KIÍRÁS AZ SML-BEN

Kiírás

- `{TextIO.}print : string -> unit`

  `print s` = kiírja az `s` értékét a standard kimenetre, és azonnal kiüríti a puffert.
- `{General.}makestring : numtxt-> string` (Csak mosml!)

  `makestring v` = eredménye a `v` érték *ábrázolása.*
- `{Meta.}printVal : 'a -> 'a` (Csak mosml! Az Alice-ben: `'a -> unit`)

  `printVal e` = kiírja az `e` kifejezés értékét a standard kimenetre pontosan úgy, ahogyan az értelmező írja ki a „legfelső szinten”, és azonnal kiüríti a puffert. Eredményül visszaadja az `e` kifejezés értékét. *Csak interaktív módban használható.*

  1. megjegyzés. A `{` és `}` kapcsos zárójelek között opcionális modulnév áll. Pl. `{TextIO.}print` azt jelenti, hogy a függvény a `TextIO` modulban van definiálva, de az SML-értelmező a `print` nevet rövid alakban is felismeri.

  2. megjegyzés. `numtxt` = `int` | `real` | `word` | `word8` | `char` | `string`
- Különböző típusú egyszerű értékeket alakítanak át füzérré a `toString` függvények:

```
Bool.toString  : bool -> string      String.toString  : string -> string
Char.toString  : char -> string      Time.toString  : time -> string
Date.toString  : date -> string      Word.toString  : word -> string
Int.toString   : int -> string       Word8.toString : word8 -> string
Real.toString  : real -> string
```


Kiírás (folyt.)

- Példák:

```
print("alma"^"Korte\n");
almaKorte
> val it = () : unit

makestring ~5.8e~3;
> val it = "~0.0058" : string

printVal("alma"^"Korte\n");
"almaKorte\n"> val it = "almaKorte\n" : string

makestring("alma"^"Korte\n");
> val it = "\"almaKorte\\n\"" : string
```

- `printVal`-lal tetszőleges típusú érték írható ki, például ennes, rekord vagy lista:

```
printVal (3, 5.0);
(3, 5.0)> val it = (3, 5.0) : int * real

printVal {m2 = 3, m1 = 5.0};
{m1 = 5.0, m2 = 3}> val it = {m1 = 5.0, m2 = 3} : {m1 : real, m2 : int}

printVal [#"A",#"Z",#":"];
[#"A", #"Z", #":"]> val it = [#"A", #"Z", #":"] : char list
```


Kiírás (folyt.)

- `printVal`-lal `datatype`-deklarációval létrehozott típusú érték is kiíratható, pl.

```
datatype t = L | B of t * t;
> New type names: =t
  datatype t = (t,con B : t * t -> t, con L : t)
  con B = fn : t * t -> t
  con L = L : t

val fa = B(B(B(L,B(L,B(L,B(B(L,L),L)))),L),B(L,L));
> val fa = B(B(B(L, B(L, B(L, B(B(L, L), L)))), L), B(L, L)) : t

printVal fa;
B(B(B(L, B(L, B(L, B(B(L, L), L)))), L), B(L, L))> val it = B(...
```

- A kiírt sor túl hosszú, a folytatását . . .-tal helyettesítettük.
- Törjük el a sort a > *válaszjel* előtt *szekvenciális* kifejezés alkalmazásával:

```
(printVal fa; print "\n");
B(B(B(L, B(L, B(L, B(B(L, L), L)))), L), B(L, L))
> val it = () : unit
```

- Sajnos, az eredményül kapott érték nem `fa` értéke!

Kiírás (folyt.)

- Hogyan írjunk ki egy újsor-jelet úgy, hogy az eredmény mégis `fa` értéke legyen? Pl. így:

```
let val res = printVal fa; val _ = print "\n"
in
   res
end;
B(B(B(L, B(L, B(L, B(B(L, L), L)))), L), B(L, L))
> val it = B(B(B(L, B(L, B(L, B(B(L, L), L)))), L), B(L, L)) : t
```

- Ez elég körülményes. A `before` operátort az ilyen esetek kezelésére vezették be:

```
printVal fa before print "\n";
B(B(B(L, B(L, B(L, B(B(L, L), L)))), L), B(L, L))
> val it = B(B(B(L, B(L, B(L, B(B(L, L), L)))), L), B(L, L)) : t
```

- Szekvenciális kifejezés alkalmazásával további magyarázó szöveget írhatunk ki:

```
(print "`fa' értéke =\n"; printVal fa before print "\n");
`fa' értéke =
B(B(B(L, B(L, B(L, B(B(L, L), L)))), L), B(L, L))
> val it = B(B(B(L, B(L, B(L, B(B(L, L), L)))), L), B(L, L)) : t
```


Kiírás (folyt.)

- Hosszú lista, ill. egymásba skatulyázott adatszerkezetek esetén `printVal` (és maga az mosml-értelmező is) alapesetben csak az első 200 listaelemet, ill. legfeljebb 20 szintet ír ki. A hosszat a `printLength`, a szintek számát a `printDepth` *frissíthető változó* szabályozza. Mindkét érték felülírható.

```
printLength : int ref      printLength := 7;  !printLength;
printDepth  : int ref      printDepth  := 3;  !printDepth;
```

- Példák:

```
printLength := 6; printVal [1,2,3,4,5,6,7,8] before print "\n";
[1, 2, 3, 4, 5, 6, ...]
> val it = [1, 2, 3, 4, 5, 6, ...] : int list

printDepth := 4; printVal fa before print "\n";
B(B(#, #), B(#, #))
> val it = B(B(#, #), B(#, #)) : t
```

- Figyelem: a `printLength` és a `!printLength` kifejezések különböznek!

```
printLength;                 | !printLength;
> val it = ref 7 : int ref   | > val it = 7 : int
```


NYOMKÖVETÉS KIÍRÁSSAL: LISTÁK

Nyomkövetés kiírással: `length` (nem iteratív)

- Az mosml-ben nyomkövetés csak a program szövegébe beírt kiíró függvényekkel lehetséges.
- Példa: a `length` függvény két változatának kiértékelése
- A `length` „naív” változata

```
fun length (_::xs) = 1 + length xs
  | length []      = 0
```

- A `length` „naív” változata kiíró függvényekkel (**félkövér** szedéssel az eredeti szöveg látható)

```
fun length ((_ : int) :: xs) =
        printVal(1 + (print " & "; printVal(length(printVal xs))
                                   before print " $ "
                    )
                )
        before print " #\n"
  | length []      = (print " * "; printVal 0
                                   before print " %\n")
```

Nyomkövetés kiírással: `lengthi` (iteratív)

- A `length` iteratív változata

```
fun lengthi xs = let fun len (i, _::xs) = len(i+1, xs)
                       | len (i, [])    = i
                 in len(0, xs)
                 end
```

- A `length` iteratív változata kiíró függvényekkel (**félkövér** szedéssel az eredeti szöveg látható)

```
fun lengthi xs =
    let fun len (i, (_ : int) :: xs) =
            len((print " "; printVal((printVal i
                                       before print " $ ") + 1)),
                (print " & "; printVal xs)
               )
                before print "#\n"
          | len (i, []) = (print " * "; printVal i
                                        before print " %\n")
    in len(0, xs)
    end
```


Nyomkövetés kiírással: `length` egy alkalmazása

- `length` egy alkalmazása

```
fun length ((_ : int) :: xs) =
        printVal(1 + (print " & "; printVal(length(printVal xs))
                                   before print " $ "
                     )
                 )
        before print " #\n"
  | length []      = (print " * "; printVal 0
                                   before print " %\n");

length [1,2,3];
 & [2, 3] & [3] & [] * 0 %
0 $ 1 #
1 $ 2 #
2 $ 3 #
```


Nyomkövetés kiírással: `lengthi` egy alkalmazása

- `lengthi` egy alkalmazása

```
fun lengthi xs =
    let fun len (i, (_ : int) :: xs) =
            len((print " "; printVal((printVal i
                                       before print " $ ") + 1)),
                (print " & "; printVal xs)
               )
                before print "#\n"
          | len (i, []) = (print " * "; printVal i
                                        before print " %\n")
    in len(0, xs)
    end

lengthi [1,2,3];
 0 $ 1 & [2, 3] 1 $ 2 & [3] 2 $ 3 & [] * 3 %
#
#
#
```


Nyomkövetés kiírással: `length` és `lengthi` összehasonlítása

- `length` és `lengthi` kiértékelésének összehasonlítása

```
length [1,2,3];
 & [2, 3] & [3] & [] * 0 %
0 $ 1 #
1 $ 2 #
2 $ 3 #
```

```
lengthi [1,2,3];
0 $ 1 & [2, 3] 1 $ 2 & [3] 2 $ 3 & [] * 3 %
#
#
#
```

HIBAKERESÉS ÉS NYOMKÖVETÉS POLY/ML-BEN

Hibakeresés és nyomkövetés Poly/ML-ben

- Töréspontot elhelyezni csak olyan segédfüggvényben lehet, amelyet egy másik függvény törzsében egy `let`-kifejezésben definálunk (példákat a következő diákon mutatunk).
- A hibakeresést és nyomkövetést *először* a `PolyML.Compiler.debug` kapcsoló `true` értékre állításával engedélyezni kell (ld. a következő diát), *majd* definiálni kell azokat a függvényeket, amelyek működését követni akarjuk, vagy amelyekben töréspontot akarunk elhelyezni. (Ellenkező esetben a hibakereséshez és nyomkövetéshez szükséges plusz kódot a Poly/ML értelmező nem írja bele a lefordított programrészbe.)
- A politípusúnak definiált nevek értékét nem tudják kiírni a hibakeresés kiíró függvényei (`PolyML.Debug.variables()`, `PolyML.Debug.dump()`, `PolyML.Debug.stack()` – lásd a következő diákon). Ahhoz, hogy egy név értékét ezek a függvények kiírják, a névnek *már a függvény definiálásakor* monotípusúnak kell lennie. (Egy nevet, ha a szövegkörnyezetből nem vezethető le a típusa, *típusmegkötéssel* tehetünk monotípusúvá.)
- A Poly/ML fontosabb hibakereső függvényeit lásd a következő dián (konkretizálva a `length` és a `len` függvényre utaló hivatkozásokkal).
- A `PolyML.profiling : int -> unit` függvénnyel egy kifejezés kiértékelési idejét, ill. egyes függvények futási idejét és helyfoglalását monitorozhatjuk (részletek a Poly/ML-leírásban olvashatók).

Hibakeresés és nyomkövetés Poly/ML-ben (folyt.)

`PolyML.Compiler.debug := true;`	Hibakeresés engedélyezése; engedélyezett
`open PolyML.Debug;`	állapotban kell definiálni a vizsgálandó függvényt
`breakIn "len";`	Töréspont elhelyezése, rövid változat
`breakIn "length()len";`	Töréspont elhelyezése, teljes változat
`continue();`	Folytatás a töréspont következő előfordulásáig
`down();`	Áttérés az előző hívási szintre a veremben
`up();`	Áttérés a következő hívási szintre a veremben
`dump();`	A verem teljes tartalmának kiírása
`stack();`	A hívások kiírása a veremtartalom alapján
`variables();`	A nevek értékének kiírása
`clearIn "length()len";`	Töréspont törlése, teljes változat
`trace true;`	Nyomkövetés bekapcsolása
`step();`	Adott hívási szinten tovább vagy beljebb
`stepOver();`	Adott hívási szinten tovább
`stepOut();`	Előző hívási szintre vissza

`breakIn` és `clearIn` konkrétan a `length` és a `len` függvényre hivatkozik a fenti felsorolásban.
Használatukról részletesebben itt lehet olvasni: `<http://www.polyml.org/docs/Debugging.html>`.

Első példa a Poly/ML debugger használatára

```
(* length : 'a list -> int
   length zs = a zs elemeinek száma *)
fun length zs =
      let (* len : 'a list -> int
             len xs = az xs elemeinek száma *)
          fun len []        = 0
            | len (_ :: xs) = 1 + len xs
      in
          len zs
      end;

breakIn "length()len";
val it = () : unit

length(explode "abcde");
 line:7 function:length()len
debug>

variables();
val xs = [?, ?, ?, ?] val zs = [?, ?, ?, ?, ...]
val it = () : unit
debug>
```

- Hmm. Politípusúnak definiált értéket a Poly/ML hibakeresője nem tud kiírni?

Első példa a Poly/ML debugger használatára (folyt.)

- Nem bizony! A nevek típusát pl. *típusmegkötéssel* meg kell adni ahhoz, hogy az értékek kiírásához szükséges kód fordítási időben beépüljön a lefordított programba.

```
(* length : char list -> int *)
fun length (zs : char list) =
      let (* len : char list -> int *)
          fun len []         = 0
            | len (_ :: xs : char list) = 1 + len xs
      in
          len zs
      end;
```

```
breakIn "len";
length(explode "abcde");
variables();
...
val xs = [#"b", #"c", #"d", #"e"] val zs = [#"a", #"b", #"c", #"d", ...]
...
```

```
continue(); variables();
...
 line:5 function:length()len
debug> val xs = [#"c", #"d", #"e"] val zs = [#"a", #"b", #"c", #"d", ...]
...
```


Első példa a Poly/ML debugger használatára (folyt.)

- `length` egy hibás változata a hibakeresés kipróbálásához

```
(* length : 'a list -> int
   length zs = a zs elemeinek száma *)
fun length zs =
      let (* len : 'a list * int -> int
              len xs = n + az xs elemeinek száma -- HIBÁS! *)
          fun len ([], n)      = n
            | len (_ :: xs, n) = len(xs, n)
      in len(zs,0)
      end;
```

- `length` egy másik hibás változata a hibakeresés kipróbálásához

```
(* length : char list -> int
   length zs = a zs elemeinek száma *)
fun length (zs : char list) =
      let (* len : char list * int -> int
              len xs = n + az xs elemeinek száma -- HIBÁS! *)
          fun len ([], n)      = n
            | len (_ :: xs : char list, n : int) = len(xs, n)
      in len(zs,0)
      end;
```


Második példa a Poly/ML debugger használatára

```
(* maxl : (string * string -> string) -> string list -> string
   maxl max zs = a zs lista max szerint legnagyobb eleme *)
fun maxl max zs = let fun mxl [] = NONE
                        | mxl [n] = SOME n
                        | mxl (n::m::ns) = mxl(max(n:string, m)::ns)
                  in
                      mxl zs
                  end;

(* stringMax : string * string -> string
   stringMax (s, t) = s és t közül a nagyobbik *)
fun stringMax (s : string, t) = if s > t then s else t;
```

- Hibakeresés töréspont elhelyezésével

```
breakIn "mxl";
maxl stringMax ["ec","pec","kimehetsz","holnaputan","bejohetsz"];
 line:1 function:
debug>
```

```
continue();
val it = () : unit
 line:6 function:maxl()mxl
debug>
```


Második példa a Poly/ML debugger használatára (folyt.)

```
step();
val it = () : unit
 line:4 function:stringMax
debug>
```

```
step();
val it = () : unit
 line:6 function:maxl()mxl
debug>
```

```
step();
val it = () : unit
 line:4 function:stringMax
debug>
```

```
variables();
val t = "kimehetsz" val s = "pec"
val it = () : unit
debug>
```

```
step();
val it = () : unit
 line:6 function:maxl()mxl
debug>
```

Második példa a Poly/ML debugger használatára (folyt.)

```
stepOver();
val it = () : unit
 line:6 function:maxl()mxl
debug>

variables();
val n = "pec" val m = "bejohetsz" val ns = []
val zs = ["ec", "pec", "kimehetsz", "holnaputan", ...] val max = fn
...

continue();
val it = () : unit
 line:5 function:maxl()mxl
debug>

stack();
 line:5 function:maxl()mxl
 line:6 function:maxl()mxl
 line:6 function:maxl()mxl
 line:6 function:maxl()mxl
 line:6 function:maxl()mxl
 line:8 function:maxl
...
```


Második példa a Poly/ML debugger használatára (folyt.)

```
variables();
val n = "pec" val zs = ["ec", "pec", "kimehetsz", "holnaputan", ...]
val max = fn
...

dump();
Function maxl()mxl: val n = "pec" val zs =
  ["ec", "pec", "kimehetsz", "holnaputan", ...] val max = fn
Function maxl()mxl: val n = "pec" val m = "bejohetsz" val ns = []
val zs = ["ec", "pec", "kimehetsz", "holnaputan", ...] val max = fn
Function maxl()mxl: val n = "pec" val m = "holnaputan" val ns = ["bejohetsz"]
val zs = ["ec", "pec", "kimehetsz", "holnaputan", ...] val max = fn
Function maxl()mxl: val n = "pec" val m = "kimehetsz"
val ns = ["holnaputan", "bejohetsz"]
val zs = ["ec", "pec", "kimehetsz", "holnaputan", ...] val max = fn
Function maxl()mxl: val n = "ec" val m = "pec"
val ns = ["kimehetsz", "holnaputan", "bejohetsz"]
val zs = ["ec", "pec", "kimehetsz", "holnaputan", ...] val max = fn
Function maxl: val mxl = fn
val zs = ["ec", "pec", "kimehetsz", "holnaputan", ...] val max = fn

val it = () : unit
debug>
```


Második példa a Poly/ML debugger használatára (folyt.)

- Nyomkövetés

```
clearIn "mxl";
trace true;
maxl stringMax ["ec","pec","kimehetsz","holnaputan","bejohetsz"];
 maxl entered
 val zs = ["ec", "pec", "kimehetsz", "holnaputan", ...]
 val max = fn


  maxl()mxl entered
  val n = "ec"
  val m = "pec"
  val ns = ["kimehetsz", "holnaputan", "bejohetsz"]

   stringMax entered val t = "pec" val s = "ec"
   stringMax returned "pec"

   maxl()mxl entered
   val n = "pec"
   val m = "kimehetsz"
   val ns = ["holnaputan", "bejohetsz"]
```


Második példa a Poly/ML debugger használatára (folyt.)

```
    stringMax entered val t = "kimehetsz" val s = "pec"
    stringMax returned "pec"

    maxl()mxl entered
    val n = "pec"
    val m = "holnaputan"
    val ns = ["bejohetsz"]

     stringMax entered val t = "holnaputan" val s = "pec"
     stringMax returned "pec"
     maxl()mxl entered val n = "pec" val m = "bejohetsz" val ns = []
      stringMax entered val t = "bejohetsz" val s = "pec"
      stringMax returned "pec"
      maxl()mxl entered val n = "pec"
      maxl()mxl returned SOME "pec"
     maxl()mxl returned SOME "pec"
    maxl()mxl returned SOME "pec"
   maxl()mxl returned SOME "pec"
  maxl()mxl returned SOME "pec"
 maxl returned SOME "pec"
val it = SOME "pec" : string option
```

ABSZTRAKCIÓ FÜGGVÉNYEKKEL (ELJÁRÁSOKKAL)

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:

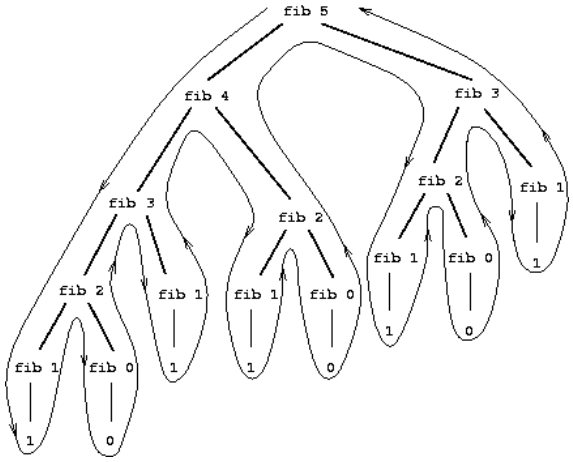
```
0, 1, 1, 2, 3, 5, 8, 13, 21, ...
```

- A Fibonacci-számok matematikai definíciója könnyen átírható SML-függvénnyé:

$F(0) = 0$
$F(1) = 1$
$F(n) = F(n-1) + F(n-2)$, ha $n > 1$
	`fun fib 0 = 0`
	`  | fib 1 = 1`
	`  | fib n = fib(n-1) + fib(n-2)`

  Emlékeztetőül: a `fib` függvény definíciójában a 3. klóznak az utolsónak kell lennie, mert az `n` minta minden argumentumra illeszkedik.
- 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 SML-függvény egy változata:

```
fun fib n = let fun fibIter (i, b, a) =
                  if i = n then b
                  else fibIter(i+1, a, a+b)
            in
                fibIter(0, 0, 1)
            end
```

- *Mintaillesztést* használhatunk, ha `i`-t nem növeljük, hanem `n`-től 0-ig csökkentjük.

  Figyelem: a klózok sorrendje, mivel nem egymást kizáróak a minták, lényeges!

```
fun fib n = let fun fibIter (0, b, a) = b
                  | fibIter (i, b, a) = fibIter(i-1, a, a+b)
            in
                fibIter(n, 0, 1)
            end
```


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 `firstDenomination` (magyarul *első címlet)* függvényt felsorolással valósítottuk meg. Tömörebb és rugalmasabb lenne a megvalósítása lista alkalmazásával.

Elágazó rekurzió (folyt.): pénzváltás

```
fun countChange amount =
  let (* cC amount kindsOfCoins = az amount összes felváltásainak száma
                                  kindsOfCoins db érmével *)
      fun cC (amount, kindsOfCoins) =
            if amount < 0 orelse kindsOfCoins = 0 then 0
            else if amount = 0 then 1
            else cC (amount, kindsOfCoins - 1) +
                 cC (amount - firstDenomination kindsOfCoins, kindsOfCoins)
      and firstDenomination 1 = 1
        | firstDenomination 2 = 5
        | firstDenomination 3 = 10
        | firstDenomination 4 = 25
        | firstDenomination 5 = 50
  in
     cC(amount, 5)
  end;

countChange 10 = 4; countChange 100 = 292;
```

Gyakorló feladatok.

- Írja át a `firstDenomination` függvényt úgy, hogy a címleteket egy lista tartalmazza.
- Írja meg a `cC` függvény mintaillesztést használó változatát!

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 SML-be.

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

```
fun expt (b, 0) = 1
  | expt (b, n) = b * expt(b, 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.

```
fun expt (b, n) =
      let fun exptIter (0, product) = product
            | exptIter (counter, product) =
                              exptIter (counter-1, b * product)
      in
          exptIter(n, 1)
      end
```

- $O(n)$ 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

```
fun expt (b, n) =
  let fun exptFast 0 = 1
        | exptFast n =
            if even n
            then square(exptFast(n div 2))
            else b * exptFast(n-1)
      and even i = i mod 2 = 0
      and square x = x * x
  in  exptFast n  end
```

- A lépések száma és a tár mérete $O(\lg n)$-nel arányos. Konstans tárigényű iteratív változata:

```
fun expt (b, 0) = 1   (* Nem hagyható el! Miért nem? *)
  | expt (b, n) = let fun exptFast (1, r) = r
                        | exptFast (n, r) =
                            if even n then exptFast(n div 2, r*r)
                            else           exptFast(n-1, r*b)
                      and even i = i mod 2 = 0
                  in  exptFast(n, b)  end
```


LISTÁK

Adott számú elem egy lista elejéről és végéről (`take`, `drop`)

- Legyen $xs = [x_0, x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_{n-1}]$, akkor `take(xs, i)` $= [x_0, x_1, \ldots, x_{i-1}]$ és `drop(xs, i)` $= [x_i, x_{i+1}, \ldots, x_{n-1}]$.
- `take` egy megvalósítása (jobbrekurzív-e? jobbrekurzívvá tehető-e? robosztus-e?)

```
(* take : 'a list * int -> 'a list
   take (xs, i) = ha i < 0, xs;, ha i >= 0,
                  az xs első i db eleméből álló lista *)
fun take (_, 0)      = []
  | take ([], _)     = []
  | take (x::xs, i) = x :: take(xs, i-1)
```

- `drop` egy megvalósítása (jobbrekurzív-e? jobbrekurzívvá tehető-e? robosztus-e?)

```
(* drop : 'a list * int -> 'a list
   drop(xs, i) = ha i < 0, xs; ha i >= 0,
                 az xs első i db elemének eldobásával előálló lista *)
fun drop ([], _)     = []
  | drop (x::xs, i) = if i > 0 then drop (xs, i-1) else x::xs
```

- Könyvtári változatuk – `List.take`, ill. `List.drop` – ha az `xs` listára alkalmazzuk, $i < 0$ vagy $i >$ `length xs` esetén `Subscript` néven kivételt jelez.

Lista redukciója kétoperandusú művelettel

Idézzük föl az egészlista maximális értékét megkereső `maxl` függvény két változatát:

- `maxl` jobbról balra egyszerűsítő (nem jobbrekurzív) változata

```
(* maxl : int list -> int
   maxl ns = az ns egészlista legnagyobb eleme
*)
fun maxl [] = raise Empty
  | maxl [n] = n
  | maxl (n::ns) = Int.max(n, maxl ns)
```

- `maxl` balról jobbra egyszerűsítő (jobbrekurzív) változata:

```
(* maxl' : int list -> int
   maxl' ns = az ns egészlista legnagyobb eleme
*)
fun maxl' [] = raise Empty
  | maxl' [n] = n
  | maxl' (n::m::ns) = maxl'(Int.max(n,m)::ns)
```

- Amint ez a példa is mutatja, vissza-visszatérő feladat egy lista redukciója kétoperandusú művelettel.
- Közös bennük, hogy n db értékből egyetlen értéket kell előállítani, ezért is beszélünk *redukcióról*.

Lista redukciója kétoperandusú művelettel (`foldr`, `foldl`)

- `foldr` jobbról balra, `foldl` balról jobbra haladva egy kétoperandusú műveletet (pontosabban egy *párra alkalmazható, prefix* pozíciójú *függvényt*) alkalmaz egy listára. Példák szorzat és összeg kiszámítására:

```
foldr op* 1.0 [] = 1.0;                    foldl op+ 0 [] = 0;
foldr op* 1.0 [4.0] = 4.0;                 foldl op+ 0 [4] = 4;
foldr op* 1.0 [1.0, 2.0, 3.0, 4.0] = 24.0; foldl op+ 0 [1, 2, 3, 4] = 10;
```

- Jelöljön $\oplus$ tetszőleges kétoperandusú infix operátort. Akkor

  foldr op$\oplus$ e [x_1, x_2, ..., x_n] = (x_1 $\oplus$ (x_2 $\oplus$ ... $\oplus$ (x_n $\oplus$ e) ...))
  foldr op$\oplus$ e [] = e
  foldl op$\oplus$ e [x_1, x_2, ..., x_n] = (x_n $\oplus$ ... $\oplus$ (x_2 $\oplus$ (x_1 $\oplus$ e)) ...)
  foldl op$\oplus$ e [] = e

- A $\oplus$ művelet e operandusa néhány gyakori műveletben – összeadás, szorzás, konjunkció (logikai „és”), alternáció (logikai „vagy”) – a (jobb oldali) *egységelem* szerepét tölti be.
- Ha a művelet asszociatív és kommutatív, `foldr` és `foldl` eredménye azonos.

Példák `foldr` és `foldl` alkalmazására

- `isum` egy egészlista elemeinek összegét, `rprod` egy valóslista elemeinek szorzatát adja eredményül.

```
val isum = foldr op+ 0;          val rprod = foldr op* 1.0;
val isum = foldl op+ 0;          val rprod = foldl op* 1.0;
```

- A `length` függvény is definiálható `foldl`-lel vagy `foldr`-rel. Kétoperandusú műveletként olyan segédfüggvényt (`inc`) alkalmazunk, amelyik *nem használja* az első paraméterét.

```
(* inc : 'a * int -> int
   inc (_, n) = n + 1 *)
fun inc (_, n) = n + 1;

(* lengthl, lengthr : 'a list -> int *)
val lengthl = fn ls => foldl inc 0 ls;
fun lengthr ls = foldr inc 0 ls;
val lengthl = foldl inc 0;

lengthl (explode "tengertanc");
lengthr (explode "hajdu sogor");
```


Lista: `foldr` és `foldl` definíciója

- foldr op$\oplus$ e [x_1, x_2, ..., x_n] = (x_1 $\oplus$ (x_2 $\oplus$ ... $\oplus$ (x_n $\oplus$ e) ...))
  foldr op$\oplus$ e [] = e

```
(* foldr f e xs = az xs elemeire jobbról balra haladva
                  alkalmazott, kétoperandusú, e egységelemű
                  f művelet eredménye
   foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b *)
fun foldr f e (x::xs) = f(x, foldr f e xs)
  | foldr f e [] = e;
```

- foldl op$\oplus$ e [x_1, x_2, ..., x_n] = (x_n $\oplus$ ... $\oplus$ (x_2 $\oplus$ (x_1 $\oplus$ e)) ...)
  foldl op$\oplus$ e [] = e

```
(* foldl f e xs = az xs elemeire balról jobbra haladva
                  alkalmazott, kétoperandusú, e egységelemű
                  f művelet eredménye
   foldl : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b *)
fun foldl f e (x::xs) = foldl f (f(x, e)) xs
  | foldl f e [] = e;
```

További példák `foldr` és `foldl` alkalmazására

- Egy lista elemeit egy másik lista elé fűzi `foldr` és `foldl`, ha kétoperandusú műveletként a *cons* konstruktorfüggvényt – azaz az `op::`-ot – alkalmazzuk.

  foldr op:: ys [x_1, x_2, x_3] = (x_1 :: (x_2 :: (x_3 :: ys)))

  foldl op:: ys [x_1, x_2, x_3] = (x_3 :: (x_2 :: (x_1 :: ys)))

- A `::` nem asszociatív és kommutatív, ezért `foldl` és `foldr` eredménye különböző!

```
(* append : 'a list -> 'a list -> 'a list
   append xs ys = az xs ys elé fűzésével előálló lista *)
fun append xs ys = foldr op:: ys xs;

(* revApp : 'a list -> 'a list -> 'a list
   revApp xs ys = a megfordított xs ys elé fűzésével
                  előálló lista *)
fun revApp xs ys = foldl op:: ys xs;
```

`append [1, 2, 3] [4, 5, 6] = [1, 2, 3, 4, 5, 6];` (vö. Prolog: `append`)

`revApp [1, 2, 3] [4, 5, 6] = [3, 2, 1, 4, 5, 6];` (vö. Prolog: `revapp`)

További példák `foldr` és `foldl` alkalmazására

- `maxl` két megvalósítása

```
(* maxl : ('a * 'a -> 'a) -> 'a list -> 'a
   maxl max ns = az ns lista max szerinti legnagyobb eleme
*)

(* nem jobbrekurzív *)

fun maxl max []      = raise Empty
  | maxl max (n::ns) = foldr max n ns

(* jobbrekurzív *)

fun maxl' max []      = raise Empty
  | maxl' max (n::ns) = foldl max n ns
```


Példa listák használatára: futamok előállítása

- A futam egy olyan lista, amelynek az elemei egy adott feltételnek megfelelnek.
- Az adott feltételt az előző és az aktuális elemre alkalmazandó *predikátumként* adjuk át a futamot előállító fügvénynek.
- A feladat: írjunk olyan SML függvényt, amely (az elemek eredeti sorrendjének megőrzésével) egy lista egymás utáni elemeiből képzett futamok listáját adja eredményül.
- Az első változatban egy-egy segédfüggvényt írunk egy lista első (prefix) futamának, valamint a maradéklistának az előállítására.
- A `futam` segédfüggvénynek két argumentuma van: az első egy predikátum, amely a kívánt feltételt megvalósítja, a második pedig egy pár. A pár első tagja az előző elem, a második tagja pedig az a lista, amelynek az előző elemmel induló futamát kell `futam`-nak előállítania.
- A `maradek` segédfüggvény két argumentuma azonos `futam` argumentumaival. Eredményül azt a listát kell visszaadnia, amelyet az első futam leválasztásával állít elő a pár második tagjaként átadott listából.
- A következő diákon a `futam` és a `maradek` segédfüggvények, valamint a `futamok` függvény különböző változatai láthatók.

Példa listák használatára: futamok előállítása (folyt.)

- Első változat: futam és maradék előállítása két függvénnyel

```
(* futam : ('a * 'a -> bool)  -> ('a * 'a list) -> 'a list
   futam p (x, ys) = az x::ys p-t kielégítő első (prefix) futama *)
fun futam p (x, [])     = [x]
  | futam p (x, y::ys) = if p(x, y) then x :: futam p (y, ys) else [x]

(* maradek : ('a * 'a -> bool) ->  ('a * 'a list) -> 'a list
   maradek p (x, ys) = az x::ys p-t kielégítő futama utáni maradéka *)
fun maradek p (x, [])           = []
  | maradek p (x, yys as y::ys) =
                  if p(x ,y) then maradek p (y, ys) else yys

(* futamok1 : ('a * 'a -> bool) -> 'a list -> 'a list list
   futamok1 p xs = az xs p-t kielégítő futamaiból álló lista *)
fun futamok1 p []      = []
  | futamok1 p (x::xs) =
      let val fs = futam p (x, xs)
          val ms = maradek p (x, xs)
      in
          if null ms then [fs] else fs :: futamok1 p ms
      end
```

Példa listák használatára: futamok előállítása (folyt.)

- Hatékonyságot rontó tényezők
  1. `futamok1` kétszer megy végig a listán: először `futam`, azután `maradek`,
  2. `p`-t, bár sohasem változik, paraméterként adjuk át `futam`-nak és `maradek`-nak,
  3. egyik függvény sem használ akkumulátort.
- Javítási lehetőségek
  1. `futam` egy párt adjon eredményül, ennek első tagja legyen a futam, második tagja pedig a maradék; a futam elemeinek gyűjtésére használjunk akkumulátort,
  2. `futam` legyen lokális `futamok2`-n belül,
  3. az `if null ms then [fs] else` szövegrész törölhető: a rekurzió egy hívással később mindenképpen leáll.

Példa listák használatára: futamok előállítása (folyt.)

- Második változat: futam és maradék előállítása egy lokális függvénnyel

```
(* futamok2 : ('a * 'a -> bool) -> 'a list -> 'a list list
   futamok2 p xs = az xs p-t kielégítő futamaiból álló lista
*)
fun futamok2 p []      = []
  | futamok2 p (x::xs) =
      let (* futam : ('a * 'a list) -> 'a list * 'a list
             futam (x, ys) zs = olyan pár, amelynek első tagja az x::ys p-t
                                kielégítő első (prefix) futama a zs elé
                                fűzve, második tagja pedig az x::ys maradéka
          *)
          fun futam (x, []) zs           = (rev(x::zs), [])
            | futam (x, yys as y::ys) zs = if p(x, y)
                                             then futam (y, ys) (x::zs)
                                           else (rev(x::zs), yys);
          val (fs, ms) = futam (x, xs) []
      in
          fs :: futamok2 p ms
      end
```


Példa listák használatára: futamok előállítása (folyt.)

- Harmadik változat: az egyes futamokat és a futamok listáját is gyűjtjük

```
(* futamok3 : ('a * 'a -> bool) -> 'a list -> 'a list list
   futamok3 p xs = az xs p-t kielégítő futamaiból álló lista
*)
fun futamok3 p []      = []
  | futamok3 p (x::xs) =
      let (* futamok : ('a * 'a list) -> 'a list -> 'a list * 'a list
             futamok (x, ys) zs zss = az x::ys p-t kielégítő futamaiból
                                      álló lista zss elé fűzve
          *)
          fun futamok (x, []) zs zss           = rev(rev(x::zs)::zss)
            | futamok (x, yys as y::ys) zs zss =
                if p(x, y)
                    then futamok (y, ys) (x::zs) zss
                else futamok (y, ys) [] (rev(x::zs)::zss)
      in
          futamok (x, xs) [] []
      end;
```


Példa listák használatára: futamok előállítása (folyt.)

- Példák a függvények alkalmazására:

```
futam   op<= (1, [9,19,3,4,24,34,4,11,45,66,13,45,66,99]) =
    [1,9,19];

maradek  op<= (1, [9,19,3,4,24,34,4,11,45,66,13,45,66,99]) =
    [3,4,24,34,4,11,45,66,13,45,66,99];

futamok1 op<= [1,9,19,3,4,24,34,4,11,45,66,13,45,66,99] =
    [[1,9,19], [3,4,24,34], [4,11,45,66], [13,45,66,99]];

futamok1 op<= [99,1] = [[99], [1]];

futamok1 op<= [99] = [[99]];

futamok1 op<= [] = [];
```

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 *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.)

- Tegyük föl, hogy
  - van olyan *konstruktorműveletünk,* amely egy `n` számlálóból és egy `d` nevezőből létrehozza a racionális számot: `makeRat(n,d)`, továbbá
  - van egy-egy olyan *szelektorműveletünk*, amelyek egy `q` racionális szám számlálóját, ill. nevezőjét előállítják: `num q`, `den q`.
- A jól ismert műveleteket írjuk át SML-programmá:

  $n_1/d_1 + n_2/d_2 = (n_1d_2 + n_2d_1)/(d_1d_2)$, $n_1/d_1 - n_2/d_2 = (n_1d_2 - n_2d_1)/(d_1d_2)$,
  $(n_1/d_1)(n_2/d_2) = (n_1n_2)/(d_1d_2)$, $(n_1/d_1)/(n_2/d_2) = (n_1d_2)/(d_1n_2)$,
  $n_1/d_1 = n_2/d_2$ akkor és csak akkor, ha $n_1d_2 = n_2d_1$.

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


Adatabsztrakció: racionális számok (folyt.)

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

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

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

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

Adatabsztrakció: racionális számok (folyt.)

- Ezzel racionális számokat megvalósító programunk első változata kész is van. A teljes program:

```
type rat = int * int;
fun makeRat (n, d) = (n, d) : rat;
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");
```


Adatabsztrakció: racionális számok (folyt.)

- 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; a szelektorműveleteken nem változtatunk:

```
(* gcd : int * int -> int
   gcd (a, b) = a és b legnagyobb közös osztója
*)
fun gcd (a, 0) = a
  | gcd (a, b) = gcd(b, a mod b)

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

- 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 megvalósítása SML-ben
```

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
```