-change amount)
  (cc amount 5))
(define (cc amount kind-of-coins)
  (cond ((= amount 0) 1)
        ((or (< amount 0) (= kinds-of-coins 0)) 0)
(else (+ (cc amount (- kinds-of-coins 1))
         (cc (- amount
          (first-denomination kinds-of-coins))
        kinds-of-coins)))))
(define (first-denomination kinds-of-coins)
  (cond ((= kinds-of-coins 1) 1)
        ((= kinds-of-coins 2) 5)
        ((= kinds-of-coins 3) 10)
        ((= kinds-of-coins 4) 25)
        ((= kinds-of-coins 5) 50)))
```

Hatványozás

- Az eddig látott folyamatokban a kiértékelési (végrehajtási) lépések száma az adatok n számával lineárisan, ill. exponenciálisan nőtt. Most olyan példa következik, amelyben a lépések száma az n logaritmusával arányos.
- A b szám n-edik hatványának definícióját ugyancsak könnyű átrakni Scheme-be.

$$b^0 = 1$$
$$b^n = b \cdot b^{n-1}$$

```
(define (expt b n)
  (if (= n 0) 1
        (* b (expt (- n 1)))))
```

- A létrejövő folyamat lineáris-rekurzív. $O(n)$ lépés és $O(n)$ méretű tár kell a végrehajtásához.
- A faktoriálisszámításhoz hasonlóan könnyű felírni lineáris-iteratív változatát.

```
(define (expt2 b n) (expt-iter b n 1))

(define (expt-iter b counter product)
        (if (= counter 0) product
 (expt-iter b (- counter 1) (* b product))))
```

- $O(n)$ lépés és $O(1)$ – azaz konstans – méretű tár kell a végrehajtásához.

Hatványozás (folyt.)

- Kevesebb lépés is elég, ha kihasználjuk az alábbi egyenlőségeket:

$b^0 = 1$

$b^n = (b^{n/2})^2$, ha n páros

$b^n = b \cdot b^{n-1}$, ha n páratlan

```
(define (fast-expt b n)
  (cond ((= n 0) 1)
        ((even? n)
 (square (fast-expt b (/ n 2))))
 (else (* b (fast-expt b (- n 1))))))

(define (even? n) (= (remainder n 2) 0))
```

- A lépések száma és a tár mérete $O(\lg n)$-nel arányos. Természetesen ezt is meg lehet írni konstans tárigényű iteratív változatban.

ABSZTRAKCIÓ FÜGGVÉNYEKKEL (ELJÁRÁSOKKAL)

Legnagyobb közös osztó

- Következő példánk 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 a Scheme-eljárás.

$$\gcd(a, 0) = a$$
$$\gcd(a, b) = \gcd(b, a \bmod b)$$

```
(define (gcd a b)
  (if (= b 0) a
      (gcd (remainder a b))))
```

- 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.

Prímteszt

- A `prime` eljárás egy n szám prím voltát teszteli. A `find-divisor` 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 $O(\sqrt{n})$.

```
(define (prim? n)
  (define (smallest-divisor m) (find-divisor m 2))
  (define (find-divisor m test-divisor)
      (cond ((> (square test-divisor) m) m)
            ((divides? test-divisor m) test-divisor)
    (else (find-divisor m (+ test-divisor 1)))))
  (define (divides? a b) (= (remainder b a) 0))
  (= n (smallest-divisor n)))
```

Prímteszt (folyt.)

- A következő Scheme-eljárás egy szám prím voltát *valószínűségi módszerrel* teszteli. A lépések száma $O(\lg n)$.
- Az algoritmus a kis Fermat-tételen 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 algoritmusa ezek után a következő :
  - 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` segédeljárás a `base` szám `exp`-edik hatványának modulo `m` szerinti maradékát adja eredményül.

```
(define (expmod base exp m)
  (cond ((= exp 0) 1)
        ((even? exp)
  (remainder (square (expmod base (/ exp 2) m)) m))
(else
  (remainder (* base (expmod base (- exp 1) m)) m))))
```

- Nagyon hasonló felépítésű `expt-fast`-hoz. A lépések száma a kitevő logaritmusával arányos.
- Ennek segítségével a prímteszt:

```
(define (fermat-test n)
  (define (try-it a)
    (= (expmod a n n) a)
  (try-it (+ 1 (random (- n 1))))))
```

Prímteszt (folyt.)

Innentől nincs átírva...

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.

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.

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
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
```

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.

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` típusvá 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.
  ```

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
```

Függvény fi xpontjának 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:

  $f x, f(f x), f(f(f x)), f(f(f(f x))), \ldots$

  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 fi xpontjának 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 fi xpontjának 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 $fy = x/y$ 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 $y1$, a második $y2 = x/y1$, a harmadik $y3 = x/y2 = x/(x/y1) = y1$. 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 értékként 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 *átlagcsillapításnak* (angolul *average damping)* nevezik.

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 eredényül.
- 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 fi xpont-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.): a fi xpontmódszer 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)
  ```
  ```

ABSZTRAKCIÓ ADATOKKAL

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:

  $n_1/d_1 + n_2/d_2 = (n_1 d_2 + n_2 d_1)/(d_1 d_2)$, $n_1/d_1 - n_2/d_2 = (n_1 d_2 - n_2 d_1)/(d_1 d_2)$,

  $(n_1/d_1)(n_2/d_2) = (n_1 n_2)/(d_1 d_2)$, $(n_1/d_1)/(n_2/d_2) = (n_1 d_2)/(d_1 n_2)$,

  $n_1/d_1 = n_2/d_2$ akkor és csak akkor, ha $n_1 d_2 = n_2 d_1$.

```
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

```
          ------------------------------------------------
---------       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 valamilyen megvalósítása
```

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}
```

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.)

- 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.)

- 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 pozíciójúvá alakíthatók:

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

Adatabsztrakció modulokkal: racionális számok (folyt.)

- A szokásos alapműveleti 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.)

- *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;
```