sapientia
tankönyvek

Márton Gyöngyvér

Funkcionális programozás Haskell-alapismeretek

Scientia Kiadó

MÁRTON GYÖNGYVÉR

FUNKCIONÁLIS PROGRAMOZÁS
HASKELL-ALAPISMERETEK

SAPIENTIA ERDÉLYI MAGYAR TUDOMÁNYEGYETEM
MAROSVÁSÁRHELYI KAR
MATEMATIKA–INFORMATIKA TANSZÉK

MÁRTON GYÖNGYVÉR

FUNKCIONÁLIS PROGRAMOZÁS

HASKELL-ALAPISMERETEK

Scientia Kiadó
Kolozsvár · 2021

sapientia tankönyvek

informatika

Megvalósult a Magyar Kormány Támogatásával

Miniszterelnökség Nemzetpolitikai Államtitkárság

Bethlen Gábor Alap

Felelős kiadó:
dr. Sorbán Angella

Lektorok:
Donkó István (Budapest)
Kaposi Ambrus (Budapest)
Kovács András (Budapest)

Borítóterv:
Tipotéka Kft.

Kiadói koordinátor:
Szabó Beáta

A szakmai felelősséget teljes mértékben a szerző vállalja.

Első magyar nyelvű kiadás: 2021

Descrierea CIP a Bibliotecii Naţionale a României
MÁRTON, GYÖNGYVÉR
Funkcionális programozás : Haskell-alapismeretek / Márton Gyöngyvér. - Cluj-Napoca : Scientia, 2021
Conţine bibliografie
ISBN 978-606-975-050-6

004

TARTALOMJEGYZÉK

CONTENTS

CUPRINS

1. fejezet

Bevezető

Jelen egyetemi jegyzet célja azon funkcionális programozási alapismeretekbe bevezetni az olvasót, amelyekre elengedhetetlenül szükség van az informatika világában. A jegyzet kifejezetten főiskolás diákoknak készült, és azokat a kérdésköröket ismerteti, amelyeket az elméleti és gyakorlati órákon a hallgatóknak el kell sajátítaniuk. A bemutatásra kerülő programozási nyelv a Haskell lesz, amelynek elsajátítása érdekében elemi feladatok bemutatásával indítunk, majd fokozatosan térünk át bonyolultabb nyelvi elemekre és komplexebb algoritmusok tárgyalására. A jegyzet megértéséhez nem szükséges különösebb programozói tapasztalat, algoritmikus gondolkodás, hiszen a jegyzet tanulmányozásával azt szeretnénk elérni, hogy a diákokban pontosan ezek a készségek fejlődjenek ki. A jegyzetben bemutatásra kerülő Haskell-kódok letölthetők a következő linkről:

https://ms.sapientia.ro/~mgyongyi/Funk_Log/Jegyzet/Jegyzet.zip

A jegyzet megírásához nagy segítséget nyújtottak a könyvészetben megjelenő szakkönyvek, weboldalak, éppen ezért legtöbbjükről néhány mondat erejéig egy rövid összefoglalót közlünk.

Richard Bird[2] könyve matematikai szemszögből közelíti meg a Haskellben való programozást, ahol a szerző elsősorban az algoritmikus gondolkodás kifejlesztésére helyezi a hangsúlyt. Mondhatjuk azt is, hogy összehasonlítja a matematikus és programozói látásmódot. A könyv számos kitűzött feladatot és azoknak az egyszerű és elegáns megoldásait is tartalmazza.

Hal Daume[4] tutoriálja, ahogy a cím is jelzi, egy viszonylag rövid Haskell-ismertető, tartalmazza a Haskell-alapismeretek gyors elsajátításához szükséges információkat.

Diviánszky Péter[5] oldala, funkcionális programozással kapcsolatos előadások mellett, számos érdekes kitűzött feladatot tartalmaz.

Graham Hutton[8] könyve, habár nem nagy terjedelmű, igazi szakértői munka. Pontos magyarázatok és példák segítik az olvasót a különböző fogalmak gyors elsajátításában. A fejezetek végén rövid összefoglalót, olvasásra ajánlott könyvcímeket, illetve megoldásra javasolt feladatokat is találunk. A szerző nem bocsátkozik részletes leírásokba, de ezek inkább segítik az olvasót abban, hogy minél hamarabb kezdjen el önállóan programozni.

Miran Lipovača[10] könyvét nagyon szokták szeretni a diákok. Könnyedén, viccesen illusztrálva vezeti be az olvasót a nehezebb Haskell-nyelvi elemek világába is. Az első lépések elsajátításában valóban hasznos könyvnek számít.

Alejandro Serrano Mena[11] munkájának nagyobb Haskell-projektek írásakor lehet hasznát venni. Tulajdonképpen egy funkcionális szoftver megírásához ad kitűnő segítséget, mert az első oldalaktól kezdődően részletesen kitér, magyarázza és intenzíven használja a különböző Haskell-könyvtárakat.

Nyékyné Gaizler Judit[12] könyve alapos összehasonlítása a különböző programozási nyelveknek, nyelvi eszközöknek. Egy viszonylag rövid fejezet a funkcionális programozás elemeit mutatja be, számos lényeges, megoldott feladattal segítve az olvasót.

Bryan O'Sullivan és társai[13] műve egy átfogó, részletes, minden Haskell-nyelvi elemre kitérő munka. Megtanítja gyakorlati célokra használni a Haskellt úgy, hogy közben a funkcionális programozási stílust is magáévá teszi az olvasó. Tanulmányozásával az alapoktól kezdve lehet elsajátítani a Haskellt úgy, hogy közben az egészen komplex alkalmazások fejlesztéséig is el lehet jutni. Online változata ingyenesen elérhető.

Simon Thompson[14] könyvének első kiadása még 1996-ban jelent meg és az akkori funkcionális programozói trendet követte. Az első volt azon könyvek egyikének, amely tartalmazta a Haskell elsajátításához szükséges alapelemeket, olyan formában, hogy azokat könnyedén tudják elsajátítani azok is, akik kevésbé jártasak a programozás világában. A harmadik kiadásban több minden más szemszögből kerül bemutatásra, de könnyen érthető stílusát továbbra is megtartotta.

Szükségesnek tartjuk még megjegyezni, hogy a jegyzetben bemutatásra kerülő azonosítóknak, paramétereknek, illetve függvényeknek szándékosan adtunk magyar neveket, szemben azzal a szokással, ami a programozók világában megszokott, ahol kizárólagosan angol neveket használnak erre a célra. Az oktatói tapasztalat azt mutatja, hogy kezdő programozóknak kifejezetten

megerőltető különbséget tenni a beépített és bemutatásra szánt függvények között, ez pedig elsősorban úgy könnyíthető meg, ha az elnevezések szintjén különbséget teszünk.

2. fejezet

Programozási paradigmák

A programozás története során rengeteg programozási nyelv jelent meg, amelyek hátterében különböző programozási paradigmák állnak. A programozási paradigma elsősorban azt határozza meg, hogy a feladatokat, azaz az algoritmusokat milyen gondolatmenet alapján építjük fel, de meghatározza azt is, hogy a megoldási folyamat során, milyen eszközök állnak a rendelkezésünkre, azaz milyen utasításokból építhetjük fel a kódunk, az utasításokat milyen sorrendben végezhetjük el, milyen adatszerkezetek definiálására van lehetőségünk, stb.

A legismertebb programozási paradigmák az imperatív, az objektumorientált, a funkcionális és a logikai. Fontos azt is kiemelnünk, hogy vannak olyan programozási nyelvek, amelyek nem csak egy, hanem különböző programozási paradigmában való gondolkodást, kódolást is megengednek. Ilyenek a jól ismert C++, C#, Java, Python, Scala, F# stb. nyelvek. Jelen jegyzet elsősorban a tiszta funkcionális programozási paradigmán alapuló gondolkodást fogja bemutatni, ahol a Haskell programozási nyelv elsajátítása lesz a fő cél.

Az imperatív programnyelvek alap vezérlési szerkezete a ciklus utasítás, ahol a műveletsorok parancsok, utasítások egymás után való fűzését jelentik, amelyek tetszőleges módon megváltoztathatják a tárolt adatok (változók) értékét. Ezeknél a nyelveknél a lényeg, hogy milyen sorrendbe adjuk meg az utasításokat, hogyan dolgozzuk fel a bemeneti adatok értékét.

A funkcionális programnyelvek alapeszköze a kifejezések, a függvények kiértékelése. A függvények megadása legtöbb esetben rekurzívan történik, és sok esetben rejtve marad a háttérben történő kiértékelési módszer lényege. Ezekben a nyelvekben nincs változó, pontosabban nincs értékmódosító

művelet. A programkódok tömörek, átláthatók, helyességük akár matematikai eszközök segítségével is ellenőrizhető. A cél ezekben a nyelvekben, hogy tömör, logikailag könnyen követhető, helyesen felépített algoritmusokat adjunk meg.

Az imperatív stílus gyakorlati szemléltetése végett a továbbiakban megadjuk a faktoriális függvény C programozási nyelvben megírt kódját, ciklus vezérlő szerkezettel, majd rekurzív függvénnyel.

Az alábbi függvény a `for` ciklus szerkezettel határozza meg `n` faktoriálisát, azaz `n!`-t, ahol az eredményt a `res` változóban tároljuk, ami egyben a függvény visszatérési értékéül is szolgál. A kódsor elején a két `if` utasítás a triviális eseteket tárgyalja, a `for` keretén belül pedig a `res` változó értékének a változtatásával azt érjük el, hogy a műveletsor végén a `res` a kívánt eredményt tárolja.

```
1. változat:
int fakt1 (int n) {
   int i, res;
   if (n < 0) return -1;
   if (n == 0) return 1;
   for (i = 1, res = 1; i <= n; i++)
       res *= i;
   return res;
}
```

A következőkben a rekurzív függvénydefinícióknak két változatát is megadjuk, amivel a funkcionális nyelvekben meglévő lehetőségeket szeretnék előrevetíteni, illetve azzal, hogy a feladatot rekurzív függvénnyel oldjuk meg, közelebb kerülünk a funkcionális paradigma szerinti programozási stílushoz. Kezdő programozók a következő kódsorokat könnyebben fogják megérteni, mert azon túl, hogy tömörebb, közelebb is áll a középiskolában elsajátított matematikai gondolkodáshoz.

```
2. változat:
int fakt2 (int n) {
   if (n < 0) return -1;
   if (n == 0 ) return 1;
   else return n *  fakt2 (n-1);
}

3. változat:
 int fakt3 (int res, int n) {
```

```
    if (n < 0) return -1;
    if (n == 0) return res;
    return fakt3 (n * res, n-1);
}
```

A `fakt2` függvény, a matematikából is jól ismert rekurzív definíció alapján van megírva. A végső eredményt a függvény akkor tudja kiszámolni, amikor minden rekurzív függvényhívás visszatérési értékét már meghatározta. A `fakt3` szerkezete a `fakt2` függvényhez képest bonyolultabb, hiszen az eredmény tárolásához bevezet egy új paramétert, a `res`-t. A kívánt eredmény meghatározását pedig úgy oldja meg, hogy amikor önmagát hívja, akkor az első paramétere a módosított `res` értéke lesz, éppen ezért a rekurzív hívás legalsó szintjén az eredmény a `res` paraméterben már ki lesz számolva, így az `if (n == 0)` feltétel teljesülésekor ezzel az értékkel tér vissza a függvény.

A két függvény a rekurzív függvénydefiníciók két fontos példáját szemléltetik. A `fakt2` függvény az eredmény meghatározásához akkor számol, amikor *jön vissza* a rekurzióból, a `fakt3` pedig pont fordítva, amikor *megy be* a rekurzióba.

A függvényhívások, a három megadott függvénytörzs esetében, `10!` meghatározásához a következők lesznek, ahol az eredmény az `x` változóba kerül:

```
x = fakt1 (10);
x = fakt2 (10);
x = fakt3 (1, 10);
```

A faktoriális függvény Haskell-nyelvben megírt következő változatai, a fenti rekurzív definíciók után azt a célt szolgálják, hogy össze tudjuk hasonlítani az imperatív és funkcionális programozási stílus alapelemeit.

```
1. változat:
fakt1 :: Int -> Int
fakt1 0 = 1
fakt1 n = n * fakt1 (n-1)

2. változat:
fakt2 :: Int -> Int
fakt2 n
    | n < 0 = -1
    | n == 0 = 1
    | otherwise = n * fakt2 (n-1)
```

```
3. változat:
fakt3 :: Int -> Int -> Int
fakt3 res n
   | n < 0 = -1
   | n == 0 = res
   | otherwise = fakt3 (n * res) (n - 1)
```

A függvényeket úgy tudjuk kiértékelni, ha elindítjuk a Haskell interpretert, majd a `Prelude>` prompt után megadjuk rendre a függvényneveket és a bemeneti paraméter vagy paraméterek értékeit. A paramétereket ne tegyük zárójelbe, ahogyan azt más programozási nyelveknél esetleg megszoktuk, mert a Haskellben a zárójeleknek más szerepük van, például ha `fakt3` függvény hívásakor zárójelbe tesszük a paramétereket, futási hibát kapunk. A függvénynév és a paraméterek közé szóközt kell tenni. Ilyen módon a hívások, illetve az eredmények a következők lesznek:

```
Prelude> fakt1 10
3628800

Prelude> fakt2 10
3628800

Prelude> fakt3 1 10
3628800
```

Szerkezeti szempontból vegyük észre, hogy a függvénytörzsek előtti sorban megadtuk a függvények szignatúráját, azaz a típusdeklarációt, amelynek segítségével explicit módon lehet jelezni a függvény bemeneti paramétereinek és kimenetének típusát, illetve a bemeneti paraméterek számát. A paramétereket a `->` szimbólummal kell elválasztani, ahol az utolsó `->` után a függvény kimenetének típusát kell megadni. A függvénytörzsben a műveletsorok tördelve, tabulátort használva vannak megadva, amelyeket tartsunk be, mert ez is része a Haskell szabályrendszerének.

Működés szempontjából a megadott függvények hasonlók a C-ben megadott rekurzív kódokhoz, itt azonban mindhárom változat rekurzív. Az első két függvény esetében a tulajdonképpeni számítások akkor kerülnek elvégzésre, amikor *jövünk vissza a rekurzióból*, a harmadik esetben pedig amikor *megyünk be a rekurzióba*. A három változat iskolapéldája a Haskellben alkalmazható technikai eszközöknek. Az első változatban a mintaillesztés technikáját alkalmaztuk, ahol a többi megoldással ellentétben nem kezeltük le a negatív bemenetet, mert ez nem oldható meg mintaillesztéssel. A másodiknál és a harmadiknál feltételek segítségével különítettük el a lehetséges

eseteket. A `fakt3` működése algoritmikailag különbözik az előző kettőtől, ugyanúgy, ahogyan a C változatokban láttuk, itt is egy új paraméter, a `res` bevezetésével oldottuk meg az eredmény meghatározását.

Első körben megállapítható, hogy algoritmikailag két különböző módszer lehetséges: a számításokat akkor végezzük, amikor jövünk vissza a rekurzióból, illetve amikor megyünk be a rekurzióba. A faktoriális feladat esetében hatékonyság szempontjából nincs különbség a két módszer között, azonban fogjuk látni, hogy számos feladat esetében nem mindegy, hogy melyik módszert alkalmazzuk.

A Haskell számos beépített függvénnyel is rendelkezik, amelyek használata egyszerű, és tömör kódok megírását teszik lehetővé, ezek alkalmazásakor azonban figyeljünk arra, hogy ne rontsuk el a kódunk hatékonyságát.

A következő három függvény mindegyike az `n` faktoriális értékét határozza meg, a `foldr`, `foldl`, illetve `product` könyvtárfüggvényekkel. Ezeknek működéséről a későbbi fejezetekben bővebben is szó lesz, egyelőre annyit róluk, hogy ezekben az esetekben rejtett rekurzióról beszélünk, azaz nem jelenik meg explicit módon a rekurzió.

```
fakt4 :: Int -> Int
fakt4 n = foldr (*) 1 [1..n]

fakt5 :: Int -> Int
fakt5 n = foldl (*) 1 [1..n]

fakt6 :: Int -> Int
fakt6 n = product [1..n]
```

Meghívásuk hasonló módon történik, mint ahogyan azt korábban tettük:

```
Prelude> fakt4 10
3628800

Prelude> fakt5 10
3628800

Prelude> fakt6 10
3628800
```

3. fejezet

Alapfogalmak

3.1. Az első lépések

A funkcionális paradigmán alapuló gondolkodás, programozás egyidős a számítástechnika történetével. Elméleti hátterét a lambda kalkulus szolgáltatja, amely pontos leírást ad a függvények definiálására, kiértékelésére. Ennek az elméletnek a kidolgozója Alonzo Church volt, aki munkáját 1930-ban publikálta, majd ennek a mai napig is helytálló formáját 1936-ban jelentette meg[3]. Az első funkcionális programozási elemeket tartalmazó nyelv a Lisp volt, amelynek fejlesztését 1950-ben a massachusettsi Műszaki Egyetemen kezdték el. 1987-ben egy nemzetközi bizottság más működési elven alapuló, funkcionális programozási nyelv fejlesztése mellett döntött, és Haskell Curry amerikai matematikus neve után a programozási nyelvnek a Haskell nevet adták. Az első igazán megbízható Haskell-verzió azonban csak 2003-ban jelent meg.

A Haskell programozási nyelvnek több implementációja is ismert, az egyik a Hugs, amely egy interpreter, és amelyet leginkább oktatásban használnak. A másik fontos implementáció a GHC (Glasgow Haskell Compiler), amit valós alkalmazások fejlesztéséhez használnak. Ez az implementáció natív kódra fordít, és a fordítást követően a program futtatható lesz a GHC környezetétől függetlenül is. Biztosítja a párhuzamos végrehajtást, a debugolást (azaz a hibakeresést), a hatékonyság elemzését. Jelen jegyzet keretén belül ezzel az implementációval fogunk dolgozni.

A GHC komponensei a következők:

- a *ghc*, a tulajdonképpeni fordító, ami a natív kódot generálja,
- a *ghci*, az interaktív interpreter és debugger,

– a *runghc* a Haskell programokat fordítás nélkül futtató komponens.

A Haskell Platform a programozáshoz szükséges eszközöket tartalmazó csomag, a *https://www.haskell.org/platform/* oldalról tölthető le, amelynek Windows alatti egyszerűbb telepítése végett a fejlesztők előbb a Chocolatey telepítését ajánlják: *https://chocolatey.org/install*. Mindkettő telepítését a `PowerShell`-en keresztül, `admin`-ként kell végezni. A Haskell telepítésekor automatikusan felkerül a számítógépünkre a `cabal` eszközkezelő, amelynek segítségével az alapcsomagok mellett további Haskell-könyvtárak telepíthetők, illetve Haskell-projektek menedzselhetők. Megjegyezzük továbbá, hogy a bemutatásra kerülő Haskell-kódok kipróbálásához legalább a `8.10.4`-es verziót használjunk.

A Haskell és Chocolatey telepítése után szükség lesz még egy kódszerkesztőre, amire érdemes a Visual Studio Code-ot használni: *https://code.visualstudio.com/download*. A Visual Studio Code telepítése után egy terminál elindításával, majd a **`ghci`** parancs kiadásával lehet végül a Haskellt is elindítani.

A Haskell sikeres indításakor a terminál ablakban megjelenik a prompt: **`Prelude>`**, amely többek között azt jelzi, hogy a standard könyvtár is sikeresen betöltődött. A prompt után parancsokat, kifejezéseket írhatunk, amelyeket a Haskell interpretere azonnal megpróbál végrehajtani, kiértékelni. `GHC` parancsokkal állományok betöltését is megvalósíthatjuk, amely után az állományokban található függvényeket kiértékelhetjük.

3.2. GHC-parancsok

A prompt után számos parancs írható, amelyeket rövidített formában is használhatunk, a parancsnév kezdőbetűjének a megadásával. A továbbiakban felsoroljuk a leggyakrabban használtakat:

– **`:load fnev.hs`**, vagy **`:l fnev.hs`**, az `fnev.hs` nevű állomány betöltése, fordítása,
– **`:reload`**, vagy **`:r`** az aktuálisan betöltött állományok újratöltése, újrafordítása,
– **`:type kif`**, vagy **`:t kif`** a `kif` kifejezés típusának a lekérdezése,
– **`:?`** az összes GHC parancs lekérdezése,
– **`:quit`** kilépés a GHC-ből,
– **`:cd C:\Diak`** az aktuális mappa kiválasztása, jelen parancs esetében a `C:\ Diak` lesz.

A következő parancsokkal beállításokat adhatunk meg:

- **`:set +t`** egy kifejezés kiértékelése után megjelenik a kifejezés típusa,
- **`:unset +t`** az előző beállítás visszavonása,
- **`:set +s`** egy kifejezés kiértékelése után megjelenik a kiértékeléshez szükséges idő és a felhasznált bájtok száma,
- **`:unset +s`** az előző beállítás visszavonása,
- **`:set +m`** egy hosszú kifejezés több sorba való tördelésének lehetővé tétele,
- **`:unset +m`** az előző beállítás visszavonása,
- ...

3.3. Alaptípusok

A Haskellben, hasonlóan más programozási nyelvekhez, több alaptípus is használható.

Az **`Int`** rögzített méretű egész számok kezelésére alkalmas. Az `Int` típusú számok terjedelme gépfüggő, ami azt jelenti, hogy egy `32` bites számítógépen az `Int` mérete `32` bit, míg a `64` bites számítógépen `64` bit.

Az **`Integer`** tetszőleges méretű egész számok kezelésére alkalmas. Az `Int` és `Integer` közötti választás elsősorban a hatékonyság és a szükségesség közötti mérlegelést jelenti.

A **`Float`** és **`Double`** típusok valós számok kezelésére alkalmasak. A `Double` 64 bites lebegőpontos ábrázolást jelent, aminek inkább ajánlott a használata, ellentétben a `Float` típussal.

A **`Bool`** egy logikai típusú adat kezelését teszi lehetővé, kétfajta értéket vehet fel: **`False, True`**.

A **`Char`** egyetlen (Unicode) karakter eltárolására alkalmas, ahol aposztróf közé kell írni az értéket:

```
'A', '\n', '+', 'ő'.
```

A **`[Char]`** típus karakterláncok kezelésére alkalmas, de a **`String`** típusnév is karakterláncot jelent, ahol idézőjel között lehet megadni az értékeket.

```
"Hello Sapientia", "10 * 0 = 0".
```

3.4. A Haskell mint számológép

A Haskell számológépként is használható, mert mint említettük, a prompt után beírt kifejezés rögtön kiértékelődik. Az alábbi példákban azt

mutatjuk be, hogyan használjuk az aritmetikai operátorokat, függvényeket, azaz hogyan adjunk össze két vagy több számot, hogyan szorozzunk, hogyan járjunk el, ha meg akarjuk határozni két szám osztási egészrészét, osztási maradékát.

Egy adott operátor, függvény többféle formában is meghívható: infix és prefix formában egyaránt. Az infix forma azt jelenti, hogy a műveleti jelet, a függvény nevét az operandusok közé írjuk, míg prefix formában a műveleti jel, a függvénynév megelőzi a két operandust. Haskellben az operátorok is függvények, ahol a különbség a választott névben mutatkozik meg, az operátorok nevében ugyanis nem szerepelnek angol ábécébeli betűk. Operátorok esetében, a prefix forma használatakor, az operátor nevét zárójelbe kell tenni. Függvények esetében, infix forma használatakor pedig a függvény nevét ` ` jelek közé kell tenni.

```
Prelude> 10 + 3
13

Prelude> (+) 3 + 10
13

Prelude> 7.5 * 4.3
32.25

Prelude> (*) 7.5 4.3
32.25
```

```
Prelude> div 10 3
3

Prelude> 10 `div` 3
3

Prelude> mod 13 7
6

Prelude> 13 `mod` 7
6
```

A hatványozáshoz, aszerint, hogy egész, vagy valós számokon végezzük, más-más műveleti jelet használunk, ily módon a négyzetgyök, a köbgyök stb. értékét könyvtárfüggvény nélkül is meg tudjuk határozni:

```
Prelude> 10 ** 3
1000.0

Prelude> 10 ^ 3
1000
```

```
Prelude> 2 ** 0.5
1.4142135623730951

Prelude> 2 ** (1/3)
1.2599210498948732
```

Az **Integer** típus bevezetésével a Haskell képes tetszőlegesen nagy számokat kezelni, például a hatványozó operátor, anélkül, hogy bármiféle könyvtárcsomagot importálnánk, helyesen határozza meg a következő számítások eredményét:

```
Prelude> 2 ^ 100
1267650600228229401496703205376

Prelude> 2 ** 100
1.2676506002282294e30
```

Ha a kifejezések megadása során hibát követünk el, akkor azokat a Haskell nem fogja kiértékelni, helyette hibaüzenetet ad. Például a következő kifejezést a Haskell nem tudja kiértékelni, mert a `^` operátor csak egész típusú értékekre alkalmazható:

```
Prelude>  2 ^ 0.5
... error:
Could not deduce (Integral b0) arising from a use of '^'
```

A következőkben a `Prelude`-ot el fogjuk hagyni a promptra való hivatkozásból, ilyen formában tehát ha egy lekérdezést, függvényhívást szeretnénk bemutatni, egyszerűen csak a `>` jelet fogjuk használni. A jegyzet további részében azt az elvet fogjuk követni, hogy megadjuk a lekérdezések eredményeit is.

Egész, illetve valós típusú adatok kezelése mellett könnyedén karakterlánc típusú adatokkal is tudunk műveleteket végezni. A **`++`** operátor segítségével akár több karakterláncot is egymás után tudunk fűzni, az **`==`** operátorral pedig megvizsgálhatjuk, hogy két karakterlánc egyforma-e:

```
 > "Hello " ++ "szamitastechnika" ++ "!"
"Hello szamitastechnika!"

> "Hello " ++ "mat-info!" == "Hello mat-info!"
True
```

A Haskell prompt után függvények definiálására is van lehetőségünk. Egy Haskell-függvénytörzs a függvény nevéből, a név után írt bemeneti paraméterekből, az egyenlőségjelből és az egyenlőség jobb oldalára írt kifejezésekből áll.

A következő **`teruletK`** nevű függvény egy kör területét számolja ki az `r` paraméterként megadott sugár értéke alapján. Az egyenlőség jobb oldalán egy `if` kifejezés áll. Hosszú függvénytörzsek esetében érdemes beállítani a sortörést a `:set +m GHC` paranccsal:

```
> :set +m
> teruletK r = if r < 0 then error "negativ a bemenet!"
| else r * r * pi
```

```
> teruletK 5
78.53981633974483

> teruletK -10
... error:
Non type-variable argument in the constraint...

> teruletK (-10)
*** Exception: negativ a bemenet! ...
```

A fenti lekérdezésekből jól látható, hogy ha negatív számot akarunk bemenetként megadni, akkor azt zárójelbe kell tenni, ellenkező esetben futási hibába ütközünk. A Haskell megköveteli ugyanis az összetett kifejezések zárójelezését, egy negatív szám pedig összetett kifejezésnek számít. A kódsorban használt `error` kivételek kezelésére, hibaüzenet megadására alkalmas.

3.5. Az első Haskell-állomány

A prompt után írt kifejezéseink, függvényeink elvesznek, ha elhagyjuk a Haskell környezetét, habár az utolsóként beírt kifejezések a ↑, ↓ gombok lenyomásával újra betölthetőek. Egy sokkal kényelmesebb megoldás azonban az, ha programkódjainkat egy szövegállományba írjuk, és egy olyan szerkesztőt használunk, ami lehetőséget ad ezek lementésére, megnyitására, módosítására, rendszerezésére. Egy kényelmes megoldás, amit korábban is említettünk, a Visual Studio Code használata.

Legyen a továbbiakban **`Elso.hs`** az első Haskell-függvényt tartalmazó állomány neve, amelybe írjuk be a korábbi `teruletK` területszámoló függvényt. A Haskell-függvényeket, -kifejezéseket tartalmazó szövegállományok kiterjesztése ugyanis **`hs`**.

A függvénytörzs előtti sorban megadjuk a függvény szignatúráját, azaz a típusdeklarációját, amely jelen esetben azt jelzi, hogy a `teruletK` függvénynek egy bemeneti paramétere van, aminek típusa `Double`, azaz valós, és kimenete is `Double` típusú.

Az állomány tartalma a következő lesz, ahol a kódsor begépelésekor vigyázzunk a tördelésre, mert annak szintaktikai szerepe van!

```
teruletK :: Double -> Double
teruletK r =
   if r < 0 then error "negativ a bemenet!"
   else r * r * pi
```

A Haskell elindítása után válasszuk ki azt a mappát, amelybe az állományt mentettük. Feltételezve, hogy ez a `C:\Users\Documents\Haskell`, akkor járjunk el a következőképpen:

```
> :cd C:\Users\Documents\Haskell
```

Ezt követően írjuk be a prompt után a következőket:

```
> :l "Elso.hs"
```

Ekkor a Haskell elemezni fogja a kódsort, és ha nem talál benne szintaktikai hibát, azaz a beírt kódsorok megfelelnek a szabályrendszerének, akkor lefordítja. Ekkor lesz kiértékelhető a függvényünk. Ha egy konstans bemenetre szeretnénk kiértékelni a függvényünket, akkor a következőképpen járjunk el, ahol a meghívás utáni sorban feltüntettük az eredményt is:

```
> teruletK 10
314.1592653589793
```

Az `Elso.hs` állomány további függvényekkel egészíthető ki. Az állomány módosítása után, ha azt szeretnénk, hogy a Haskell értelmezni tudja az újonnan hozzáadott függvényeket, akkor mindig szükség van egy mentésre és egy új fordításra. Az új fordítást végezhetjük a `> :r` paranccsal is, ha nincs mappa-, illetve állománynév-változtatás.

Futtatható állomány létrehozása érdekében tegyük hozzá az állományhoz a következő sorokat:

```
main = do
   print (teruletK 10)
   print "Press any key to continue ..."
   getLine
   return ()
```

A futtatható állomány, az `Elso.exe`, a

```
> :! ghc --make "Elso.hs"
```

parancs megadásával jön létre. A futtatható állománynak nevet is választhatunk, a

```
> :! ghc --make "Elso.hs" -o ElsoExe.exe
```

parancs segítségével. Ne felejtsük megkeresni az aktuális mappában, majd futtatni a létrehozott állományt! A `main` függvényben használt `print, getLine` függvények hasonló szerepet töltenek be, mint más programozási nyelvben, róluk és a `return`-ről is későbbi fejezetekben lesz szó.

3.6. Megjegyzések

A Haskell-állományok tartalmazhatnak, egy- vagy többsoros megjegyzéseket, amelyeket a Haskell a fordítás során figyelmen kívül hagy. A megjegyzéseket erre a célra fenntartott szimbólumok után, illetve közé kell írni. Szerepük, hogy a programozó rövid magyarázatokkal láthassa el saját programkódját.

A következő két példa egy egysoros és egy többsoros megjegyzést mutat:

```
-- ez egy egysoros megjegyzés

{-
ez egy többsoros
megjegyzés
-}
```

3.7. A lista típus

A lista adatszerkezet az alaptípusok mellett a legalapvetőbb összetett adattároló. Egy listában az elemek száma változó, de csak ugyanolyan típusú értékeket tárolhatunk bennük. Jelölésükre szögletes zárójelet használunk.

A következő példákban az `ls1` egész számok listáját, az `ls2` karakterek listáját, míg az `ls3` valós számok listájának listáját jelöli.

```
> ls1 = [1,2,3,4]
> ls2 = ['a'..'z']
> ls3 = [[7.5,8.25], [6.33,7.75,9.5], [10,9.5, 8.75,6.3]]
```

A fenti példákban az `ls1` típusa `[Int]`, az `ls2` típusa `[Char]`, míg az `ls3` típusa: `[[Double]]`.

Figyeljük meg, hogy a két egymás után használt pont `..` segítségével, listákat tudunk generálni, anélkül, hogy egyenként megadnánk a listaelemeket. Lépésközt is meg lehet adni, amely az alapján lesz kiszámolva, hogy mennyi a különbség az elsőnek és másodiknak megadott elem között:

```
> ls4 = [0,5..40]
> ls4
[0, 5, 10, 15, 20, 25, 30, 35, 40]

> ls5 = [-3, -6.. -20]
```

```
> ls5
[-3, -6, -9, -12, -15, -18]
```

Számos könyvtárfüggvény létezik a listák feldolgozására. A következő példákban a `length` egy lista elemszámát állapítja meg, a `reverse` megfordítja a listát, a `maximum` pedig egy lista legnagyobb elemét adja meg:

```
> length [3.66, 2.5, 10.33, 7.25, 5.75]
5
> reverse "Sapientia"
"aitneipaS"
> maximum "erdelyi magyar tudomanyegyetem"
'y'
```

Egy karaktereket tartalmazó lista esetében az értékeket megadhatjuk egyenként is, de természetesen kényelmesebbek a fenti kifejezések:

```
> reverse ['e','g','y','e','t','e','m']
"meteyge"
```

Egy Haskell-függvényre, ha különböző típusú adatokra is alkalmazható, azt mondjuk, hogy polimorf. A következő lekérdezésekben, a `length` karaktereket tároló lista esetében adja meg az elemek számát, a `reverse`-t egész elemtípusú listára hívjuk, a `maximum` pedig valós számokat tároló listában keresi meg a legnagyobb elemet. Mindhárom függvény tehát polimorf.

```
> length "Marosvasarhely"
14
> reverse [1,2,3,4]
[4,3,2,1]
> maximum [12, 56, 7.8, 23, 11.9]
56.0
```

Elöljáróban még bemutatjuk a `head`, a `tail`, a `null`, az `init` és a `last` függvények használatát. A `head` megadja a lista első elemét, a `tail` levágja a lista első elemét, a `null` megvizsgálja, hogy üres-e a lista, az `init` levágja a lista utolsó elemét, míg a `last` meghatározza a lista utolsó elemét.

```
> head "Keleti-Karpatok"
'K'

> tail ["Kelemen", "Gyergyoi", "Hargita", "Csalho"]
["Gyergyoi", "Hargita", "Csalho"]
```

```
> null []
True

> init ["Kelemen", "Gyergyoi", "Hargita", "Csalho"]
["Kelemen","Gyergyoi","Hargita"]

> last "Nagy-Hagymas"
's'

> last ["Kelemen", "Gyergyoi", "Hargita", "Csalho"]
"Csalho"
```

3.8. A tuple típus

A lista típus mellett legalább olyan fontos és gyakran használt a `tuple` típus. Egy `tuple` típusú adat különböző típusú elemek kombinációját jelöli, ahol az elemek száma rögzített. A magyar terminológiában a `tuple` típust rendezett *n-esnek* mondjuk. Jelölésükre kerek zárójelet használunk.

A következő példában a `t1` egy kételemű, tuple típusú adatot jelöl:

```
> t1 = ("Pietrosz", 2102)
```

Egy kételemű tuple esetében a leggyakrabban használt könyvtárfüggvények az `fst` és az `snd`, ahol az `fst` (first) a tuple első elemét, míg az `snd` (second) a tuple második elemét adja meg:

```
> fst t1
"Pietrosz"
> snd t1
2102
```

Három- vagy többelemű tuple-ök esetében nem működnek ezek a függvények, de könnyedén megírhatók, ahogy azt a következő sorokban láthatjuk:

```
> myFst (t1, t2, t3) = t1
> myFst ("Mari", 1990, 8.50)
"Mari"

> mySnd (t1, t2, t3) = t2
> mySnd ("Mari", 1990, 8.50)
```

```
1990

> myThd (t1, t2, t3) = t3
> myThd ("Mari", 1990, 8.50)
8.5
```

3.9. Könyvtármodulok

A `Prelude` függvényei mellett számos könyvtárcsomag függvényeit használhatjuk, ha megtörténik a könyvtárcsomag importálása, amelyet az `import` kulcsszó és a könyvtármodul nevének a használatával valósíthatunk meg.

A következő példákban a `Data.Char` könyvtármodul néhány függvényének a használatát mutatjuk be. Az `isDigit` `True` vagy `False` értéket ad, aszerint, hogy a bemeneti paramétere számjegy vagy sem. Az `isAlpha` is egy logikai tesztet végez, és `True` vagy `False` értéket ad, aszerint, hogy a bemeneti karakter ábécébeli betű vagy sem.

```
> import Data.Char
> isDigit '3'                    > isAlpha 'a'
True                             True
> isDigit 'w'                    > isAlpha '?'
False                            False
```

Meg is írhatjuk a saját `isDigit` függvényünket, ahol a függvénytörzs egyetlen logikai kifejezésből fog állni, amelyben a más programozási nyelvekben is használt relációs operátorokat `>=`, `<=`, és az `&&` logikai operátort fogjuk használni, ahol az utóbbi az **és** logikai kapcsolatot jelöli. Itt jegyezzük meg, hogy a Haskellben a logikai **vagy** kapcsolatot a `||` jelöli.

```
> myIsDigit x = x >= '0' && x <= '9'
> myIsDigit '3'
True
```

A függvényünk típusdeklarációja a korábban ismertetett `:t` GHC-paranccsal kérhető le, amely jól mutatja, hogy a függvénynek egy bemeneti, `Char` típusú paramétere van, kimenetének típusa pedig `Bool`.

```
> :t myIsDigit
myIsDigit :: Char -> Bool
```

A `Data.List` könyvtárcsomagban a listák kezelését, feldolgozását segítő függvények találhatók, ezek közül most az `inits`, a `tails`, a `nub` és a `sort` függvények használatára adunk példát. Figyeljük meg, hogy ezek is polimorf függvények, illetve kérdezzük le a típusdeklarációkat, hogy megtudhassuk, hogy a függvények paraméterei, illetve a kimenetek milyen típusúak.

Az `inits` a bemeneti listából előállítja a *prefixeket*, azaz a kezdőszeleteket, a `tails` a *postfixeket*, azaz a listavégeket, a `nub` törli egy lista többször előforduló elemeit, a `sort` pedig a bemeneti elemeket rendezi növekvő sorrendbe.

```
> import Data.List
> inits "koros"
["","k","ko","kor","koro","koros"]

> tails "koros"
["koros","oros","ros","os","s",""]

> nub "erdelyi-szigethegyseg"
"erdlyi-szgth"

> nub [1, 4, 2, 1, 4, 5, 3]
[1,4,2,5,3]

> sort [1, 6, 5, -10, 7]
[-10,1,5,6,7]
```

A `Data.Ratio` könyvtárcsomag a **`Rational`** típus használatát biztosítja. Ennek a típusnak a segítségével a valós számokon végzett műveletek során adódó kerekítési hibák küszöbölhetőek ki. A `Rational` típus megengedi a számláló és nevező külön egységként való kezelését, ahol mindkét érték típusa `Integer` lesz. Egy `Rational` típusú érték létrehozásához meg kell adni a számlálót, a nevezőt, és a két érték közé **`%`** jelet kell tenni.

```
> import Data.Ratio
> 1 % 5 + 2 % 3
13 % 15
```

A `Data.Complex` könyvtármodul a komplex számok kezeléséhez szükséges függvényeket tartalmazza, ahol a komplex számok valós, illetve imaginárius részei a **`:+`** szimbólum használatával adhatók meg, a `:+` előtt a valós rész, utána az imaginárius rész található. A `sum` a szögletes zárójelek között megadott számok összegét határozza meg. Ez is egy polimorf függvény, alkalmazható egész, komplex stb. számok összeadására.

```
> sum [3, 2, 10, 7, 5]
27

> import Data.Complex
> sum [3 :+ (-2.6), 11 :+ 3.4, 1 :+ (-2.41)]
15.0 :+ (-1.6100000000000003)
```

3.10. Lokális definíciók

Egy adott függvénytörzsben lehetőség van függvények, lokális kifejezések, azonosítók definiálására, nem is egy-, hanem kétféleképpen is. A `where`, illetve a `let...in` kifejezést használhatjuk erre a célra.

A következő `tupleMax` függvény meghatározza a paraméterként megadott háromelemű tuple típusú adatok közül azt, amelyiknek a harmadik eleme a nagyobb. A `where` kulcsszó használatával és a megfelelő tördeléssel a függvénytörzs keretén belül három lokális kifejezést adunk meg. Az első két kifejezés segítségével egy-egy háromelemű tuple elemeit nevezzük el, lehetőséget teremtve arra, hogy külön-külön is hivatkozhassunk rájuk. A harmadik kifejezésben pedig az `m` a `max` beépített függvény kimeneti értékét fogja jelölni.

```
tupleMax :: (String, Int, Double) -> (String, Int, Double)
            -> (String, Int, Double)
tupleMax t1 t2 = if m == x3 then t1 else t2
   where
   (x1, x2, x3) = t1
   (y1, y2, y3) = t2
   m = max x3 y3

> tupleMax ("Mari", 1990, 8.50) ("Feri", 1991, 9.25)
("Feri",1991,9.25)
```

A tuple harmadik elemének a lekérésére nem használtuk a korábban megírt `myThd` függvényt, de nyugodtan megtehettük volna.

Megjegyezzük, hogy az alkalmazott tördelés, azaz a margózás meghatározza a kifejezések láthatósági tartományát, ami azt jelenti, hogy a függvénytörzsön kívül az `x1, x2, x3, y1, y2, y3`, illetve `m` azonosítók már nem használhatók.

A következő `tupleMin` függvényben minimumot keresünk egy háromelemű tuple típusú adat második eleme szerint. Egy `let...in` kifejezés

keretén belül a minimum elemet a `min` függvény meghívásával határozzuk meg, a tuple második elemének kiválasztása pedig a korábban megírt `mySnd` függvénnyel történik.

```
tupleMin :: (String, Int, Double) -> (String, Int, Double)
            -> (String, Int, Double)
tupleMin t1 t2 =
   let
       m = min x y
       x = mySnd t1
       y = mySnd t2
   in
       if m == x then t1 else t2

> tupleMin ("Mari", 1990, 8.50) ("Feri", 1991, 9.25)
("Mari",1990,8.5)
```

Vegyük észre, hogy a `let...in` kifejezésekben megadott jelölések sorrendje más, mint a `tupleMax` függvénynél, itt azelőtt használjuk az `x` és `y` azonosítókat, mielőtt ők valamilyen értéket jelölnének. Megállapíthatjuk, hogy nem számít a kifejezések megadásának a sorrendje, a lényeg, hogy a blokk keretén belül minden azonosító jelöljön egy értéket vagy kifejezést.

3.1. feladat Definiáljunk egy `Pont` típusú értéket, és írjunk három Haskell-függvényt. A `kezdoP` függvény egy kezdeti értékadást végezzen. A `mozgat` függvény mozgassa el a pontot a paraméterként megadott pontba. A `tavolsag` függvény pedig határozza meg két pont között a távolságot.

A típusok átláthatósága végett saját típust is definiálunk: a `Szin` és a `Pont` típusok a `type` kulcsszóval kerülnek megadásra. Itt jegyezzük meg, hogy a Haskell különbséget tesz a kis- és nagybetűk között, és megköveteli, hogy a függvények, az azonosítók nevei mindig kisbetűvel kezdődjenek, a típusnevek azonban kötelező módon nagybetűvel kell hogy kezdődjenek.

```
type Szin = String
type Pont = (Double, Double, Szin)

kezdoP :: Szin -> Pont
kezdoP szin = (0, 0, szin)

mozgat :: Pont -> Double -> Double -> Pont
mozgat (x, y, szin) xTav yTav
   = (x + xTav, y + yTav, szin)
```

```
tavolsag :: Pont -> Pont -> Double
tavolsag (x1, y1, szin1) (x2, y2, szin2)
   = sqrt (dx * dx + dy * dy)
   where
   dx = x2 - x1
   dy = y2 - y1

> p1 = kezdoP "fekete"
> p2 = mozgat p1 10 15
> tavolsag p1 p2
18.027756377319946
```

A függvénytörzsben használt **sqrt** a más programozási nyelvekben is használt négyzetgyököt meghatározó könyvtárfüggvény. Használatához nem kell importálni semmilyen könyvtárcsomagot, mert benne van az alapértelmezetten betöltött Prelude modulban.

3.11. Típusosztályok

A Haskell azonosítók, függvényparaméterek egy megadott kifejezésben, függvényben nem csak egy adott típushoz tartozó értéket jelölhetnek. A Haskell bevezeti a **típusváltozó**, illetve **típusosztály** fogalmakat, amelyek segítségével lehetőség nyílik, hogy az azonosítók, a függvényparaméterek ugyanazon kifejezésben, függvényben a különböző kiértékelések, meghívások során más és más típusú értéket vegyenek fel.

Ha a parancssorban, a korábban használt `sort` függvény szignatúráját lekérdezzük, akkor a következő választ kapjuk:

```
> import Data.List
> :t sort
sort :: Ord a => [a] -> [a]
```

Ez azt jelenti, hogy a `sort` függvény bemeneti és kimeneti paraméterének típusa lista, ahol a lista elemei lehetnek akár **Int**, akár **String** típusúak is, egyetlen megszorítás vonatkozik rájuk, hogy az **Ord** típusosztályba tartozzanak, ami tulajdonképpen azt jelenti, hogy megköveteljük, hogy az elemek között rendezési reláció álljon fenn.

Ily módon egész számok rendezése mellett karakterláncokat, karakterláncokból álló adathalmazt is rendezhetünk, ugyanazzal a függvénnyel:

```
> sort "aranyos"
"aanorsy"

> sort ["sebes", "kukullo", "aranyos", "nyarad"]
["aranyos","kukullo","nyarad","sebes"]
```

A Haskellben használt típusok osztályokba vannak sorolva és a kifejezésekkel, függvényparaméterekkel végzett műveletek döntik el, hogy milyen típusosztályt vagy típusosztályokat kell megadni a függvény szignatúrájában. A típusváltozók jelölésére az angol ábécé kisbetűit használjuk, a típusosztályokat pedig a függvény szignatúrájában a `::`, és `=>` közé kell írni.

Az `init` függvény típusdeklarációjának lekérdezésekor azonban azt látjuk, hogy nincs specifikálva típusosztály, ez azt jelenti, hogy a függvénynek bármilyen elemtípusú listát megadhatunk bemeneti paraméterként, és a kimenet is tetszőleges elemtípusú lista lesz.

```
> :t init
init :: [a] -> [a]
```

Típusváltozók használatakor adott esetben tehát nem szükséges megszorítást megadni, azaz nem szükséges jelezni, hogy a típusváltozók milyen típusosztályhoz tartoznak. Abban az esetben kell ezt megtenni, ha a függvénytörzs keretén belül megadott kifejezések, operátorok használata ezt megköveteli.

A korábbi `teruletK` függvény esetében, a függvény szignatúrája típusváltozókat használva, a következő lesz:

```
teruletK :: (Ord a, Floating a) => a -> a
teruletK r =
    if r < 0 then error "negativ a bemenet!"
    else r * r * pi
```

Ezzel az új típusdeklarációval azt jelezzük a fordítónak, hogy a bemeneti és kimeneti paraméterek típusa **a**, azaz egyforma, az **Ord** és a **Floating** típusosztályok specifikációjával pedig két megszorítást is megadtunk a használható típusokra vonatkozóan.

Az ilyen típusú, polimorf függvény sokkal gyakoribb a Haskellben, mint más programozási nyelvben. Másfelől a függvények szignatúráját nem is szükséges megadni, mert a Haskell rendelkezik egy komoly típusellenőrző rendszerrel, ami még fordítási időben megtalálja a legáltalánosabb típusdeklarációt, hogyha az lehetséges.

3.2. feladat Írjunk egy Haskell-függvényt, amely meghatározza egy szám abszolút értékét.

```
abszolut x
    | x < 0 = -x
    | otherwise = x

> abszolut (-10)
10
```

Vegyük észre, hogy az `abszolut` függvény kiértékelhető, mert a fordító meghatározta az `x` paraméter, illetve a kimeneti érték típusát, annak ellenére, hogy mi nem adtuk meg ezeket. Ezt le is tudjuk ellenőrizni, a következőképpen:

```
> :t abszolut
abszolut :: (Ord a, Num a) => a -> a
```

Az eredményként megjelenő szignatúra azt jelzi, hogy a függvény paramétereire két megszorítást is alkalmazott a fordító: a függvény bemenete, illetve a kimenet típusa az `Ord` és `Num` típusosztályokhoz kell tartozzanak. A Haskell tehát meg tudja határozni a helyes függvényszignatúrát, függetlenül attól, hogy a programozó ezt megadta-e vagy sem, és a fordító által meghatározott szignatúra a lehető legáltalánosabb lesz. Ez a szignatúra teszi lehetővé, hogy az `abszolut` függvény különböző típusú bemenetre is kiértékelhető lesz, például valós számokra is:

```
> abszolut 5.65
5.65
```

Az `abszolut` függvény szignatúráját többféleképpen is meg lehet adni, például a következők közül bármelyik helyes fordítást eredményez, de vigyázzunk, egyszerre csak egyet adjunk meg:

```
abszolut :: Int -> Int
abszolut :: Integer -> Integer
abszolut :: Double -> Double
```

A fenti szignatúrák mindegyike azonban szűkíteni fogja a paraméterek értékhalmazát, mind a három esetben a bemeneti argumentumok, illetve a kimenet értéke korlátozva lesz. Az `abszolut :: Int -> Int` esetében, ha a következő bemenetre hívjuk a függvényt, futási hibát kapunk:

```
> abszolut 5.6
No instance for (Fractional Int)...
```

Ahhoz, hogy a legáltalánosabb szignatúrát tudjuk megadni, mindig azt figyeljük, hogy a paraméterekkel milyen műveleteket végzünk, milyen operátorok kerülnek alkalmazásra. A Haskell egy típusosztály definiálásakor megadja a típusosztályhoz tartozó operátorokat, így ezek alapján mindig el lehet dönteni egy paraméterről, hogy azt milyen típusosztályba soroljuk. A típusosztályok között egy jól meghatározott függőségi kapcsolat is létezik. A függőségi kapcsolat vagy származtatás az osztálydefiníciók alapján mindig egyértelmű, amelyet szintén számításba kell venni, amikor a függvények szignatúráját megadjuk.

A továbbiakban megadjuk azoknak a típusosztályoknak a definícióját, amelyeket gyakrabban fogunk használni, és példákat is adunk a típusosztályokban definiált operátorok használatára.

Az **Eq** típusosztályt olyan típusváltozók esetében kell használni, amikor az == (egyenlőség) és az /= (nem egyenlőség) operátorokkal végzünk műveleteket, definíciója a következő:

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    x /= y = not (x == y)
    x == y = not (x /= y)

> "Erdelyi-medence" /= "erdelyi-medence"
True

> 13.5 == 3.4
False
```

A `not` függvénynek `True` bemenetre `False`, `False` bemenetre `True` a visszatérési értéke, típusdeklarációja pedig a következő:

```
not :: Bool -> Bool
```

Az **Ord** típusosztály az `Eq` típusosztályból van származtatva, és olyan típusváltozók esetében használjuk, amikor az elemek között rendezettségi kapcsolat áll fenn.

```
class Eq a => Ord a where
    (<) :: a -> a -> Bool
    (<=) :: a -> a -> Bool
    (>) :: a -> a -> Bool
    (>=) :: a -> a -> Bool
    min :: a -> a -> a
```

```
max :: a -> a -> a
compare :: Ord a => a -> a -> Ordering
```

A típusosztály keretén belül definiált operátorok, függvények közül csak a `compare` függvényre térünk ki, mert a többi szerepe és használata egyértelmű.

A **`compare`** függvény a paraméterként megkapott két értéket hasonlítja össze, kimenete az `LT, GT` vagy `EQ` konstans lesz, aszerint, hogy az első érték a kisebb, az első érték a nagyobb, vagy a két érték megegyezik. A típusdeklarációból jól látszik, hogy a függvény bemeneti paraméterei az `Ord` típusosztályhoz tartoznak, a kimenet pedig `Ordering` típusú. Egy `Ordering` típusú adat háromfajta értéket vehet fel: `LT, GT, EQ`.

```
> compare 2 3
LT -- 2 Less Than 3

> compare "olt" "maros"
GT -- "olt" Greater Than "maros"

> compare [1, 2, 3] [1, 2, 3]
EQ -- [1, 2, 3] Equal to [1, 2, 3]
```

A **`Num`** típusosztályt akkor használjuk, amikor numerikus értékekkel dolgozunk, és a következőképpen van definiálva:

```
class Num a where
    (+), (-), (*) :: a -> a -> a
    negate :: a -> a
    abs, signum :: a -> a
    fromInteger :: Integer -> a
```

A `negate` megváltoztatja a bemenet előjelét, az `abs` meghatározza a bemenet abszolút értékét, míg a `signum` kimenete `(-1)`, ha a bemenet negatív, `1`, ha a bemenet pozitív szám, és `0`-t határoz meg, ha a bemenet `0`.

```
> negate (-7.8)
7.8
> negate 4
-4
```

```
> signum (-5.4)
-1.0
> signum 5
1
```

A `fromInteger` explicit típuskonverziót tesz lehetővé, például ha két `Integer` típusú értéken akarjuk az osztás műveletét alkalmazni, akkor típuskonverziót kell alkalmazni, mert ellenkező esetben fordítási hiba lép fel.

A következő `osztas` függvény fordításakor nem lesz hiba, míg az `osztas_` esetében fordítási hiba lép fel.

```
osztas :: Integer -> Integer -> Double
osztas x y = fromInteger x / fromInteger y

> osztas 758375832 21171189
35.82112615403887

osztas_ :: Integer -> Integer -> Double
osztas_ x y = x / y
... error:
Couldn't match expected type 'Double' with...
```

Figyeljük meg, hogy a `fromInteger` használatakor nem azt mondtuk meg, hogy mire, hanem azt, hogy miről, azaz milyen típusról történjen az átalakítás.

A **Real** típusosztály a `Num` és `Ord` típusosztályokból van származtatva, és egyetlenegy függvényt tartalmaz, a `toRational`-t, amely a numerikus és a `Rational` típusok közötti átalakításért felelős.

```
class (Num a, Ord a) => Real a  where
   toRational ::  a -> Rational

> toRational 0.5
1 % 2
```

Az **Integral** típusosztály a `Real`, illetve `Enum` típusosztályokból van származtatva, ahol az `Enum` típusosztályba a felsorolható típusok tartoznak. A következő kódsorok az `Enum`-ban definiált `pred` és `succ` függvények használatát mutatják be. A `Napok` egy új típus lesz, amelyre a `deriving` kulcsszó segítségével az `Enum` és `Show` (erről később lesz szó) típusosztályokhoz tartozó példányokat automatikusan származtatjuk. A `Napok` definiálásakor megadjuk a hét lehetséges konstans értéket, amelyet egy ilyen típusú adat majd felvehet.

```
> data Napok = Vas | Het | Ked | Sze | Csut | Pen | Szo
                   deriving (Enum, Show)
> pred Het
Vas
> succ Csut
Pen

> pred 100
```

```
99
> succ 'a'
'b'
```

Az `Integral` típusosztályt akkor használjuk, amikor egész számokkal szeretnénk műveleteket végezni. Magába foglalja az `Int` és az `Integer` típust.

```
class (Real a, Enum a) => Integral a  where
   quot, rem, div, mod :: a -> a -> a
   quotRem, divMod     :: a -> a -> (a,a)
   toInteger           :: a -> Integer
```

```
> mod (-3) 4
1
> rem (-3) 4
-3
```

A fenti két lekérdezés kapcsán vegyük észre, hogy a Haskell a legkisebb *pozitív* osztási maradék meghatározására a `mod` operátort vezeti be, míg a legkisebb osztási maradék kiszámításához a `rem` függvényt használja.

A **`Fractional`** típusosztályt akkor használjuk, amikor a `Num` osztályba tartozó paraméteren valós osztást, reciprok műveletet szeretnénk végrehajtani, illetve amikor egy `Rational` típusú adatot szeretnénk tizedestörtté alakítani.

```
class (Num a) => Fractional a where
    (/) :: a -> a -> a
    (recip) :: a -> a
    fromRational :: Rational -> a
```

```
> recip 2.0
0.5
```

```
> import Data.Ratio
> fromRational (13 % 15)
0.8666666666666667
```

A **`Floating`** magába foglalja a valós számokat kezelő, azaz a `Float` és `Double` típusokat, definíciója pedig a következő:

```
class (Fractional a) => Floating a where
    pi :: Floating a => a
    (**), logBase :: Floating a => a -> a -> a
    sqrt, exp, log :: Floating a => a -> a
```

```
    sin, cos, tan :: Floating a => a -> a
⋮
```

Utolsónak, a definíció megadása nélkül, a **Foldable** típusosztályt említjük meg. Ehhez a típusosztályhoz tartozik a lista típus, és azok a típusok is, amelyek olyan szerkezetű adatokat tárolnak, hogy alkalmazható rajtuk egyfajta *hajtogatásnak, foldingnak* nevezett művelet. Például egy egész számokat tartalmazó lista elemeit össze tudjuk adni, össze tudjuk szorozni, az eredmény pedig egy egész szám lesz. Ez hajtogatási műveletnek számít, mert egy adathalmazon egy olyan műveletet végzünk, amelynek eredménye *egyetlen* adat lesz. A `length`, `maximum`, `sum` stb. függvények típusdeklarációjában meg is jelenik a `Foldable` típusosztály mint megszorítás:

```
length :: Foldable t => t a -> Int
maximum :: (Foldable t, Ord a) => t a -> a
sum :: (Foldable t, Num a) => t a -> a
```

Típusosztályokat mi is írhatunk, például egy hasonló működésű típusosztály, mint az `Eq`, a következő lehetne:

```
class MyEq a where
    egyenloF :: a -> a -> Bool
    egyenloF x y = not (nemEgyenloF x y)

    nemEgyenloF :: a -> a -> Bool
    nemEgyenloF x y = not (egyenloF x y)
```

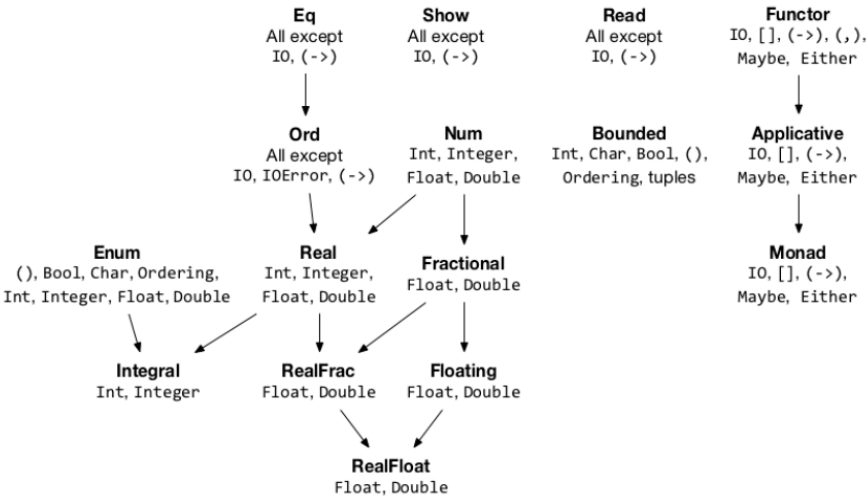
A `MyEq` keretén belül megadtunk két függvénydefiníciót, az `egyenloF` és `nemEgyenloF` függvényekét, amelyek hasonlóan működnek az `Eq` típusosztálynál definiált ==, illetve /= operátorokhoz. Ha használni szeretnénk, akkor példányosítani kell őket, de csak az egyik függvényt kell megírni, mert az `Eq`-ban az == operátor a /= operátorral volt megadva, és fordítva, a /= operátor az == operátorral volt értelmezve.

```
instance MyEq Bool where
    egyenloF True True = True
    egyenloF False False = True
    egyenloF _ _ = False

> egyenloF True True
True
```

Megjegyezzük még, hogy az `egyenloF` három esetet kezel, ahol az utolsó akkor kerül kiértékelésre, ha az előző kettő nem teljesül, és a használt _ szimbólum lehetővé teszi, hogy ne kelljen azonosítónevet választani a bemeneti paramétereknek.

A típusosztályok közötti kapcsolatrendszert a következő diagram szemlélteti:



4. fejezet

Jellemzők

Ebben a fejezetben a Haskell nyelv fontosabb jellemzőit mutatjuk be, amelyek legtöbbje bármilyen, tiszta funkcionális programozási nyelv elemei között megtalálhatók. A jellemzők bemutatása során példákat mutatunk, elmagyarázzuk a függvénytörzseket, a komplexebb tulajdonságokra pedig a jegyzet későbbi fejezeteiben még visszatérünk.

4.1. A Haskell típusrendszere

A Haskell típusrendszere **statikus** (static), ami azt jelenti, hogy a fordító már a fordítás során meghatározza a definíciók, kifejezések típusait, és a fennálló típushibákat jelzi. A lefordított program kifejezései a típushasználat szempontjából helyesek lesznek, még a program futtatása előtt.

Másfelől a Haskell típusrendszere **szigorú** (strong), ami azt jelenti, hogy az adatok típusa futási időben már nem változtatható meg. Ez azt is jelenti, hogy a programozó feladata a típusok közötti konverziók explicit megadása.

A `sum` és a `length` könyvtárfüggvények segítségével, ha meg akarjuk határozni egy valós elemű lista elemeinek átlagértékét, akkor explicit típuskonverziót kell használnunk. Ennek hiányában, ahogy az alábbi lekérdezésekből is látszik, futásidejű hibával szembesülünk.

```
> ls = [8.50, 9.75, 8.75, 7.50, 10, 8.25]
> sum ls / length ls
... error:
No instance for (Fractional Int)...
```

Az osztás során az a probléma adódott, hogy egy valós értéket akartunk osztani egy egész típusú értékkel, ugyanis a `sum` függvény kimenete a `Num` típusosztályba tartozik, a `length` függvény kimenete pedig egy `Int` típusú érték. Ugyanakkor a `/` operátor bemenetként egy `Fractional` típusosztályhoz tartozó értékeket vár, ahol a `Fractional`-hoz `Float` vagy `Double` típusok tartozhatnak. Mindezeket a következő lekérdezések is mutatják:

```
> :t sum
sum :: (Foldable t, Num a) => t a -> a

> :t length
length :: Foldable t => t a -> Int

> :t (/)
(/) :: Fractional a => a -> a -> a
```

A futási hiba elkerülése végett explicit típuskonverziót kell végrehajtanunk. Alkalmazzuk a `fromIntegral` függvényt, jelezve, hogy milyen típusról történjen az átalakítás.

```
> sum ls / fromIntegral (length ls)
8.791666666666666
```

4.2. Őrfeltételek

Egy Haskell-függvénytörzs keretén belül a `|` szimbólum használatával különböző feltételeket, pontosabban őrfeltételeket lehet megadni. Angolul azt mondjuk, hogy *guard*okat definiálunk. A függvény kimeneti értékét az első teljesülő feltételhez tartozó kifejezés adja.

4.1. feladat Írjunk egy Haskell-függvényt, amely meghatározza egy szám előjelét, azaz írjuk meg a `signum` függvényt.

Az `elojel` függvény kimenetének típusa `Int`, ahol a függvénytörzsben három feltételt adunk meg, így a végeredmény aszerint lesz `-1`, `1` vagy `0`, hogy az `x` milyen értéket jelöl.

```
elojel :: (Ord a, Num a) => a -> Int
elojel x
    | x < 0 = -1
    | x > 0 = 1
    | x == 0 = 0
```

```
> elojel (-10)
-1
```

4.2. feladat Írjunk egy Haskell-függvényt, amely a korábban definiált `Napok` típusú bemeneti érték esetében jelzi, hogy *hétvége* van vagy *hétköznap*.

A függvény megadása előtt szükséges, hogy módosítsuk a `Napok` definícióját azért, hogy alkalmazni tudjuk az == operátort.

```
data Napok = Vas | Het | Ked | Sze | Csut | Pen | Szo
   deriving (Enum, Show, Eq)

hetvege :: Napok -> String
hetvege x
   | x == Vas || x == Szo = "hetvege"
   | otherwise = "hetkoznap"

> hetvege Szo
"hetvege"
```

Figyeljük meg, hogy a `hetvege` függvény törzsében megjelenik egy új típusú feltétel, az `otherwise` ág. Ehhez a feltételhez tartozó kifejezés akkor értékelődik ki, amikor az őt megelőző sorokban megadott feltételek egyike sem teljesül. Az `otherwise` ág tehát a mindig igaz, azaz a `True` feltételnek felel meg.

4.3. A margószabály

Haskellben nem használunk kapcsos zárójeleket műveletblokkok jelölésére, helyette a bal oldali margó alapján különbözteti meg a fordító az összetartozó kifejezéseket, éppen ezért a margózás, vagy angolul a *layout rule*, fontos szintaktikai elem.

4.3. feladat Írjuk meg az `aritM` Haskell-függvényt, amely két egész szám esetében meghatározza a számok összegét, szorzatát, különbségét, osztási hányadosát, osztási egészrészét és osztási maradékát. Az eredmény hatelemű tuple típusú érték legyen.

```
aritM :: Int -> Int -> (Int, Int, Int, Double, Int, Int)
aritM x y = (r1, r2, r3, r4, r5, r6)
```

```
    where
    r1 = x + y
    r2 = x * y
    r3 = x - y
    r4 = fromIntegral x / fromIntegral y
    r5 = div x y
    r6 = mod x y

> aritM 11 7
(18, 77, 4, 1.5714285714285714, 1, 4)
```

A fenti kódsorban a margószabály betartása mellett arra is oda kell figyelni, hogy az osztási hányados meghatározásához típuskonverzióra van szükség, alkalmazni kell a `fromIntegral`-t.

A függvényszignatúrát megadhatjuk típusváltozók segítségével, és amennyiben ezt nem szeretnénk egy sorba írni, akkor figyelembe kell venni itt is a margószabályt. Több sorba tördelve természetesen olvashatóbb formát kapunk:

```
aritM ::
    (Integral a, Fractional a1) =>
    a -> a -> (a, a, a, a1, a, a)
```

Margózással különböző kifejezések, függvénydefiníciók láthatóságát is korlátozhatjuk, kiterjeszthetjük, ahogy azt a következő feladat mutatja.

4.4. feladat Írjunk egy Haskell-függvényt, amely meghatározza egy másodfokú egyenlet valós gyökeit.

```
mEgyenlet :: (Floating a, Ord a)
             => a -> a -> a -> (a, a)
mEgyenlet a b c
   | delta < 0 = error "Komplex gyokok"
   | otherwise  = (x1, x2)
       where
       x1 = (-b + sqrt delta) / tmp
       x2 = (-b - sqrt delta) / tmp
       delta = b * b - 4 * a * c
       tmp = 2 * a

delta :: Num a => a -> a
delta a = 3 * a
```

A fenti kódsorokban két `delta` kifejezést is definiáltunk, felvetődik a kérdés, hogy az alábbi lekérdezésekben mikor melyik `delta` kifejezés értékelődik ki?

```
> mEgyenlet 2 3 1
> delta 2
```

Az `mEgyenlet` a `where` kulcsszó alatt definiált `delta` kifejezést használja, mert helyileg ez tartozik hozzá, a második lekérdezésben pedig a `delta a = 3 * a` kifejezés kerül kiértékelésre. Ha azonban rosszul tördelünk, fordítási hibába ütközhetünk.

4.4. Rekurzió

A rekurzió a funkcionális nyelvek alap vezérlési szerkezete, ami azt jelenti, hogy a függvények hivatkozhatnak önmagukra és kölcsönösen egymásra. Az első fejezetet egy Haskell-függvény megadásával kezdtük, a faktoriális függvény törzsét adtuk meg több módszerrel, most további függvényeket írunk, szintén rekurzívan.

4.5. feladat Írjunk egy-egy Haskell-függvényt, ahol az egyik kivonásos módszerrel, a másik Eukleidész módszerével határozza meg két egész szám legnagyobb közös osztóját.

Kivonásos módszer:

```
lnko :: (Ord a, Num a) => a -> a -> a
lnko a b
   | a < 0 || b < 0
       = error "ez csak pozitiv szamokra mukodik"
   | a > b = lnko (a - b) b
   | a < b = lnko a (b - a)
   | otherwise = a
```

Eukleidész módszere:

```
euclid :: Integral a => a -> a -> a
euclid a b
   | b == 0 = abs a
   | otherwise = euclid b (mod a b)
```

A lekérdezések és az eredmények pedig a következők:

```
> lnko 24 204
12

> euclid 33989590165734525335 42391251154308366336
13
```

Algoritmikailag mindkét függvény jól ismert, az `lnko` esetében azt az elgondolást követjük, hogy a nagyobbik paraméterből ki kell vonni a kisebbik paraméter értékét, mígnem két egyforma értéket nem kapunk, ami a kezdeti bemenetek legnagyobb közös osztója lesz. Az `euclid` függvény esetében, a rekurzív függvényhíváskor az első paraméter a `b` lesz, a második pedig az `a` és `b` osztási maradéka. A rekurzív függvénytörzset tehát úgy is értelmezhetjük, hogy ha `b` nulla, akkor az `a` és `b` legnagyobb közös osztója `a`, ellenkező esetben megegyezik a `b` és `mod a b` legnagyobb közös osztójával.

Rekurzív függvényhívások esetében különösen oda kell figyelni arra, hogy a sajátos, a triviális eseteket külön feltételben kezeljük, és amikor önmagát hívja egy függvény, akkor úgy válasszuk meg az új paramétereket, hogy az ne eredményezzen végtelen számítási folyamatot.

A Haskell `gcd` (greatest common divisor) függvénye az euklideszi algoritmus alapján határozza meg két szám legnagyobb közös osztóját:

```
> gcd 3792853 187589173
47
```

4.6. feladat Írjunk egy Haskell-függvényt, amely meghatározza az `n`-edik Fibonacci-számot.

Definíció szerint az első Fibonacci-szám a `0`, a második az `1`, a számsorozat `n`-edik tagját pedig úgy határozzuk meg, hogy összeadjuk az `n-1`-edik, illetve `n-2`-edik tagot.

A kitűzött példát háromféleképpen is megoldjuk. Az első módszer során a fenti definíció alapján fogjuk a rekurzív függvényhívásokat megadni, amely esetben azt fogjuk tapasztalni, hogy a `25`-ödik Fibonacci-szám meghatározására már *várnunk* kell.

```
fibonacciE :: (Ord a, Num a, Num b) => a -> b
fibonacciE n
    | n < 0 = error "hibas bemenet"
    | n == 0 = 0
    | n == 1 = 1
    | otherwise = fibonacciE (n - 1) + fibonacciE (n - 2)
```

Ahhoz, hogy lássuk, hogy mennyi idő szükséges a kiértékeléshez, kapcsoljuk be az *időmérést*:

```
> :set +s
> fibonacciE 25
75025
(0.45 secs, 77,318,704 bytes)
```

A fenti algoritmus futási ideje exponenciális, azaz lassú, és ezt a lassúságot nem a rekurzió okozza. Ahhoz, hogy az algoritmus futási idején javíthassunk, például lineáris futásidejű algoritmust találjunk, másképp kell gondolkodnunk. Ehhez két függvényt fogunk írni. A `fibonacci` függvény szerepe az lesz, hogy kezelni tudjuk a hibás bemenetet, illetve inicializálhassuk a tulajdonképpeni számításokat végző `auxFib` függvény paramétereit. Az `auxFib` első paramétere az `n` értékét (a hátralevő lépések számát), a második, illetve a harmadik paramétere pedig az `i`-edik, illetve `i+1`-edik Fibonacci-szám értékét fogja jelölni. A kezdeti meghíváskor, a helyes eredmény meghatározásához, a nulladik és az első Fibonacci-szám értékét kell megadni.

```
fibonacci :: (Ord a, Num a, Num b) => a -> b
fibonacci n
    | n < 0 = error "hibas bemenet"
    | otherwise = auxFib n 0 1
    where
    auxFib :: (Eq a, Num a, Num b) => a -> b -> b -> b
    auxFib n a b
        | n == 0 = a
        | otherwise = auxFib (n - 1) b (a + b)

> fibonacci 25
75025
(0.00 secs, 59,088 bytes)
```

A kapott futási idő is azt mutatja, hogy lényegesen hatékonyabb a második változat. A gyakorlatban létezik logaritmikus futási idejű algoritmus is, amelyről a fejezet végén található kitűzött feladatoknál lesz szó.

A harmadik változat nem fog javítani az algoritmus hatékonyságán, összehasonlításképpen adjuk meg. Az `auxFib` függvény a számításokat másképp végzi, akkor fog számolni, amikor *jön vissza* a rekurzióból, ellentétben az előzővel, amely akkor számol, amikor *megy be* a rekurzióba. A `fibonacci_` visszatérési értéke pedig az `auxFib` által meghatározott kételemű tuple első eleme lesz.

```
fibonacci_ :: (Eq a, Num a, Num b) => a -> b
fibonacci_ n = t1
    where
```

```
    (t1, t2) = auxFib n
    auxFib :: (Eq a, Num a, Num b) => a -> (b, b)
    auxFib n
        | n == 0 = (0, 1)
        | otherwise = (b, a + b)
            where
            (a, b) = auxFib (n - 1)
```

Fontosnak tartjuk megjegyezni, hogy egy `hs` állományon belül ugyanolyan nevű függvényeket akkor használhatunk, ha azok láthatósági zónája nincs átfedésben. A fenti kódsorokban az `auxFib` függvények láthatóságát a tördelés segítségével korlátoztuk, ezért a hívó függvények tudni fogják, melyiket értékeljék ki.

4.5. Mintaillesztés

Az argumentumok mintaillesztése, a *pattern matching* azt jelenti, hogy a függvény értékét az a kifejezés határozza meg, amelyre a függvényparaméterek egy megadott minta alapján illeszkednek. A mintaillesztést és a feltételek megadását lehet együttesen is alkalmazni. Mintaillesztést használva egy függvénytörzset egyszerűbbé, olvashatóbbá lehet tenni, ezek használata különösen jellemző a funkcionális programozási nyelvek körében.

4.7. feladat Írjunk egy Haskell-függvényt, amely meghatározza egy szám számjegyeinek összegét.

```
szOsszeg :: Integral a => a -> a
szOsszeg 0 = 0
szOsszeg x = mod x 10 + szOsszeg (div x 10)

> szOsszeg 1234
10
```

4.8. feladat Írjunk egy Haskell-függvényt, amely meghatározza egy szám számjegyeinek szorzatát.

```
szSzorzat :: Integral a => a -> a
szSzorzat 0 = 0
szSzorzat x = auxSz 1 x
   where
   auxSz res 0 = res
```

```
    auxSz res x = auxSz (res * mod x 10) (div x 10)

> szSzorzat 1234
24
```

Mindkét függvény mintaillesztést alkalmaz, és két-két esetet vizsgál. A függvénytörzsek első sorai azt az esetet kezelik, amikor a bemenet nulla, a második sorok pedig általános esetben adják meg, hogy mit kell kiértékelni.

A `szOsszeg` függvény a számjegyek összegzését akkor számítja ki, amikor *jön vissza* a rekurzióból, ezért a triviális esetben a függvény visszatérési értéke `0`. A `szSzorzat` másképp jár el, akkor szoroz, amikor *megy be* a rekurzióba. A `szSzorzat`, amikor a bemenete nem nulla, a lokálisan megadott `auxSz` függvényt fogja kiértékelni. A tulajdonképpeni számításokat tehát az `auxSz` végzi. Az `auxSz` úgy kerül meghívásra, hogy az első paraméterének értékét `1`-re inicializáltuk, amely minden egyes rekurzív híváskor aktualizálódik, és a részszorzat aktuális értékét fogja jelölni. Az eredmény tehát a rekurzió legalsó szintjén, az `auxSz` első argumentumában meg lesz határozva, így a triviális esetben ezt az értéket fogja visszaadni az `auxSz`.

Vegyük észre, hogy a függvények nem kezelik a negatív bemeneti értéket, az olvasóra bízzuk, hogy módosítsa úgy a függvénytörzseket, hogy azok ne eredményezzenek futási hibát negatív bemenetre.

Megjegyezzük, hogy az `szOsszeg` függvény az `szSzorzat` függvénynél alkalmazott technikával is megoldható, illetve a `szSzorzat` esetében is alkalmazhattuk volna azt a technikát, hogy akkor számolunk, amikor jövünk vissza a rekurzióból. Ezekben az esetekben sincs hatékonyságbeli különbség a két módszer között. Gyakorlás végett írjuk meg őket.

4.6. Feltételes kifejezések

Az `if...then...else` kifejezésre már korábban is láttunk példát, most azonban azt is ki szeretnénk hangsúlyozni, hogy az `if` a Haskellben nem utasítás vagy állítás, hanem egy feltételes kifejezés, ezért az `else` ág kötelező.

4.9. feladat Írjunk egy Haskell-függvényt, amely megvizsgálja, hogy `x` osztható-e `y`-nal.

```
oszthato :: (Integral a) => a -> a -> Bool
oszthato x y = if mod x y == 0 then True else False
```

```
> oszthato 21 7
True
```

A korábban megírt faktoriális függvényt is lehet feltételes kifejezésekkel definiálni:

```
fakt :: (Eq a, Num a) => a -> a
fakt n = if n == 0 then 1 else n * fakt (n-1)

> fakt 100
933262154439441526816992388562667004907159682643 8...
```

Többágú kifejezések megadására a `case...of` kifejezés alkalmazható, amelynek használatát egy feladaton keresztül mutatjuk be, ahol szükségünk lesz a `chr` függvényre is. Ahhoz, hogy tudjuk ezt használni, importálni kell a `Data.Char` könyvtárcsomagot. A `chr` függvény meghatározza azt az Unicode kódolás szerinti szimbólumot, amely a paraméterként megadott egész számhoz tartozik.

4.10. feladat Írjuk meg a `hexaSz` Haskell-függvényt, amely meghatározza azt a `16`-os számrendszerben használt szimbólumot, amelyet egy `0` és `15` közötti *számjegyhez* rendelnek hozzá. A kiíratást az `A, B, C, D, E, F` szimbólumok segítségével végezzük.

```
import Data.Char
hexaSz :: Int -> Char
hexaSz c
   | c >= 0 && c < 16 =
       case c of
           10 -> 'A'
           11 -> 'B'
           12 -> 'C'
           13 -> 'D'
           14 -> 'E'
           15 -> 'F'
           _ -> chr (c + 48)
   | otherwise = error "hibas bemenet"

> hexaSz 15
'F'
```

```
> hexaSz 19
*** Exception: hibas bemenet...
```

A fenti kódsorban a `case` hét esetet tárgyal aszerint, hogy a `c` értéke mennyi. Az utolsó eset, azaz a `case` utolsó sora azt az esetet kezeli, amikor a korábbi feltételek közül egyik sem teljesült. A `_` szimbólum *mindenes* operátor, szerepe az, hogy olyan helyzeteket kezeljen, amikor nem számít a bemenet vagy a mintázott érték. Használatával a fenti függvényben a *minden más* esetben adjuk meg a függvény kimeneti értékét, azaz ha a `c` nem egyenlő `10, 11, 12, 13, 14, 15`-tel, akkor a `chr (c + 48)` kifejezés kerül kiértékelésre. Mivel a `'0'` karakter kódja `48`, a `chr (c + 48)` kifejezés meghatározza a `c` számjegy *karakter-értékét*.

4.11. feladat Írjunk egy Haskell-függvényt, amely meghatározza egy `16`-os számrendszerben megadott számsorozatnak azon alakját, amelyben a számértékeket a `16`-os számrendszerben használt szimbólumokkal helyettesítjük. Használjuk a korábban megírt `hexaSz` függvényt.

```
hexaLs :: [Int] -> [Char]
hexaLs [] = ""
hexaLs (k : ve) = hexaSz k : hexaLs ve

> hexaLs [12, 4, 5, 15, 7, 0, 11, 4]
"C45F70B4"

> hexaLs [12, 4, 5, 17, 7, 0, 11, 4]
*** Exception: hibas bemenet...
```

A függvénytörzs mintaillesztés alapján választja szét a triviális esetet az általánostól, azaz amikor a bemenet üres lista, illetve amikor nem. Ez utóbbi esetben a `:` operátor segítségével választottuk le a lista első elemét, a `k`-t a lista többi részétől, a `ve`-től. Az egyenlőség jobb oldalán a `k`, illetve `ve` étékekkel így műveleteket tudunk végezni. Ez jelen esetben azt jelenti, hogy a bemeneti egész számokat tartalmazó lista alapján felépítünk egy új listát. Az új listát úgy építjük fel, hogy alkalmazzuk a `hexaSz` függvényt a bemeneti lista első elemén, a `k`-n, majd ezt az értéket a `:` operátor segítségével a rekurzív hívás során nyert lista elé fűzzük. Az újonnan épített lista lesz a függvény kimenetének értéke. Figyeljük meg a `:` operátor szerepét az egyenlőség bal, illetve jobb oldalán, a későbbiekben még visszatérünk rá.

4.7. Halmazkifejezések

Halmazkifejezéseket vagy listagenerátorokat iteratív adatszerkezetek (listák, halmazok, sorozatok) elemeinek megadásakor alkalmazunk. Az angol terminológia erre a *list comprehension* megnevezést használja. Ezzel a jelölésrendszerrel nagyon egyszerűen lehet listaelemeket megadni, generálni, feldolgozni. A legegyszerűbb feladatok egy lista elemeiből kiindulva azon listaelemekből hoznak létre új listát, amelyek eleget tesznek egy adott feltételnek.

4.12. feladat Írjunk egy Haskell-függvényt, amely meghatározza a paraméterként megadott szám osztóit.

```
osztok :: Integral a => a -> [a]
osztok n = [i | i <- [1..n], mod n i == 0]

> osztok 60
[1,2,3,4,5,6,10,12,15,20,30,60]
```

A kódsorban azokból az `i` értékekből hozzuk létre az eredménylistát, amelyek az `[1..n]` listának elemei, és eleget tesznek a `mod n i == 0` feltételnek.

Halmazkifejezések megadásakor egy adott lista elemei alapján hozunk létre egy új listát, ahol az új listaelemekre vonatkozó szabályokat szögletes zárójelben kell megadni. A szögletes zárójelbe írtak két részre oszthatók, a `|` jel előtti részre, illetve az utána következő részre. Az első részben az új lista elemeit vagy a listaelemekre vonatkozó kifejezéseket adhatunk meg. A második részben a `<-` szimbólum segítségével a listaelemek generálási módját határozzuk meg. A `,` utáni rész nem kötelező, ide logikai kifejezések, megszorítások írhatók.

4.13. feladat Írjunk egy Haskell-függvényt, amely meghatározza egy `16`-os számrendszerben megadott számsorozatnak azon alakját, ahol a számértékeket a `16`-os számrendszerben használt szimbólumokkal helyettesítjük. Használjuk a korábban megírt `hexaSz` függvényt.

```
hexaLc :: [Int] -> [Char]
hexaLc ls = [hexaSz k | k <- ls]

> hexaLc [12, 4, 5, 15, 7, 0, 11, 4]
"C45F70B4"
```

A feladatot korábban is megoldottuk, most halmazkifejezéssel azonban egy egyszerűbb kódsort tudtunk írni. Fontos ugyanakkor leszögezni, hogy hatékonyság szempontjából ez a módszer nem lesz jobb.

4.14. feladat Írjunk egy Haskell-függvényt, amely egy olyan háromelemű tuple-ökből álló listát generál, ahol a tuple-elemek a `0, 1,...,n` összes lehetséges értékeit felveszik, az összes lehetséges módon.

```
tupleLc :: (Num a, Enum a) => a -> [(a, a, a)]
tupleLc n = [(x, y, z) | x <- [0..n], y <- [0..n],
   z <- [0..n]]

> tupleLc 1
[(0,0,0),(0,0,1),(0,1,0),(0,1,1),(1,0,0),(1,0,1),
(1,1,0),(1,1,1)]
```

4.15. feladat Írjunk egy Haskell-függvényt, amely meghatározza adott `n`-ig a pitagoraszi számhármasokat, ahol az `x, y, z` három természetes szám, pitagoraszi számhármast alkot, ha fennáll: x^2 `==` y^2 `+` z^2.

A következő két függvényben kétféleképpen is megadjuk a generálási szabályt, illetve a feltételeket, ahol a második módszer lesz a hatékonyabb.

```
pitagorasz :: (Num a, Enum a, Ord a) => a -> [(a, a, a)]
pitagorasz n = [(x, y, z) | x <- [1..n], y <- [1..n],
   z <- [1..n], x * x == y * y + z * z, y < z]

pitagorasz_ :: (Num a, Enum a, Eq a) => a -> [(a, a, a)]
pitagorasz_ n = [(x, y, z) | x <- [1..n], y <- [1..n],
   z <- [y + 1..n], x * x == y * y + z * z]

> pitagorasz 17
[(5,3,4),(10,6,8),(13,5,12),(15,9,12),(17,8,15)]
```

4.16. feladat Írjunk egy Haskell-függvényt, amely kiválasztja egy listából a négyzetszámokat.

```
valasztN :: Integral a => [a] -> [a]
valasztN ls = [i | i <- ls, negyzetV i]

negyzetV :: Integral a => a -> Bool
negyzetV x = temp * temp == x
   where
```

```
    temp = truncate (sqrt (fromIntegral x))

> valasztN [12, 144, 200, 196, 154, 9, 8, 4, 6, 100, 625]
[144,196,9,4,100,625]
```

A négyzetszámok vizsgálatát a `negyzetV` függvény végzi, amely előbb négyzetgyököt számol a bemeneti paraméteren, majd a `truncate` függvénnyel levágja a tizedesjegyeket, és az így kapott szám négyzetéről vizsgálja meg, hogy egyenlő-e a bemenettel. Az `sqrt` bemenete `Float` vagy `Double` típusú, ezért a `fromIntegral` függvénnyel típuskonverziót hajtottunk végre.

4.17. feladat Írjunk egy Haskell-függvényt, amely a bemeneti lista elemeit kétfelé válogatja, meghatározza egy listába a prímszámokat, egy másikba pedig az összetett számokat.

```
import Data.List

valogatNr :: (Integral a) => [a] -> ([a], [a])
valogatNr ls = (primL, osszL)
   where
   primL = [i | i <- ls, primT 3 i]
   osszL = ls \\ primL

> valogatNr [24, 97, 5, 11, 74, 41, 61, 19, 100]
([97,5,11,41,61,19],[24,74,100])
```

A kért listákat előállító főfüggvény a `valogatNr` lesz, amelyben a `primT` segítségével határoztuk meg a prímszámokat. A `primT`-nek két paramétere van, a másodikról vizsgálja, hogy prímszám-e úgy, hogy az első paraméterrel való oszthatóságot figyeli.

```
primT :: (Integral a) => a -> a -> Bool
primT k nr
   | nr <= 1 = error "hibas bemenet"
   | nr == 2 = True
   | even nr = False
   | nr < k * k = True
   | mod nr k == 0 = False
   | otherwise = primT (k + 2) nr

> primT 3 1789
True
```

```
> primT 3 100
False
```

A `primT` első három feltétele a triviális eseteket kezeli. Az `even` könyvtárfüggvény, amelynek `True` a visszatérési értéke, ha a bemenet páros szám, ellenkező esetben `False`. A negyedik feltétel akkor teljesül, ha a szám négyzetgyökéig nem találtunk páratlan osztót, ebben az esetben `True` lesz a kimenet értéke, mert a szám prímszám. A négyzetgyök vizsgálata nem explicit módon történik, hanem helyette megvizsgáljuk, hogy az osztó négyzete nem nagyobb-e a számnál. Az ötödik feltételben kerül sor az oszthatóság vizsgálatára, amely ha fennáll, akkor `False` lesz a függvény kimeneti értéke, mert a szám összetett. Az `otherwise` ágban a rekurzív függvényhívásra kerül sor. Itt figyeljük meg, hogy a `k` értékét kettesével léptetjük, mert a hatékonyság miatt az oszthatóságot csak páratlan számokra teszteljük, a páros bemenetet ugyanis egy korábbi feltételben kezeltük. Ahhoz, hogy páratlan számokkal végezhessük az oszthatóságot, a `primT` függvény első paraméterének kezdőértékét `3`-ra állítottuk.

A `valogatNr` függvényben a `primL` listába azokat a számokat tesszük, amelyek prímszámok. Az `osszL` az összetett számok listáját jelöli, ahol két lista különbségének a meghatározásához az `\\` operátort használtuk.

A `\\` operátor a `Data.List` könyvtármodulban van, tetszőleges elemtípusú listákon alkalmazható. Az eredeti listában a megadott elemek első előfordulását törli, meghagyva a többi előfordulási értéket. Figyeljük meg, hogyan kell eljárni, ha egy könyvtárcsomagból csak egy függvényt szeretnénk használni. A következő lekérdezések előtt a `\\` használatához nem importáljuk a teljes könyvtárcsomagot.

```
> import Data.List ( (\\) )

> [45, 7, 8, 12, 3, 9, 10] \\ [45, 10, 12]
[7,8,3,9]

> [45, 7, 8, 12, 45, 3, 45, 9, 10] \\ [45, 10, 12]
[7,8,45,3,45,9]

> "Nyugati-Karpatok" \\ "aeiou"
"Nygt-Karpatk"
```

4.8. Lambda kifejezések

Lambda kifejezésekkel függvények egy alternatív definiálási módját tudjuk megvalósítani, amelyet akkor használunk, ha nem akarunk nevet választani egy adott segédfüggvénynek, ezért ezeket névtelen függvényeknek is hívják. A lambda-függvények definiálásakor a `\` szimbólumot használjuk. Lehetnek egy- vagy többparaméteresek, de a függvénytörzs nem tartalmazhat őrfeltételeket, mintaillesztéseket. Ezért vigyázni kell, hogy a függvénydefiníció minden lehetséges helyzetre kiértékelődjön, ne lépjen fel futási hiba.

4.18. feladat Írjunk egy Haskell-függvényt, amely a paraméterként megadott lista elemeinek értékét `1`-gyel növeli.

```
novelLs :: (Num a) => [a] -> [a]
novelLs ls = [(\ x -> x + 1) k | k <- ls]

> novelLs [5.75, 7.33, 8.75, 6.25]
[6.75,8.33,9.75,7.25]
```

4.19. feladat Írjunk egy Haskell-függvényt, amely a paraméterként megadott lista elemeinek értékét `p`-vel növeli.

```
novelLsP :: (Num a) => a -> [a] -> [a]
novelLsP p ls = [(\ x -> x + p) k | k <- ls]

> novelLsP 5 [-10, 8, -6, -4, 23, 101]
[-5,13,-1,1,28,106]
```

4.20. feladat Írjunk egy Haskell-függvényt, amely létrehoz egy `n` elemű listát, amely váltakozva tartalmaz `True` és `False` értékeket.

```
valtakoz :: Integral a => a -> [Bool]
valtakoz n = [(\ x -> even x) k | k <- [0..n-1]]

> valtakoz 5
[True,False,True,False,True]
```

A fenti függvények megadhatók lambda kifejezések használata nélkül, a következő kódsorok ezt mutatják:

```
novelLs_ :: (Num a) => [a] -> [a]
novelLs_ ls = [k + 1 | k <- ls]
```

```
novelLsP_ :: (Num a) => a -> [a] -> [a]
novelLsP_ p ls = [k + p | k <- ls]

valtakoz_ :: Integral a => a -> [Bool]
valtakoz_ n = [even k | k <- [0..n-1]]
```

4.9. Magasabb rendű függvények

Haskellben a magasabb rendű függvények olyan függvények, amelyeknek bemeneti paraméterük és kimeneti értékük is lehet függvény. Angolul *higher-order function* a megnevezésük.

4.21. feladat Írjunk egy Haskell-függvényt, amely a paraméterként megadott *függvényt* duplán alkalmazza egy megadott bemeneten.

```
duplaz :: (a -> a) -> a -> a
duplaz fg x = fg (fg x)

> duplaz (\ x -> x + 1) 10
12

> duplaz sqrt 2
1.189207115002721
```

A `duplaz` magasabb rendű függvény, mert az `fg` paraméter egy függvény. Működés szempontjából az `fg` függvényt kétszer alkalmazza a második paraméterére.

A Haskellben több könyvtárfüggvény is létezik, amely magasabb rendű függvény, ilyenek például a `map`, `filter`, `foldr`, `foldl` stb. A továbbiakban a `map` és a `filter` függvények működését tárgyaljuk, a többi függvényre a jegyzet későbbi fejezeteiben kerül sor.

A `map` függvénynek két argumentuma van: a második egy lista, az első pedig egy függvény, amelyet alkalmaz a lista minden elemére.

4.22. feladat Írjunk egy olyan Haskell-lekérdezést, amely meghatározza a paraméterként megadott `Double` elemtípusú lista elemeinek négyzetgyökét.

```
> map sqrt [3.0, 4.0, 5.0, 6.0, 7.0]
[1.7320508075688772, 2.0,2 .23606797749979,
 2.449489742783178, 2.6457513110645907]
```

Az előző fejezetben megírt `novelLs`, `novelLsP`, illetve `valtakoz` függvények is megadhatók `map`-et használva. Figyeljük meg, hogy a függvények lista típusú bemeneti paraméterei nincsenek feltüntetve. Haskellben egy függvény utolsó paraméterei ugyanis elhagyhatók, ha a törzsében is, számításba véve a sorrendet, utolsó paraméterként jelennek meg.

```
novelLsMap :: (Num a) => [a] -> [a]
novelLsMap = map (+ 1)

novelLsPMap :: (Num a) => a -> [a] -> [a]
novelLsPMap p = map (+ p)

valtakozMap :: Integral a => a -> [Bool]
valtakozMap n = map even [0..n-1]
```

4.23. feladat Írjunk egy olyan Haskell-lekérdezést, amely meghatározza minden listabeli elemre a páros osztók listáját.

```
parosO :: (Integral a) => a -> (a, [a])
parosO n = (n, [i | i <- [2, 4..div n 2], mod n i == 0])

> parosO 60
(60,[2,4,6,10,12,20,30])
```

Első lépésként írtunk egy `parosO` függvényt, amely egy kételemű tuple típusú értéket határozott meg, ahol a tuple első eleme maga a bemenet, második eleme pedig egy lista, amelyben a bemenet páros osztóit generáltuk ki. Ha a bemenetnek nincs páros osztója, akkor az eredmény üres lista lesz.

Második lépésként meghívjuk a `map` függvényt úgy, hogy első paramétere a `párosO` függvény, második pedig egy lista legyen, amelybe azokat az számokat írjuk, amelyeknek a páros osztóit szeretnénk meghatározni.

A következő lekérdezés megadja az `50` és `59` közötti számok páros osztóinak listáját.

```
> map parosO [50..59]
[(50,[2,10]),(51,[]),(52,[2,4,26]),(53,[]),(54,[2,6,18]),
(55,[]),(56,[2,4,8,14,28]),(57,[]),(58,[2]),(59,[])]
```

A `filter` függvénynek is két argumentuma van: a második egy lista, az első pedig egy logikai függvény, amely alapján kiválasztásra kerülnek a listabeli elemek.

4.24. feladat Írjunk egy Haskell-függvényt, amely kiválasztja egy lista elemei közül a páros számokat.

```
parosLs :: (Integral a) => [a] -> [a]
parosLs = filter even

> parosLs [1..20]
[2,4,6,8,10,12,14,16,18,20]
```

Bemenetként általunk megírt logikai függvényt is megadhatunk. Például a négyzetszámok listáját a következőképpen is kigenerálhatjuk, ahol alkalmazni fogjuk a korábban megírt `negyzetV` függvényt:

```
> filter negyzetV [1..100]
[1,4,9,16,25,36,49,64,81,100]
```

A magasabb rendű függvényeknek van egy másik fontos jellemzőjük is: részlegesen lehet paraméterezni őket. Ezt curryzésnek is mondjuk, Haskell Brooks Curry matematikus után. A részleges paraméterezés azt jelenti, hogy a függvényhívás megengedett *látszólag* kevesebb paraméterrel is.

Például ha a páratlan prímszámok listáját szeretnénk kigenerálni `100`-ig, akkor a `filter` és a korábban megírt `primT` részleges paraméterezésével ezt egy egysoros lekérdezésben megtehetjük:

```
> filter (primT 3) [3,5..100]
[3,5,7,11,13,17,19,...,83,89,97]
```

4.25. feladat Írjunk egy Haskell-függvényt, amely az `x` és `k` bemenetekre, ahol `k` egész szám, meghatározza az x^0, x^1, ..., x^k értékeket.

```
fugv1 :: (Integral a, Num b) => b -> a -> [b]
fugv1 x k = map (x ^) [0..k]

> fugv1 7 6
[1,7,49,343,2401,16807,117649]
```

Az implementáció során a `map` könyvtárfüggvényt használtuk, ahol a `map` függvény első paraméterét a hatványozó operátort, infix formában, részlegesen paraméterezve adtuk meg.

4.26. feladat Írjunk egy Haskell-függvényt, amely az `x` és `k` bemenetekre meghatározza az 0^k, 1^k, ..., x^k értékeket.

```
fugv2 :: (Integral a, Num b, Enum b) => b -> a -> [b]
fugv2 x k = map (^ k) [0..x]

> fugv2 7 6
[0,1,64,729,4096,15625,46656,117649]
```

Vegyük észre, hogy a feladat most különböző alapú, de ugyanolyan hatványkitevőn levő értékeket kell kiszámoljon. Ezért a `^` operátort megint infix formában hívtuk úgy, hogy a `^` operátor első argumentuma rendre a `[0..x]` lista elemei legyenek.

Megjegyezzük azt is, hogy ha módosítjuk a zárójelezést, akkor a `^` operátor prefix formában kerül meghívásra, ami más eredményt fog adni. A következő kód a k^0, k^1, ..., k^x értékeket határozza meg.

```
fugv2_ :: (Integral a, Num b) => a -> b -> [b]
fugv2_ x k = map ((^) k) [0..x]

> fugv2_ 7 6
[1,6,36,216,1296,7776,46656,279936]
```

A következőkben másképp járunk el. A beépített operátor helyett megírjuk a saját hatványozó függvényeinket, és a `map` függvénynek ezeket adjuk meg paraméternek.

A `myPow1` függvény tulajdonképpen a gyorshatványozás algoritmusa, első paramétere az alap, második a hatványkitevő lesz, ahol a rekurzív hívásra a `where` kifejezésben kerül sor. A rekurzív hívás eredményét a `temp` fogja jelölni, amelyet az őrfeltételekben szorzótényezőként használunk. A szükséges szorzásokat az határozza meg, hogy a hatványkitevő páros vagy páratlan szám.

```
myPow1 :: (Integral a) => a -> a -> a
myPow1 x n
    | n < 0 = error "Negativ kitevo"
    | n == 0 = 1
    | even n = temp * temp
    | otherwise = x * temp * temp
        where
        temp = myPow1 x (div n 2)
```

A következő függvényekben háromféleképpen kerül kiértékelésre a `myPow` függvény. A `fugvA`-ban prefix formában használjuk, hogy az x^0, x^1, ..., x^k értékeket tudjuk meghatározni.

```
fugvA :: Integral a => a -> a -> [a]
fugvA x k = map (myPow1 x) [0..k]

> fugvA 7 6
[1,7,49,343,2401,16807,117649]
```

A `fugvB`-ben infix formában alkalmazzuk a `myPow1` függvényt, azért, hogy a 0^k, 1^k, ..., x^k értékek kerüljenek kiértékelésre.

```
fugvB :: Integral a => a -> a -> [a]
fugvB x k = map (`myPow1` k) [0..x]

> fugvB 7 6
[0,1,64,729,4096,15625,46656,117649]
```

A `fugvC`-ben alkalmazásra kerül a beépített `flip` függvény, amely segítségével a paraméterek sorrendjét lehet megváltoztatni, így most is a 0^k, 1^k, ..., x^k értékeket határozza meg a kiértékelés.

```
fugvC :: Integral a => a -> a -> [a]
fugvC x k = map (flip myPow1 k) [0..x]

> fugvC 7 6
[0,1,64,729,4096,15625,46656,117649]
```

A `flip` függvény megértéséhez figyeljük meg a következő kiértékelések eredményeit.

```
> myPow1 2 10
1024
> flip myPow1 2 10
100
```

A következő `myPow2` függvény is az x^n értékét határozza meg, de algoritmikailag különbözik a `myPow1`-től. Az alap négyzetre emelését a rekurzív függvényhíváskor valósítja meg, illetve a paraméterek sorrendjét is felcserélték.

```
myPow2 :: (Integral a, Num b) => a -> b -> b
myPow2 n x
    | n < 0 = error "negativ kitevo"
    | n == 0 = 1
    | even n = temp
    | otherwise = x * temp
        where
        temp = myPow2 (div n 2) (x * x)
```

Az olvasóra bízzuk, hogy az előző `fugvA`, `fugvB`, illetve `fugvC` függvények mintájára, egy `map`-ben részlegesen paraméterezve alkalmazza a `myPow2`-t.

4.27. feladat Írjunk egy Haskell-függvényt, amely kiválasztja egy lista elemei közül az `x`-szel osztható számokat.

```
fugv3 :: (Integral a) => a -> [a] -> [a]
fugv3 x ls = filter (oszthato x) ls
   where
   oszthato :: (Integral a) => a -> a -> Bool
   oszthato x y = mod y x == 0

> fugv3 7 [1..100]
[7,14,21,28,35,42,49,56,63,70,77,84,91,98]
```

Az `oszthato` függvénnyel korábban is foglalkoztunk. A függvény megvizsgálja, hogy `x` osztója-e `y`-nak, ahol a függvénytörzset a korábbi változathoz képest egysorosra módosítottuk, és felcseréltük a paraméterek szerepét.

4.10. Függvénykompozíció

A függvénykompozíció a matematikából jól ismert művelet. A Haskellben lehetőségünk van hasonló művelet definiálására, jelölésére a pont (`.`) szimbólumot használjuk. A következő példában két könyvtárfüggvény kompozíciójával dolgozunk, a `not`-tal és az `even`-nel.

4.28. feladat Írjunk egy Haskell-függvényt, amely kiválasztja egy lista elemei közül a páratlan számokat.

```
paratlanLs :: (Integral a) => [a] -> [a]
paratlanLs = filter (not . even)

> paratlanLs [1..20]
[1,3,5,7,9,11,13,15,17,19]
```

A korábbi `duplaz` függvény is megadható függvénykompozíciót alkalmazva:

```
duplaz_ :: (a -> a) -> a -> a
duplaz_ fg = fg . fg
```

```
> duplaz_ (+1) 10
12
```

A következő példában két könyvtárfüggvény kompozíciójával fogunk dolgozni, alkalmazzuk az `init` és `tail` könyvtárfüggvényeket.

4.29. feladat Írjunk egy Haskell-függvényt, amely levágja a paraméterként megadott lista első és utolsó elemét.

```
levag :: [a] -> [a]
levag = init . tail

> levag "gezakekazeg"
"ezakekaze"
```

4.30. feladat Írjunk egy Haskell-függvényt, amely a paraméterként megadott `1`-nél nagyobb természetes számokat tartalmazó listából kiválogatja az összetett számokat.

Az összetett számok kiválogatását az `osszetett` függvény végzi, a `not` és `primT` függvényeken függvénykompozíciót alkalmazva, ahol a `primT` függvényt korábban implementáltuk:

```
osszetett :: (Integral a) => [a] -> [a]
osszetett = filter (not . primT 3)

> osszetett [2..20]
[4,6,8,9,10,12,14,15,16,18,20]
```

4.11. A $ operátor

A **`$`** szimbólum használata megengedi, hogy a függvények kiértékelési sorrendjén változtassunk, illetve használatával fölöslegessé válik a zárójelezés. Az operátor jobbról asszociatív, azaz előbb a jobb oldalon levő kifejezés értékelődik ki. A `$` operátornak az operátorok között legkisebb a prioritása.

A következő példában előbb $\sqrt{2}$ kerül meghatározásra, amihez hozzáadódik `3`, majd `5`:

```
> sqrt 2 + 3 + 5
9.414213562373096
```

Zárójelek használatával azt érjük el, hogy előbb összeadódnak a számok, és csak ez után kerül alkalmazásra a négyzetgyök függvény, azaz az eredmény $\sqrt{10}$ lesz:

```
> sqrt (2 + 3 + 5)
3.1622776601683795
```

A `$` operátor használatával nem kell zárójeleket használni, az előző példához hasonlóan előbb összeadódnak az operátor jobb oldalán levő számok, ami után alkalmazásra kerül a négyzetgyök függvény, azaz most is a $\sqrt{10}$ lesz kiszámítva:

```
> sqrt $ 2 + 3 + 5
3.1622776601683795
```

A következő példában előbb az `abs` függvény kerül kiértékelésre, és csak utána az `sqrt`:

```
> sqrt $ abs (-16)
4.0
```

A következő kódsorban először a számok lesznek összeadva, a kapott összegre kiértékelődik az `abs`, és csak a legvégén lesz a négyzetgyök függvény alkalmazva:

```
> sqrt $ abs $ (-16) + 9
2.6457513110645907
```

A következő példában másképp használjuk a `$` operátort, ezért először az `abs` függvény kerül kiértékelésre, az eredményhez hozzáadódik a `9`, és erre az értékre lesz alkalmazva a négyzetgyök függvény:

```
> sqrt $ abs (-16) + 9
5.0
```

A `$` és a `.` szimbólumokat együtt is lehet alkalmazni. A következő példában előbb kerül meghívásra az `abs` függvény, és csak utána az `sqrt`:

```
> (sqrt . abs) (-16)
4.0
```

A következő példában ugyanazt érjük el, mint az előbb, de kevesebb zárójelet használtunk:

```
> sqrt . abs $ -16
4.0
```

A következő példában alkalmazni fogjuk a `Data.Char`-ban definiált `toUpper` függvényt, amely a paramétereként megadott karakterlánc első karakterét átcseréli nagybetűre. A példában előbb a `head` függvény kerül meghívásra, majd a `toUpper` függvény, és csak ezután a `:` operátor, amelynek segítségével egy új listát építünk úgy, hogy az első elemét nagybetűre cseréljük, és a többi betűt változatlanul hagyjuk:

```
import Data.Char
fugvNB :: [Char] -> [Char]
fugvNB ls = (toUpper . head) ls : tail ls

> fugvNB "deli-Karpatok"
"Deli-Karpatok"
```

A `words` és `unwords` függvények a standard könyvtárban vannak, gyakran fogjuk őket használni, éppen ezért egy pár példát mutatunk velük kapcsolatban, ahol a két függvény típusdeklarációja a következő:

```
words :: String -> [String]
unwords :: [String] -> String
```

A `words` a paraméterként megadott `String`-et szavakra bontja, azaz a bemeneti karakterláncot a szóközök, sortörések és tabulátorok mentén feldarabolja, míg az `unwords` a fordított műveletet végzi úgy, hogy az összefűzött szavak közé szóközöket tesz:

```
> ls = "paring fogaras kiralyko"
> words ls
["paring","fogaras","kiralyko"]

> unwords ["paring","fogaras","kiralyko"]
"paring fogaras kiralyko"
```

A következő lekérdezés a bemeneti karakterlánc kezdőbetűit átalakítja nagybetűkké, felhasználva a korábban megadott `fugvNB` függvényt.

```
> unwords $ map fugvNB $ words ls
"Paring Fogaras Kiralyko"
```

A fenti lekérdezésnek más formáit is megadhatjuk, például lambda kifejezések használatával a következőképpen módosulnak a függvénytörzsek, függvényhívások:

```
fugvM1 :: String -> String
fugvM1 ls =  unwords $ map aux $ words ls
```

```
    where
    aux = \ x -> (toUpper . head) x : tail x

fugvM2 :: String -> String
fugvM2 = unwords . map aux . words
    where
    aux = \ x -> (toUpper . head) x : tail x

> fugvM1 "retyezet kudzsiri bucsecs jezer"
"Retyezet Kudzsiri Bucsecs Jezer"

> fugvM2 "szebeni csernai vulkan"
"Szebeni Csernai Vulkan"
```

4.12. A kiértékelési stratégia

A funkcionális programozási nyelvek esetén kétféle kiértékelési stratégiát ismerünk: lusta (lazy) és mohó (eager).

A Haskell kiértékelési stratégiája, hasonlóan a Clean és a Miranda funkcionális programozási nyelvek stratégiájához, **lusta**, ami alatt a következőket értjük:

- egyetlenegy kifejezés sem értékelődik ki mindaddig, amíg a kifejezés által meghatározott értékre nincs szükség,
- legelőször **a legbaloldalibb, legkülső redex** (redukálható kifejezés) kerül kiértékelésre.

Ez a stratégia, ha létezik, mindig megtalálja a normál formát, azaz azt a végső formát, amely már nem redukálható tovább. Ez a stratégia teszi lehetővé, hogy a függvényeket kötetlen módon definiáljuk, ami azt jelenti, hogy egy függvény akkor is kiértékelhető, ha egyik argumentuma részlegesen definiált, és e miatt a stratégia miatt lesz lehetséges a végtelen adatszerkezetek kezelése is.

A lusta kiértékelési stratégia a mohó kiértékelési stratégiához képest kevésbé hatékony, azonban a kiértékelési stratégia hatékonyságát többféleképpen is lehet javítani. Például az azonos részkifejezéseket be lehet azonosítani, és amikor szükség van rájuk, az azonosítóval megjelölt eredményt használjuk.

A funkcionális nyelvek másik kiértékelési stratégiája a mohó (eager), ami a következőket jelenti:

– **a legbaloldalibb, legbelső redex**, azaz az argumentumok helyettesítése történik meg először.

A mohó kiértékelési stratégia esetében nem mindig ér véget a kiértékelési folyamat. Ilyen stratégiát alkalmazó funkcionális nyelvek a Lisp, SML stb.

A következőben egy példán keresztül mutatjuk be a két kiértékelési stratégia közötti különbséget, ahol az alkalmazott függvények legyenek a következők:

```
novel :: Num a => a -> a
novel x = x + 1

negyzet :: Num a => a -> a
negyzet x = x * x

negyzetNovel :: Num a => a -> a
negyzetNovel x = negyzet (novel x)
```

A főfüggvény, a `negyzetNovel` kiértékelésekor felvetődik a kérdés, hogy a `negyzet` vagy a `novel` függvények közül melyik értékelődik ki hamarabb. A lusta, a Haskell stratégiája a következőket teszi:

```
> negyzetNovel 6
   -> negyzet (novel 6)
   -> (novel 6) * (novel 6)
   -> (6 + 1) * (6 + 1)
   -> 7 * 7 -> 49
```

A mohó stratégia lépései pedig a következők lesznek:

```
> negyzetNovel 6
   -> negyzet (novel 6)
   -> negyzet (6 + 1)
   -> negyzet (7)
   -> 7 * 7 -> 49
```

4.13. Kiíratási műveletek

A tisztán funkcionális programozási nyelvek, így a Haskell is, csak bizonyos körülmények között engedik meg az olyan műveletvégzést, amely *mellékhatással* jár. Programírás során mellékhatást okoz, ha egy változó értékét módosítjuk, ha beolvasást, kiíratást vagy ha hibakezelést végzünk. Haskellben nem megengedett az `x = x + 1` értékmódosító művelet, mert

az `x` nem egy memóriarekeszt, nem egy regisztert jelent, ahova értéket lehet betenni, illetve amelyet módosítani lehet. Az `x = 10` azonban megengedett, de ez mást jelent, mint egy imperatív, például C/C++ vagy Java programozási nyelvben. Haskellben ez a művelet azt jelenti, hogy a `10`-es értéknek választunk egy nevet, és a továbbiakban ezzel a névvel fogjuk jelölni ezt az értéket.

A funkcionális paradigma lehetővé teszi, hogy a függvénykiértékelések során ne lépjenek fel mellékhatások, azonban ha olvasás, írás műveletet vagy hibakezelést akarunk, akkor nem lehet elkerülni a mellékhatásokkal járó műveleteket. Éppen ezért a funkcionális nyelvek különböző megoldásokat nyújtanak erre a problémára, és természetesen a Haskell is biztosítja ezeket. A későbbiekben ennek a témakörnek külön fejezetet szentelünk, de elöljáróban néhány egyszerű példán keresztül a Haskell kiíratási lehetőségeit mutatjuk be.

A következő függvény bemenete egy valós számokból álló lista, kimenete pedig egy kételemű tuple, ahol az első elem a `4.5` értéknél nagyobb vagy vele egyenlő számok átlagát, a másik pedig a `4.5`-nél kisebb számok átlagértéket jelöli.

```
atlagok :: (Ord a, Fractional a) => [a] -> (a, a)
atlagok ls = (auxAtl nLs1, auxAtl nLs2)
   where
   auxAtl  = \xLs -> sum xLs / fromIntegral (length xLs)
   nLs1 = filter (>= 4.5) ls
   nLs2 = filter (< 4.5) ls

> atlagok [6.5, 7.4, 8.9, 9.5, 3.5, 6.3, 4.2, 3.75]
(7.719999999999999,3.8166666666666664)
```

A következő `foAtlagok` függvényben a **`putStr`** és **`print`** függvények használatával az eredmények elé magyarázó szöveget írunk. Vigyázzunk, a `putStr`-nek csak `String` típusú bemenetet adhatunk. Ezeknek az író függvényeknek a használata azonban kizárólag egy **`do`** blokk keretén belül lehetséges. Ezt a jelölést használva a Haskell tudni fogja, hogy most nem függvénykiértékelést kell végezzen, hanem egymás alá, egymás után írt műveleteket kell végrehajtson, a megadott sorrendben úgy, ahogyan azt az imperatív paradigmán alapuló nyelvek esetében szokás. A `print` függvényben megadott `atl1` és `atl2` a `where` kulcsszó után megadott részben kapnak értéket, ahol a változatosság kedvéért elhagytuk a lambda függvénydefiníciót. Figyeljük meg, hogy a `where` kifejezésben a függvénykiértékelések a tiszta funkcionális paradigma szerint történnek.

```
foAtlagok ls = do
   putStr "az atmeno jegyek atlaga: "
   print atl1
   putStr "a nem atmeno jegyek atlaga: "
   print atl2
        where
        atl1 = sum nLs1 / fromIntegral (length nLs1)
        atl2 = sum nLs2 / fromIntegral (length nLs2)
        nLs1 = filter (>= 4.5) ls
        nLs2 = filter (< 4.5) ls

> foAtlagok [6.5, 7.4, 8.9, 9.5, 3.5, 6.3, 4.2, 3.75]
az atmeno jegyek atlaga: 7.719999999999999
a nem atmeno jegyek atlaga: 3.8166666666666664
```

A függvény szignatúráját elhagytuk, mert ebben az esetben az `foAtlagok` kimenetének típusa nem egy kételemű tuple típus lesz. Aki kíváncsi, megkérdezheti a Haskellt, mi a függvény szignatúrája, magyarázatra egy későbbi fejezetben kerül sor.

A prímszámokat és összetett számokat szétválogató, korábban megadott `valogatNr` függvényt hasonlóan meghívhatjuk egy `foValogat` függvényben, és az eredmények kiíratásakor magyarázó szöveget írhatunk.

```
foValogat ls = do
   putStr "a primszamok listaja: "
   print (fst rLs)
   putStr "az osszetett szamok listaja: "
   print (snd rLs)
        where
        rLs = valogatNr ls

> foValogat [24, 97, 5, 11, 74, 41, 61, 19, 100]
a primszamok listaja: [97,5,11,41,61,19]
az osszetett szamok listaja: [24,74,100]
```

4.31. feladat Írjunk egy Haskell-függvényt, amely egymás alá írja a bemeneti listában található számokat, illetve a számok négyzetgyökét.

```
negyzetgy ls = do
    mapM_ print gyLs
        where
        gyLs = [(i, sqrt i) | i <- ls]
```

```
> negyzetgy [17, 2, 9, 3, 10]
(17.0,4.123105625617661)
(2.0,1.4142135623730951)
(9.0,3.0)
(3.0,1.7320508075688772)
(10.0,3.1622776601683795)
```

A `negyzetgy` függvény kiíratja a `gyLs` listában meghatározásra kerülő számpárokat. A kiíratáshoz alkalmaztuk a **`mapM_`** függvényt, amelynek működése hasonló a korábban bemutatott `map`-hez. A `mapM_` is alkalmazza az első paraméterként megadott függvényt a második paraméterként megadott lista elemeire. A különbség az, hogy a `map` csak olyan függvényt kaphat paraméterként, amely függvénykiértékelést végez, azaz *tiszta* függvényt vár, míg a `mapM_`-nek olyan függvényt kell paraméterként megadni, amely *utasításokat* hajt végre, mint ahogyan a példában is, egy kiíratást végző függvény a paraméter.

4.14. Haskell-projektek

Nagyobb programok, projektek esetén a programot részekre kell bontani, ahol ezeket a részeket moduloknak hívjuk. Egy Haskell-projekt több állományból, azaz több modulból áll, ahol egy projekten belül a modulok függetlenek kell legyenek. Egy modul első sora a `module` kulcsszót, a modul nevét és a `where` kulcsszót tartalmazza. A modul nevét nagy kezdőbetűvel kell írni, az állomány neve pedig ugyanaz kell legyen, mint a modul neve. Az állomány következő soraiba kerülnek az importok és a kódsorok.

A következő példát egy Haskell-projektet készítve, három `hs` állományba szerkesztve fogjuk megoldani.

4.32. feladat Írjunk egy Haskell-függvényt, amely az alábbi számsorozatokban a számok első számjegye szerint, előfordulási statisztikát készít, azaz figyeljük meg, hogy fennáll-e Benford törvénye:

- az x^i számsorozatban, ahol `i = 1,...,n`,
- az első `n` szám faktoriálisának sorozatában,
- az `n` elemű Fibonacci-sorozatban.

Benford törvénye szerint több számsorozatban is fennáll, hogy a számok első számjegyei között az `1`-es számjegy előfordulásának esélye kb. `31%`, a `2`-es előfordulása kb. `19%`, és a százalékok a számjegyek növekedésével csökkennek.

Az első állományba megírjuk azokat a függvényeket, amelyek meghatározzák a kért sorozatokban található számok első számjegyeit. Legyen ez az állomány `Szamsorozatok.hs`, amelynek a következő lesz a tartalma:

```
module Szamsorozatok where
import Data.Char (digitToInt)

powL :: (Eq a, Num a) => Integer -> a -> [Int]
powL x n = auxPow x 1 n
   where
   auxPow x res n
       | n == 0 = []
       | otherwise = elsoSz res : auxPow x (x*res) (n-1)

faktL :: (Ord a, Num a) => a -> [Int]
faktL n
   | n < 0 = error "hibas bemenet"
   | otherwise = auxFakt 2 1 n
       where
       auxFakt i res n
           | n == 0 = []
           | otherwise = elsoSz res :
               auxFakt (i + 1) (i * res) (n - 1)

fibL :: (Ord a, Num a) => a -> [Int]
fibL n
   | n < 0 = error "hibas bemenet"
   | otherwise = 1 : auxFib 0 1 n
   where
   auxFib a b n
       | n == 1 = []
       | otherwise = elsoSz (a + b) :
           auxFib b (a + b) (n - 1)

elsoSz :: Integer -> Int
elsoSz = digitToInt . head . show
```

A `powL`, `faktL`, `fibL` függvények meghatározzák a megfelelő számsorozatokban szereplő számok első számjegyeinek listáját.

```
> powL 2 10
[1,2,4,8,1,3,6,1,2,5]
> faktL 10
[1,2,6,2,1,7,5,4,3,3]
```

```
> fibL 10
[1,1,2,3,5,8,1,2,3,5]
```

Egy szám első számjegyét az `elsoSz` függvény határozza meg. A függvénytörzsben függvénykompozíciót alkalmaztunk. Először a `show` függvénnyel átalakítottuk a számot `String` típusú értékké, majd a `head` függvénnyel meghatároztuk a `String` első karakterét, azaz az első számjegyet, végül ezt a `digitToInt`-el visszaalakítottuk számmá.

A `show` egy egyparaméteres függvény, amely paraméterét, amennyiben lehetséges, átalakítja `String` típussá. Hogy jobban megértsük, próbáljuk ki az alábbi lekérdezéseket:

```
> show 579
"579"

> lsFg x y = "A " ++ show x ++ " es " ++ show y ++
                 " osszege = " ++ show (x + y)
> lsFg 2 4
"Az 2 es 4 osszege = 6"

> show ('b', True)
"('b', True)"

> show [45, 3, 12, 101, 61]
"[45, 3, 12, 101, 61]"
```

Második állományunk legyen a `Benford.hs`, amelybe azokat a függvényeket tesszük, amelyek az első számjegyek szerinti statisztikai eredményeket határozzák meg:

```
module Benford where
import Data.List (sortBy)
import Data.Ord (comparing)

csoportosit :: Eq a => [a] -> [(a, Int)]
csoportosit [] = []
csoportosit ls = lsK : csoportosit lsVe
   where
   y = head ls
   lsK = (y, length [x | x <- ls, x == y])
   lsVe = [x | x <- ls, x /= y]

benford :: Ord a => [a] -> [(a, Int)]
benford ls = sortBy (comparing fst) $ csoportosit ls
```

A `csoportosit` függvény segítségével, amelynek bemeneti paramétere az első számjegyeket tartalmazó lista lesz, egy kételemű tuple elemtípusú listát hoztunk létre, ahol az első érték a számjegyet, a második a számjegy előfordulási számát jelöli.

A `benford` függvény a `csoportosit` által meghatározott listát rendezi, a könyvtárfüggvény `sortBy`-jal, amelynek első paramétere a rendezési kritériumot meghatározó logikai függvény. A tuple-elemek első értéke szerinti rendezés meghatározásához a `comparing`, illetve az `fst` függvényeket használtuk.

```
> csoportosit [1,2,4,8,1,3,6,1,2,5]
[(1,3),(2,2),(4,1),(8,1),(3,1),(6,1),(5,1)]

> benford [1,2,4,8,1,3,6,1,2,5]
[(1,3),(2,2),(3,1),(4,1),(5,1),(6,1),(8,1)]
```

A következő lekérdezés a `sortBy` használatát mutatja egy kételemű tuple elemtípusú lista esetében:

```
> :set +m
> sortBy (comparing fst) [(1,3),(2,2),(4,1),(8,1),(3,1),
>| (6,1),(5,1)]
[(1,3),(2,2),(3,1),(4,1),(5,1),(6,1),(8,1)]
```

A harmadik állomány a `Fo.hs` lesz, amelyben a kiíratást is elvégző főfüggvény, a `foBenford` található:

```
module Fo where
import Szamsorozatok ( powL, faktL, fibL )
import Benford ( benford, benford_ )

foBenford x n = mapM_ auxB $ zip bLs nLs
   where
   auxB (k1, k2) = do
       print k1
       print k2
       putStr "\n"

   ls = [fibL n, powL x n, faktL n]
   bLs = map benford ls
   nLs = map auxSz bLs
       where
       auxSz ls = [fromIntegral (snd x) / fromIntegral n
                  | x <- ls]
```

A `foBenford` kódsorban a `bLs` listában számszerűen adjuk meg a számjegyek előfordulási értékét, míg az `nLs` lista a százalékértékeket határozza meg. A `mapM_` paramétereként megadott `auxB` függvény a kiíratás formázásáért felelős, külön sorokba fogja írni a listaelemeket, és közéjük egy-egy új sor jelet tesz a `putStr "\n"`-el.

A számításokat, ha a következő sorozatokra akarjuk végezni:

- 5^i, ahol `i = 1,...,10000`,
- az első `10000` szám faktoriálisa,
- az első `10000` Fibonacci-szám,

akkor fordítsuk le a `Fo.hs`-t, majd paraméterezzük a következőképpen a főfüggvényt:

```
> foBenford 5 10000
[(1,3011),(2,1762),(3,1250),(4,968),(5,792),(6,668),...]
[0.3011,0.1762,0.125,9.68e-2,7.92e-2,6.68e-2,...]

[(1,3011),(2,1761),(3,1249),(4,969),(5,792),(6,669),...]
[0.3011,0.1761,0.1249,9.69e-2,7.92e-2,6.69e-2,...]

[(1,2956),(2,1789),(3,1276),(4,963),(5,794),(6,715),...]
[0.2956,0.1789,0.1276,9.63e-2,7.94e-2,7.15e-2,...]
```

A statisztikai eredmények meghatározása végett eljárhattunk volna úgy is, hogy előbb rendezzük, majd utána csoportosítjuk a számsorozatokat. Alkalmazva a `sort`, majd a `group` függvényeket, a következőképpen módosítható a `benford` függvény:

```
import Data.List (sort, group)

benford_ :: Ord a => [a] -> [(a, Int)]
benford_ ls = map aux rls
   where
   rls = group $ sort ls
   aux k = (head k, length k)

> benford_ $ fibL 10000
[(1,3011),(2,1762),(3,1250),(4,968),(5,792),(6,668),...]
```

A következőkben megnézzük, hogyan tudjuk az általunk írt függvényeket a Haskell-könyvtármodulokban szereplő függvényekkel együtt használni. Probléma akkor adódik, ha egy ugyanolyan nevű függvényt írunk, akár véletlenségből is, mint amilyen nevű szerepel abban a Haskell-könyvtármodulban, amit importálni szeretnénk.

Példaként nézzük meg a következő állományt, amelybe három függvényt írunk. Az `isDigit` eldönti egy karakterről, hogy számjegy-e vagy sem, a `fugv1` a paraméterként kapott karakterláncban meghatározza a számjegy-karaktereket és a számjegy-karakterek előfordulási pozícióit, míg a `fugv2` az angol ábécébeli betűk és Unicode kódértékekből épít egy listát, ahol az Unicode kódokat a `Data.Char`-ban található `ord` függvénnyel határoztuk meg.

```
import qualified Data.Char as DC

isDigit :: Char -> Bool
isDigit x = x >= '0' && x <= '9'

fugv1 :: [Char] -> [(Char, Int)]
fugv1 ls = aux ls 0
   where
   aux :: [Char] -> Int -> [(Char, Int)]
   aux [] i = []
   aux (k : ve) i
       | isDigit k = (k, i) : aux ve (i+1)
       | otherwise = aux ve (i+1)

fugv2 :: [Char] -> [(Char, Int)]
fugv2 [] = []
fugv2 (k : ve)
   | DC.isAlpha k = (k, DC.ord k) : fugv2 ve
   | otherwise = fugv2 ve

> fugv1 "Sapientia 2001 október 3 mvh"
[('2',10),('0',11),('0',12),('1',13),('3',23)],

> fugv2 "Sapientia 2001 október 3 mvh"
[('S',83),('a',97),('p',112),('i',105),('e',101),...
```

Vegyük észre, hogy a fenti kódsorokban a `Data.Char` importálása másképp történik, mint ahogyan a korábbi példákban mutattuk. Az `import qualified Data.Char as DC` kódsorral lehetőségünk lesz arra, hogy a `Data.Char` könyvtármodulhoz tartozó függvényekre az általunk választott `DC` névvel hivatkozhassunk. Ilyen módon a `DC.isAlpha` és `DC.ord` meghívások a `Data.Char` könyvtármodulban található függvények kiértékelését végzik. Az `isDigit` függvény esetében azonban az általunk megírt függvény kerül kiértékelésre. Ha azonban sor került volna egy `DC.isDigit`

függvényhívásra, akkor a `Data.Char` könyvtármodulban található `isDigit` került volna kiértékelésre.

A `Data.Char` könyvtármodul importálása a következőképpen is történhetett volna: `import Data.Char as DC`, azaz elhagyható `qualified`. Ebben az esetben azonban a saját `isDigit` függvényünk meghívása a `Main.isDigit` sorral oldható meg.

4.15. Kitűzött feladatok

4.1. feladat Írjunk egy Haskell-függvényt, amely meghatározza egy elsőfokú egyenlet gyökét.

4.2. feladat Írjunk egy-egy Haskell-függvényt, amely meghatározza a két paramétere közül a nagyobbat, illetve a kisebbet.

4.3. feladat Írjuk meg a `meghat :: ((a, b), c) -> b` Haskell-függvényt az `fst` és `snd` függvényeket használva, amely meghatározza egy értékpár első elemének második elemét, ahol az első elem szintén egy értékpár, például:

```
> meghat ((1,'n'), "Jutka")
'n'
> meghat ((1, 'f'), "Feri")
'f'
```

4.4. feladat Írjuk meg a `meghatM :: (Eq b) => b -> [((a, b), c)] -> [c]` Haskell-függvényt, amely egy értékpárokból álló lista esetében kiválogatja a második elemeket az értékpárokból, ha az első elem második eleme egy megadott értékkel egyenlő, ahol tehát az első elem is egy értékpár. Például a következő lekérdezés `'n'` szerint válogat:

```
> meghatM 'n' [((1,'n'),"Jutka"), ((2,'f'), "Zsolt"),
             ((3,'n'),"Mari"), ((4,'n'),"Zsuzsa")]
["Jutka", "Mari", "Zsuzsa"]
```

4.5. feladat Írjuk meg a `tEgyenlo :: Eq a => (a, a) -> (a, a) -> Bool` Haskell-függvényt, amely megvizsgálja, hogy a bemeneti paraméterei "majdnem" megegyeznek-e, azaz akkor határoz meg `True` értéket, ha a két tuple típusú bemenete ugyanazokat az értékeket tartalmazza, függetlenül az elemek sorrendjétől.

```
> tEgyenlo (6, 7) (6, 7)
True
> tEgyenlo (6, 7) (7, 6)
True
```

4.6. feladat Írjunk egy-egy egyparaméteres Haskell-függvényt, amely meghatározza egy tízes számrendszerbeli szám

- első számjegyét,
- számjegyeinek összegét,
- számjegyeinek számát,
- legnagyobb számjegyét,
- páros számjegyeinek számát.

4.7. feladat Írjunk egy `szamol` háromparaméteres Haskell-függvényt, amely meghatározza, hogy egy tízes számrendszerbeli szám (első paraméter) tetszőleges számrendszerbeli alakjában (második paraméter) hány olyan számjegy van, ami egyenlő a harmadik paraméterként megadott számmal.

A következő sorokban az első kiértékelés eredménye `3`, mert a `7673573` tízes számrendszerbeli alakjában három `7`-es szerepel, a második kiértékelés eredménye `1`, mert `1024` kettes számrendszerbeli alakjában egy `1`-es szerepel.

```
> szamol 7673573 10 7
3
> szamol 1024 2 1
1
```

4.8. feladat Írjuk meg az `alakit :: [Char] -> [Bool]` Haskell-függvényt, amely a bemeneti karakterlánc alapján létrehoz egy `Bool` elemtípusú listát úgy, hogy a nagybetűk helyére `True`-t, a kisbetűk helyre `False`-t tesz.

```
> alakit "saPienTia"
[False,False,True,False,False,False,True,False,False]
```

4.9. feladat Írjunk egy-egy Haskell-függvényt, amely meghatározza egy adott szám osztóinak számát, a szám legkisebb, illetve legnagyobb prímosztóját.

4.10. feladat Írjunk egy Haskell-függvényt, amely logaritmikus futásidejű algoritmust használva meghatározza az `n`-edik Fibonacci-szám utolsó számjegyét, ahol `n > 10000000`.

Az algoritmus azon az elgondoláson alapszik, hogy az `n`-edik Fibonacci-szám meghatározása visszavezethető a gyorshatványozás algoritmusára, azaz meg kell határozni az

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

`2x2`-es mátrix `n-1`-edik hatványértékét, és az `n`-edik Fibonacci-szám az eredménymátrix első sorának az első elemével lesz egyenlő. Például az ötödik Fibonacci-szám esetében a következő mátrixot kell meghatározni:

$$F_5 = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^4 = \begin{pmatrix} 5 & 3 \\ 3 & 2 \end{pmatrix}$$

4.11. feladat Írjunk egy Haskell-függvényt, amely a `map` függvényt alkalmazva meghatározza egy egész elemű lista minden elemére külön-külön:

- az első számjegyet,
- a számjegyek összegét,
- a számjegyek számát,
- a legnagyobb számjegyet,
- a páros számjegyek számát,
- a faktoriális értékeket.

4.12. feladat Írjunk egy Haskell-függvényt, amely meghatározza egy egész számokat tartalmazó listában a számjegyek számát.

4.13. feladat Írjunk egy-egy Haskell-függvényt, amely halmazműveleteket alkalmazva meghatározza

- az első `n` természetes szám négyzetét,
- az első `n` természetes szám köbét,
- az első `n` páros szám négyzetét,
- az első `n` olyan természetes számot, amelyek egyike se négyzetszám,
- egy szám prímosztóinak listáját,
- az első `n` darab páratlan, összetett számot,
- egy adott `n` értékre a következő szerkezetű listát:
  `[n,n-1,...2,1,1,2,...,n-1,n]`,
- egy adott `n` értékre a következő szerkezetű listát:
  `[(0,n),(1,n-1),(2,n-2),...,(n-2,2),(n-1,1),(n,0)]`.

5. fejezet

Haskell-listák

Haskellben az egyik legfontosabb adatszerkezet a lista típus, ezért külön fejezetben beszélünk erről a típusról. Korábban említettük, hogy a listák egyforma típusú elemek halmazát jelölik, ahol számít az elemek sorrendje, és egy elem többször is előfordulhat. Másfelől a lista *polimorf* típus, mert lehetőség van tetszőleges típusú, azaz `Int`, `Bool`, `String` stb. elemtípusú lista feldolgozására.

Jelölésükre, mint korábban láttuk, szögletes zárójeleket `[]`, a mintaillesztések során az összetett kifejezések jelölésére pedig kerek zárójeleket `()` használunk.

A különböző elemszámú listákra a következő jelölésekkel hivatkozhatunk, ahol természetesen más azonosítókat is használhatunk:

- `[]` – egy üres listát mintáz,
- `[[]]` – egy egyelemű listát mintáz, amelynek üres lista az eleme,
- `[x]` – egy egyelemű listát mintáz, ahol a listaelemet az `x` azonosító jelöli,
- `[x, y]` – egy kételemű listát mintáz, ahol `x` az egyik, `y` a másik listaelemet jelöli,
- `(k : ve)` – egy olyan listát mintáz, melynek első eleme `k`, a lista végét jelölő elem pedig a `ve`, ahol `k` elem típusú, `ve` lista típusú.

5.1. Operátorok listákon

Listákon az egyik leggyakrabban használt operátor a kettőspont, típusdeklarációja a következő:

```
(:) :: a -> [a] -> [a]
```

A fenti típusdeklarációból látható, hogy a kettőspont operátor két különböző típusú bemeneti értéket vár, az egyik egy tetszőleges típusú elem, a másik ugyanolyan típusú elemek listája, és a kimenet is egy lista típusú érték lesz.

A kettőspont operátort akkor használjuk, amikor egy új elemet szeretnénk beszúrni a lista elejére:

```
> ls1 = [1,2,3,4]
> 0 : ls1
[0, 1, 2, 3, 4]

> ls2 = "olozsvar"
> 'K' : ls2
"Kolozsvar"

> ls3 = [[1,2,3,4], [1,2,3], [1,2]]
> [5,6,7,8,9,10] : ls3
[[5,6,7,8,9,10], [1,2,3,4], [1,2,3], [1,2]]
```

Több elemet is hozzáfűzhetünk a lista elejére, ilyenkor minden hozzáfűzött elem esetében használnunk kell a kettőspont operátort:

```
> -1 : 0 : [1,2,3,4]
[-1,0,1,2,3,4]

> 'E' : 'M' : 'T' : 'E' : " Sapientia"
"EMTE Sapientia"
```

A kettőspont operátor használatával hozzárendelhetjük a lista első elemét egy azonosítóhoz, a lista többi elemét pedig egy másik nevű azonosítóhoz, de vegyük észre, hogy az első elemet jelölő `k` egy listabeli elemet azonosít, míg a lista végét jelölő `ve` egy részlistát azonosít, azaz a `k` és `ve` típusa nem ugyanaz.

```
> k : ve = [0,1,2,3,4]
> k
0
> ve
[1,2,3,4]

> :t k
k :: Num a =>  a
> :t ve
ve :: Num a =>  [a]
```

```
> k : ve = "Csikszereda"
> k
'C'
> ve
"sikszereda"

> :t k
k ::  Char
> :t ve
ve ::  [Char]
```

```
> k : ve = [[5,6,7,8,9,10], [1,2,3,4], [1,2,3], [1,2]]
> k
[5,6,7,8,9,10]
> ve
[1,2,3,4], [1,2,3], [1,2]]

> :t k
k :: Num a =>  [a]
> :t ve
ve :: Num a =>  [[a]]
```

A következő példákban a listákat megadó kifejezéseknek paramétereket adunk meg. Sor kerül a `show` és `length` függvények használatára:

```
> ls4 x = [2 ^ x, length (show (2 ^ x))]
> ls4 10
[1024, 4]
> ls4 20
[1048576,7]

> ls5 x = 3 ^ x : length (show (3 ^ x)): (ls4 x)
> ls5 10
[59049, 5, 1024, 4]
```

A következő példában a `..` és `++` operátorokat használjuk, a kis és nagybetűk listájának kigenerálására:

```
> ls6 = ['a'..'z']
> ls7 = ['A' ..'Z'] ++ ls5
> ls7
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"
```

A `!!` operátor kiválasztja a megadott sorszámú elemet a listából, ahol a sorszámozás nullától kezdődik.

```
> ls8 !! 2
'C'

> [9.5, 8.75, 10, 7.75] !! 5
*** Exception: Prelude.!!: index too large...
```

Az utolsó lekérdezésben a megadott sorszám értéke nagyobb volt, mint a lista utolsó elemének a sorszáma, ezért kivételt dobott a függvény.

A `++`, a `!!`, illetve további listákon alkalmazható operátorok használatakor számításba kell venni azt, ahogyan a Haskell a listákat kezeli. A

lista végéhez való hozzáfűzés, illetve egy elem kiválasztása nem konstans időben történik, mint ahogy a tömbök használata során a C/C++ vagy Java nyelvekben megszoktuk, hanem annyi egységnyi idő szükséges ehhez, ahány elemet fel kell dolgoznunk a lista első elemétől kezdve a lista végéig, vagy a !! operátor esetében amíg eljutunk a megfelelő sorszámú elemig. Felvetődik a kérdés, hogy akkor miért mégis nem a tömb, hanem a lista adatszerkezettel operálnak a funkcionális nyelvek? A válasz, hogy a lista adatszerkezet az, amelynek feldolgozása történhet rekurzívan, és éppen ezért lesz alkalmas arra, hogy a funkcionális paradigma szerinti kiértékelési folyamatot megvalósíthassuk vele. Másfelől a funkcionális paradigmának ellentmond a tömbök kezelésének módszere, például mellékhatást okoz az, amikor egy tömb elemét megváltoztatjuk.

5.2. Függvények listákon

Ebben a fejezetben megadjuk a leggyakrabban használt, listákat kezelő függvények implementációit.

head :: [a] -> a

A head függvény meghatározza egy lista első elemét. Nem alkalmazható üres listákra, pontosabban ebben az esetben hibaüzenet lesz az eredmény. A típusdeklaráció alapján láthatjuk, hogy bármilyen elemtípusú listára alkalmazható.

```
myHead :: [a] -> a
myHead [] = error "ures lista"
myHead (k : ve) = k

> myHead "Sepsiszentgyorgy"
'S'
```

tail :: [a] -> [a]

A tail függvény egy listát határoz meg, amelyben nem szerepel az eredeti lista első eleme. Bármilyen elemtípusú lista lehet a bemenete, viszont ez a függvény is hibaüzenetet ad, ha üres listára alkalmazzuk.

```
myTail :: [a] -> [a]
myTail [] = error "ures lista"
myTail (k : ve) = ve
```

```
> myTail ["Szent Anna-to", "Gyilkos-to", "Medve-to"]
["Gyilkos-to", "Medve-to"]
```

`init :: [a] -> [a]`

Az `init` függvény egy listát határoz meg, amelyben nem szerepel az eredeti lista utolsó eleme, hibaüzenetet akkor kapunk, ha üres listát adunk bemenetnek. Figyeljük meg, hogy úgy érjük el, hogy az utolsó elem ne kerüljön be az eredménylistába, hogy külön kezeljük azt az esetet, amikor a bemenet egy egyelemű lista, amely esetben a kimenet üres lista lesz.

```
myInit :: [a] -> [a]
myInit [] = error "ures lista"
myInit [k] = []
myInit (k : ve) = k : myInit ve

> myInit [1..10]
[1,2,3,4,5,6,7,8,9]
```

`last :: [a] -> a`

A `last` függvény meghatározza egy lista utolsó elemét. A függvénytörzsben üres lista esetében most is kivételt dob a függvény. Egyelemű bemeneti lista esetében a függvény kimenete ezzel az elemmel lesz egyenlő, ellenkező esetben a feldolgozást folytatjuk a lista végig, azaz végigmegyünk a listaelemeken, amíg egyelemű nem lesz a bemenet.

```
myLast :: [a] -> a
myLast [] = error "ures lista"
myLast [k] = k
myLast (k : ve) = myLast ve

> myLast [9.8, 3.55, 4.9]
4.9
```

`sum :: (Num a, Foldable t) => t a -> a`

A `sum` függvény összeadja a lista elemeit, ahol az üres lista elemeinek összege `0`. A beépített függvény típusdeklarációja szerint a bemeneti paraméter a `Foldable` típusosztályhoz kell tartozzon. A következő implementációkat, az alkalmazott mintaillesztések miatt, azonban csak úgy tudjuk lefordítani,

ha ezen változtatunk, és a típusdeklarációkban lista típusú bemenetekkel dolgozunk.

A továbbiakban a `sum` függvényt háromféleképpen is implementáljuk. Az első változat a korábban megírt implementációkhoz hasonló, a második változatban a lista első és utolsó elemének a jelölésére a `head` és `tail` könyvtárfüggvényeket használjuk, a harmadik változatban pedig algoritmikailag fogunk másképp eljárni.

```
mySum1 :: Num a => [a] -> a
mySum1 [] = 0
mySum1 (k : ve) = k + mySum1 ve

mySum2 :: Num a => [a] -> a
mySum2 [] = 0
mySum2 ls = k + mySum2 ve
   where
   k = head ls
   ve = tail ls
```

A függvények alkalmasak lesznek `Rational` típusú számok összeadására, mert a típusdeklarációkban az argumentumokra csak annyi megszorítást tettünk, hogy azok a `Num` típusosztályhoz tartozzanak.

```
> import Data.Ratio
> mySum1 [1 % 1, 1 % 2, 1 % 3, 1 % 4, 1 % 5, 1 % 6]
49 % 20
```

A harmadik verzióban az `auxSum` segédfüggvény fogja végezni a tulajdonképpeni számításokat. A részösszegeket akkor számoljuk, amikor *megyünk be a rekurzióba*, úgy, ahogyan korábbi feladatoknál is végeztük. Az `auxSum` két paraméteres, a második paramétere kezdetben `0`, majd a rekurzív hívások során az aktuális részösszeget jelöli. A rekurzió legalsó szintjén tehát ez a paraméter fogja jelölni a számok összegét, éppen ezért, amikor üres lista lesz az `auxSum` első paraméterének értéke, akkor ezzel az értékkel lesz egyenlő a függvényérték.

```
mySum3 :: Num a => [a] -> a
mySum3 ls = auxSum ls 0
   where
   auxSum [] res = res
   auxSum (k : ve) res = auxSum ve (k + res)
```

A `mySum3`-mal, ha komplex számokat akarunk összeadni, akkor a következő lekérdezéssel ezt megtehetjük:

```
> import Data.Complex
> mySum3 [3 :+ (-2.3), 3 :+ 2.1, 8.54 :+ 1.3]
14.54 :+ 1.1000000000000003
```

null :: Foldable t => t a -> Bool

A `null` függvény megvizsgálja hogy egy lista üres lista, vagy tartalmaz elemeket.

```
myNull :: [a] -> Bool
myNull [] = True
myNull (k : ve) = False

> myNull "Gyergyoszentmiklos"
False
> myNull []
True
```

map :: (a -> b) -> [a] -> [b]

A `map` függvény az első paraméterként megadott függvényt alkalmazza a második paraméterként megadott lista minden elemére. A következő lekérdezésben meghatározzuk `2` hatványait `0`-tól `10`-ig.

```
> map (2^) [0..10]
[1,2,4,8,16,32,64,128,256,512,1024]
```

A `map` függvény használatára korábban már több példát is láttunk, ezúttal a különböző implementációit adjuk meg. A függvénytörzs előtt megadott típusdeklaráció explicit módon jelzi, hogy az első paraméter függvény típusú.

```
myMap1 :: (a -> b) -> [a] -> [b]
myMap1 fg [] = []
myMap1 fg (k : ve) = fg k : myMap1 fg ve
```

A függvénytörzsben `fg`-vel jelöltük a függvény típusú paramétert, ezért az egyenlőség jobb oldalán az `fg k` azt jelenteni, hogy alkalmazzuk az `fg` függvényt a `k` bemeneten. Az így kapott értéket a `:` operátor alkalmazásával, a rekurzív függvényhívás eredményeként kapott lista elejére fűzzük.

A `null` függvény segítségével a `map` függvény egy másik implementációját is megadjuk. Algoritmikailag ez az implementáció megegyezik az előzővel, amiben eltér tőle, az a feltételek kezelésében áll, illetve hogy miként jelöljük a lista első elemét, illetve a lista végét.

```
myMap2 :: (a -> b) -> [a] -> [b]
myMap2 fg ls =
   if null ls then []
   else fg k : myMap2 fg ve
        where
        k = head ls
        ve = tail ls
```

A következő lekérdezés a paraméterként megadott karakterlánc összes betűjét nagybetűre cseréli. A második lekérdezés pedig meghatározza a paraméterként megadott karakterláncok hosszát.

```
> import Data.Char
> myMap1 toUpper "keleti KArpatok"
"KELETI KARPATOK"
> myMap2 length ["radnai", "gorgenyi", "besztercei"]
[6,8,10]
```

5.1. feladat Írjunk Haskell-függvényt, amely meghatározza $g^x \pmod p$ értékét minden `x = 1,2,..., p-1` értékre.

```
hatvSz :: (Integral a) => a -> a -> [a]
hatvSz g p = myMap2 (aux g p) [1..p - 1]
   where
   aux :: (Integral a) => a -> a -> a -> a
   aux g p x = mod (g ^ x) p

> hatvSz 2 11
[2,4,8,5,10,9,7,3,6,1]
```

`filter :: (a -> Bool) -> [a] -> [a]`

A `filter` függvény kiválasztja a lista azon elemeit, amelyek eleget tesznek egy adott feltételnek. A `filter` első paramétere egy olyan típusú függvény kell legyen, amelynek kimenete `True` vagy `False`, ami a tulajdonképpeni szűrési feltételt szolgáltatja.

Hasonlóan a `map`-hez, két implementációt adunk meg itt is. Az elsőben a korábbiakhoz hasonlóan mintaillesztést alkalmazunk a triviális és általános eset szétválasztása érdekében. Az általános esetben őrfeltételek segítségével vizsgáljuk, hogy az `fg k` függvényhívás `True` vagy `False` értéket ad. A `True` esetében a `:` operátort alkalmazva a `k` értékét az elé a lista elé fűzzük, amelyet a rekurzív függvényhívás eredményeként kapunk.

```
myFilter1 :: (a -> Bool) -> [a] -> [a]
myFilter1 fg [] = []
myFilter1 fg (k : ve)
    | fg k = k : myFilter1 fg ve
    | otherwise = myFilter1 fg ve
```

A lekérdezésekben a paraméterként megadott listából kiválasztjuk a páros számokat, majd a következőben a pozitívokat:

```
> myFilter1 even [1..10]
[2,4,6,8,10]
> myFilter1 (>0) [10,-5,3,-12,7]
[10,3,7]
```

A következő implementációban egy `let...in` blokkban elnevezzük a lista első elemét `k`-nak, a lista végét pedig `ve`-nek.

```
myFilter2 :: (a -> Bool) -> [a] -> [a]
myFilter2 fg ls =
    let
        k = head ls
        ve = tail ls
    in
        if null ls then []
        else
            if fg k then k : myFilter2 fg ve
            else myFilter2 fg ve
```

A következő lekérdezésekben az `isDigit`, `isUpper` könyvtárfüggvényeket használva kiválasztjuk a paraméterként megadott karakterlánc elemei közül a számjegyeket, majd a nagybetűket:

```
> import Data.Char
> myFilter2 isDigit "Maros folyo 749 km"
"749"
> myFilter2 isUpper "Erdelyi Karpat Egyesulet"
"EKE"
```

`reverse :: [a] -> [a]`

A `reverse` függvény megfordítja a lista elemeit. Itt is három implementációt adunk meg, de a korábbi függvényekkel ellentétben itt oda kell figyelni, hogy melyik implementációval dolgozunk, mert hatékonyság szempontjából nem lesz mindegy.

```
myReverse1 :: [a] -> [a]
myReverse1 [] = []
myReverse1 (k : ve) = myReverse1 ve ++ [k]

myReverse2 :: [a] -> [a]
myReverse2 ls =
   if null ls then []
   else myReverse2 (tail ls) ++ [head ls]

> myReverse1 "Temesvar"
"ravsemeT"
```

A fenti két implementáció csupán technikailag tér el egymástól, algoritmikailag mindkettő a ++ operátor segítségével oldja meg a lista aktuális első elemének az elköltöztetését, azaz ezt hozzáfűzi az újonnan épülő lista végéhez.

A harmadik implementációban, hogy hatékonyabb legyen a kódunk, másképp járunk el. Akkor végezzük az új lista elemeinek az egymás után való fűzését, amikor *megyünk be* a rekurzióba, ahol a tulajdonképpeni számításokat az `auxReverse` fogja végezni. Nem használjuk a ++ operátort, helyette a : operátort alkalmazzuk, aminek segítségével mindig az épülő lista elejére tesszük az elemet. Az `auxReverse` függvénynek két paramétere lesz, ahol a második paraméterében előállítjuk a megfordított listát, ezért a triviális esetben ezzel az értékkel lesz egyenlő a függvényérték. Kezdetben a `auxReverse` második paramétere üres listával inicializálódik, majd a lista első elmével folytatja, utána a lista második eleme bekerül az első elé, és így tovább.

```
myReverse3 :: [a] -> [a]
myReverse3 ls = auxReverse ls []
   where
   auxReverse [] res = res
   auxReverse (k : ve) res = auxReverse ve (k : res)

> myReverse3 [[1,2], [3,4,5], [6,7,8]]
[[6,7,8],[3,4,5],[1,2]]
```

A jobb megértés végett módosíthatjuk úgy a fenti kódsort, hogy az épülő listát minden egyes rekurzív hívás előtt kiíratjuk. Ennek érdekében a `myReverseIr` függvényben az `auxReverse` függvényt átírjuk úgy, hogy az általános esetben egy `do` blokk keretén belül két műveletsort adunk meg. Az elsőnek az lesz a szerepe, hogy a `print` függvényt alkalmazva kiíratást

végezzen, a másodikban pedig rekurzív függvényhívásra kerül sor. A triviális esetben is másképp kell eljárni, itt a `return` segítségével jelezzük, hogy mi lesz a függvény kimeneti értéke.

Az olyan függvények működésére, amelyek tulajdonképpen nem függvénykiértékelést végeznek, hanem egymás után megadott műveletek sorozatát hajtják végre, egy későbbi fejezetben még visszatérünk.

```
myReverseIr ls = auxReverse ls []
   where
   auxReverse [] res = return res
   auxReverse (k: ve) res = do
       print (k : res)
       auxReverse ve (k : res)

> myReverseIr "Deva"
"D"
"eD"
"veD"
"aveD"
"aveD"
```

A fenti két függvény típusdeklarációját sem adtuk meg, mert a függvények visszatérési értékének típusáról is a későbbiekben adunk magyarázatot.

`take :: Int -> [a] -> [a]`

A `take` függvény meghatározza a második argumentumaként megadott lista első `n` elemét, ahol `n` a függvény első argumentuma. A következő lekérdezések példákon keresztül mutatják be a függvény működését.

```
> take 3 ["Zilah", "Arad", "Des", "Nagyvarad", "Torda"]
["Zilah","Arad","Des"]
> take 10 ["Zilah", "Arad", "Des", "Nagyvarad", "Torda"]
["Zilah", "Arad", "Des", "Nagyvarad", "Torda"]

> take 5 [0..]
[0,1,2,3,4]
> take 10 [ 1/i | i <- [1..]]
[1.0,0.5,0.3333333333333333,0.25,0.2,0.16666666666666666,
0.14285714285714285,0.125,0.1111111111111111,0.1]
```

A második lekérdezésből látható, hogy nem eredményez futási hibát, ha a `take` első paramétere nagyobb, mint a lista elemszáma. Az utolsó két lekérdezésnél a természetes számok végtelen listája lesz a `take` második

paramétere, amelyet a lusta kiértékelési stratégia miatt a Haskell kezelni tud, a végeredmény meghatározásához ugyanis nincs szükség a második paraméter teljes kiértékelésére.

A függvény implementációja pedig a következő:

```
myTake :: Int -> [a] -> [a]
myTake n [] = []
myTake n (k : ve)
   | n == 0 = []
   | otherwise = k : myTake (n - 1) ve

> myTake 50 "Szekelyudvarhely"
"Szekelyudvarhely"
```

takeWhile :: (a -> Bool) -> [a] -> [a]

A `takeWhile` függvény meghatározza a második argumentumaként megadott lista azon prefixét, amelyben az elemek eleget tesznek egy feltételnek, a feltételt a `takeWhile` első paramétereként adjuk meg. A feltételként megadott függvény kimenete `Bool` típusú kell legyen.

A következő lekérdezés után azt a listát kapjuk, amelyben benne lesznek az eredeti lista azon első elemei, amelyek párosak. Az első páratlan szám és az utána következő elemek már nem kerülnek be az eredmény listába, mert a keresés leáll.

```
> takeWhile even [2,4,6,8,9,10,12,14]
[2,4,6,8]
```

Figyeljük meg, hogy a `filter` implementációhoz képest hogyan módosul az `otherwise` ág.

```
myTakeWhile :: (a -> Bool) -> [a] -> [a]
myTakeWhile fg [] = []
myTakeWhile fg (k : ve)
   | fg k = k : myTakeWhile fg ve
   | otherwise = []

> myTakeWhile (>0) [1,2,3,-5,4,6,7]
[1,2,3]

> import Data.Char
> myTakeWhile isDigit "1568 - Torda, januar 13"
"1568"
```

A következő lekérdezésben meghatározzuk annak a listának az elemszámát, amelyet úgy kapunk, hogy a listába az $\frac{1}{2^i}$ képlettel addig teszünk be új elemeket, amíg 0-t nem kapunk. A 0 értéket a valós számokon alkalmazott kerekítés miatt fogjuk elérni.

```
> length $ myTakeWhile ( /= 0) [1 / (2 ^ i) | i <-[1..]]
1023
```

drop :: Int -> [a] -> [a]

A drop függvény kitörli a második argumentumaként megadott lista első n elemét, ahol n a függvény első argumentuma. Tulajdonképpen a korábban ismertetett take függvény párja. A következő lekérdezés során a bemeneti lista első három elemét töröljük.

```
> drop 3 ["Zilah", "Arad", "Des", "Nagyvarad", "Torda"]
["Nagyvarad", "Torda"]
```

Hasonlóan a take függvényhez, ha az n értéke nagyobb, mint a bemeneti lista elemszáma, akkor a kiértékelés eredménye üres lista lesz, ahogyan ez az alábbi implementációból és a lekérdezésből is látható:

```
myDrop :: Int -> [a] -> [a]
myDrop n [] = []
myDrop n (k : ve)
   | n == 0 = k : ve
   | otherwise = myDrop (n-1) ve

> myDrop 50 "Szekelyudvarhely"
""
```

dropWhile :: (a -> Bool) -> [a] -> [a]

A dropWhile a takeWhile függvény párja, kitörli a második argumentumaként megadott lista azon prefixét, amelyben az elemek eleget tesznek a feltételnek, ahol a feltételt egy Bool értéket meghatározó függvényként kell megadni. Az alábbi lekérdezés eredménye egy olyan lista, amelyben az eredeti lista elejéről töröltük a páros elemeket, ahol az even könyvtárfüggvényt használjuk annak érdekében, hogy eldöntsük egy számról, hogy az páros vagy sem.

```
> dropWhile even [2,4,6,8,9,10,12,14]
[9,10,12,14]
```

Az implementációban figyeljük meg, hogy az `otherwise` ágon nincs rekurzív függvényhívás, hiszen a lista további elemeinek a vizsgálatára már nincs szükség, ebben az esetben a kimenet egyenlő lesz a fennmaradó listaelemekkel, amelyek elé beszúrjuk a `k`-t.

```
myDropWhile :: (a -> Bool) -> [a] -> [a]
myDropWhile fg [] = []
myDropWhile fg (k : ve)
    | fg k = myDropWhile fg ve
    | otherwise = k : ve
```

`elem ::(Eq a, Foldable t) => a -> t a -> Bool`

Az `elem` függvény megvizsgálja, hogy egy adott elem szerepel-e a listaelemek között, ahol a függvény első paramétere a vizsgálandó elem, második paramétere pedig a lista, amiben keresünk. Az első lekérdezésben keressük az `a` betűt a megadott karakterlánc betűi között, a másodikban pedig az `A` betűt keressük. A függvény kimenete `True` vagy `False` aszerint, hogy sikeres volt vagy sem a keresés.

```
> elem 'e' "Nagyszeben"
True
> elem 100 [2, 4, 6, 8, 10]
False
```

Az `elem` függvény implementálásánál, azért, hogy mintaillesztéssel tudjuk kezelni az üres lista esetet, ahogy korábban is tettük, újból eltérünk a könyvtárfüggvény típusdeklarációjától.

```
myElem :: (Eq a) => a -> [a] -> Bool
myElem x [] = False
myElem x (k : ve)
    | x == k = True
    | otherwise = myElem x ve
```

Vegyük észre, hogy annak a megállapítása, hogy egy elem *nem* szerepel a listaelemek között, csak akkor lehetséges, ha megvizsgáltuk az összes listaelemet. A triviális esetben ekkor tudjuk a *nem* választ megadni, azaz ekkor lesz a függvény kimeneti értéke `False`. Az első találat esetében azonban leáll a keresés, és a függvényérték `True` lesz. Megállapíthatjuk, hogy a *nem* válasz meghatározása költségesebb, azaz időigényesebb, mint az *igen* válasz megállapítása.

A következő lekérdezés, felhasználva a `maganhangzo`, `myElem` és `myDropWhile` függvényeket, kitörli a bemeneti lista elejéről a magánhangzókat.

```
maganhangzo :: Char -> Bool
maganhangzo c = myElem c "aeiouAEIOU"

> myDropWhile (not . maganhangzo) "Brasso"
"asso"
```

5.2. feladat Írjunk egy Haskell-függvényt, amely meghatározza a bemeneti karakterlánc *madárnyelv* változatát, ahol egy karakterlánc madárnyelv változata azt jelenti, hogy minden m magánhangzót kicserélünk mpm-re

Első körben az `intercalate` függvény használatát mutatjuk be, mert a megoldás során ezt alkalmazni fogjuk:

```
intercalate :: [a] -> [[a]] -> [a]
```

A függvény a `Data.List` könyvtárban van, és két paramétert vár, ahol az első egy elválasztójel szerepet betöltő lista típusú érték, a második pedig egy listákból álló lista. A függvény egyetlenegy listát hoz létre, amelyben a megadott elválasztójeleket beszúrja a második paraméterként megadott listák közé.

```
> import Data.List (intercalate)
> intercalate [0,0] [[1,2,3], [4,5], [6,7,8,9]]
[1,2,3,0,0,4,5,0,0,6,7,8,9]

> ls = ["fenyokut","tozeglap","korond"]
> intercalate "-" ls
"fenyokut-tozeglap-korond"
```

Az `auxMadarNy` függvény abban az esetben, ha a bemeneti karakter magánhangzó, elvégzi az előírt cserét, ahol a karakter tesztelését a korábban megírt `maganhangzo` függvénnyel végezzük. Ahhoz, hogy egy karaktereket tartalmazó lista minden magánhangzóját átcseréljük, nincs más dolgunk, mint hogy a `map`-nek megadjuk paraméterként az `auxMadarNy` függvényt és a megfelelő karakterláncot.

```
auxMadarNy :: Char -> String
auxMadarNy k =
   if maganhangzo k then [k] ++ "p" ++ [k]
   else [k]
```

```
> map auxMadarNy "Lucs-tozeglap"
["L","upu","c","s","-","t","opo","z","epe","g","l",
                                              "apa","p"]
```

A végső, madárnyelv formára hozott karakterlánc meghatározásához a `madarNy` főfüggvényben az `intercalate` függvényt fogjuk alkalmazni, segítségével elválasztójelek nélkül egymás után fűzzük a kapott részeket.

```
madarNy :: String -> String
madarNy ls = intercalate "" $ map auxMadarNy ls

> madarNy "Lucs-tozeglap"
"Lupucs-topozepeglapap"
```

`zip :: [a] -> [b] -> [(a, b)]`

A `zip` függvény a bemeneti két lista alapján egy elempárokból álló listát hoz létre. Az új lista elemszámát a rövidebb lista elemszáma határozza meg. Figyeljük meg azt is, hogy a bemeneti két lista típusa nem kell megegyezzen.

```
myZip :: [a] -> [b] -> [(a, b)]
myZip [] ls = []
myZip ls [] = []
myZip (k1 : ve1) (k2 : ve2) = (k1, k2) : myZip ve1 ve2

> myZip [1..6] "mohos-tozeglap"
[(1,'m'),(2,'o'),(3,'h'),(4,'o'),(5,'s'),(6,'-') ]

> myZip ["mohos", "fenyokut", "lucs"] [1,2,3,5,6,7]
[("mohos",1),("fenyokut",2),("lucs",3)]
```

5.3. feladat Írjunk egy Haskell-függvényt, amely egy kételemű tuple elemtípusú lista esetében maximum értékeket számol a második elem szerepét betöltő listaelemeken úgy, hogy eredménynek létrehoz egy kételemű tuple listát, ahol az első elem az eredeti lista első eleme lesz, a második pedig a kiszámolt maximum érték.

```
lsSz = [("mari", [10, 6, 5.5, 8]),
        ("feri", [8.5, 9.5]),
        ("zsuzsa", [4.5, 7.9, 10]),
        ("levi", [8.5, 9.5, 10, 7.5])]

maxTu ls = mapM_ print $ zip nLs maxLs
   where
```

```
    nLs = [k1 | (k1, k2) <- ls]
    jLs = [k2 | (k1, k2) <- ls]
    maxLs = map maximum jLs

> maxTu lsSz
("mari",10.0)
("feri",9.5)
("zsuzsa",10.0)
("levi",10.0)
```

A `maxTu` függvényben az `nLs` a tuple-elemek első értékeiből, azaz a nevekből létrehozott listát jelöli. A `jLs`-t a tuple-ök második elemeiből, azaz a jegyekből álló listákból építjük fel. A maximum meghatározásához a könyvtárfüggvény `maximum`-ot használjuk, és azért, hogy ezt minden jegy-listára meghatározzuk, a `map` függvény paramétereként hívjuk meg. A `zip` függvényt arra használjuk, hogy a nevekből képezett listát összekapcsoljuk, *összezipzárazzuk* a maximumértékekkel. Az így kapott kételemű tuple lista egy elemét a `print` segítségével íratjuk ki, és azért, hogy ez minden listaelemre megtörténjen, a `mapM_`-nek paraméterként adjuk meg.

5.4. feladat Írjunk egy Haskell-függvényt, amely meghatározza egy lista elemei közül a legnagyobbat és a legnagyobb elem listabeli pozícióit.

A `myMaximum1` függvény egy elempárokból álló listát határoz meg, ahol az elempár első eleme a maximum értéket, a második eleme pedig a sorszámot fogja jelenteni.

A kódsorban a `zip` függvénnyel összezipzárazzuk az eredeti lista és a `[0, 1, ...]` sorszámokat jelölő lista elemeit, így egy tuple elemtípusú listát kapunk. A tuple elemtípusú listából a `filter` függvénnyel válogatjuk ki azokat az elemeket, amelyek első eleme megegyezik a legnagyobb elemmel, ehhez az `fst` függvényt használjuk. A legnagyobb elemet a `maximum` könyvtárfüggvénnyel állapítjuk meg.

```
myMaximum1 :: (Num b, Enum b, Ord a) => [a] -> [(a, b)]
myMaximum1 ls = filter fg $ zip ls [0, 1..]
    where
    m = maximum ls
    fg k = fst k == m

> myMaximum1 [3, 5, 6, 10, 3, 10, 8, 7, 6, 10]
[(10,3),(10,5),(10,9)]
```

A `myMaximum2` függvény az előző egy módosított változata, amelyben a `map` és az `snd` alkalmazásával az eredményt más formában adjuk meg, ez egy kételemű tuple lesz, ahol az első elem a maximumot, a második elem pedig a maximum elem pozícióit jelöli.

```
myMaximum2 :: (Num b, Enum b, Ord a) => [a] -> (a, [b])
myMaximum2 ls = (m, map snd $ filter fg $ zip ls [0,1..])
    where
    m = maximum ls
    fg k = fst k == m

> myMaximum2 [3, 5, 6, 10, 3, 10, 8, 7, 6, 10]
(10,[3,5,9])
```

`splitAt :: :: Int -> [a] -> ([a], [a])`

A `splitAt`, a megadott indexérték alapján, a bemeneti listát két listára osztja. Az eredmény egy kételemű tuple, ahol a tuple-elemek lista típusúak lesznek. A függvénytörzsben a bemeneti lista felosztására a `take` és `drop` könyvtárfüggvényeket használjuk.

```
mySplitAt :: Int -> [a] -> ([a], [a])
mySplitAt n ls = (ls1, ls2)
    where
    ls1 = take n ls
    ls2 = drop n ls

> mySplitAt 4 [1,2,3,4,5,6,7,8,9]
([1,2,3,4],[5,6,7,8,9])
> mySplitAt 6 "Almasi-barlang"
("Almasi","-barlang")
```

A következő, listákat kezelő függvények implementációját már nem adjuk meg, csak használatukat mutatjuk be.

`notElem :: (Foldable t, Eq a) => a -> t a -> Bool`

A `notElem` visszatérési értéke `True`, ha az első paraméterként megadott elem nincs benne a második paraméterként megadott listában. Ellenkező esetben `False` lesz a függvény kimenete.

```
> notElem 'O' "Maros-volgyi fatorzsbarlangok"
True
> notElem 'o' "Maros-volgyi fatorzsbarlangok"
False
```

concat :: Foldable t => t [a] -> [a]

A `concat` függvénynek egy olyan bemenetet kell megadni, amely hozzátartozik a `Foldable` típusosztályhoz, sajátos esetben egy olyan listát vár, amelynek elemei szintén lista típusúak, eredményként pedig egyetlenegy listát épít a bemeneti listákból.

```
> concat ["torjai", " Budos", "-barlang"]
"torjai Budos-barlang"
> concat [[2, 4, 6], [1, 3, 5, 7, 9]]
[2,4,6,1,3,5,7,9]
```

repeat :: a -> [a]

A `repeat` függvény használata egy végtelen számítási sorozat elindítását jelenti, azaz a paraméterként megadott elemmel egy végtelen listát fog generálni, ezért ajánlott a `take` vagy a `takeWhile` függvényekkel együtt használni.

```
> take 5 (repeat "barlang")
["barlang","barlang","barlang","barlang","barlang"]
```

replicate :: Int -> a -> [a]

A `replicate` a paraméterként megadott elemet `n`-szer fűzi be egy listába.

```
> replicate 5 "barlang"
["barlang","barlang","barlang","barlang","barlang"]
```

cycle :: [a] -> [a]

A `cycle` is egy végtelen számítási folyamatot indít el, ezért ajánlott a `take` vagy a `takeWhile` függvényekkel együtt használni. A paraméterként megadott lista elemeit végtelenszer fűzi egymás után.

```
> take 10 $ cycle "barlang"
"barlangbar"
```

iterate :: (a -> a) -> a -> [a]

Az `iterate` az első paraméterként megadott függvényt alkalmazza egy kezdeti értéken, ahol a kezdeti érték a második paramétere. Amikor önmagát hívja, akkor a meghatározott függvényértéket használja kezdeti értékként. Az `iterate` is egy végtelen iterációt jelent, ezért általában a `take` vagy a `takeWhile` függvényekkel együtt szokták alkalmazni.

```
> take 10 (iterate (\ x -> 2 * x ) 1)
[1,2,4,8,16,32,64,128,256,512]
```

any :: Foldable t => (a -> Bool) -> t a -> Bool

Az `any` függvény megvizsgálja, hogy a megadott feltételt teljesíti-e **valamelyik** listabeli elem. A feltételt mint logikai függvény, paraméterként kell megadni.

```
> any isUpper "Rozsnyoi-barlang"
True
```

all :: Foldable t => (a -> Bool) -> t a -> Bool

Az `all` megvizsgálja, hogy a megadott feltételt teljesíti-e **minden** listabeli elem.

```
> all isUpper "Rozsnyoi-barlang"
False
```

5.3. A @ minta

A **@** mintát mintaillesztések során használjuk, amikor a függvénytörzsben szeretnénk mind a lista egészére, mind a lista első elemére vagy a lista végére hivatkozni.

5.5. feladat Írjunk egy Haskell-függvényt, amely összefésüli két rendezett lista elemeit.

```
merge :: (Ord a) => [a] -> [a] -> [a]
merge [] ls = ls
merge ls [] = ls
merge ls1@(k1 : ve1) ls2@(k2 : ve2)
    | k1 < k2 = k1 : merge ve1 ls2
    | otherwise = k2 : merge ls1 ve2

> merge [1, 5, 10, 15] [2, 4, 6, 8, 20, 22, 24]
[1,2,4,5,6,8,10,15,20,22,24]
```

A `merge` függvény két triviális esetet kezel, az első azt figyeli, amikor az első lista üres lesz, a második pedig azt az esetet kezeli, amikor a második lista lesz üres. A harmadik feltétel megadásakor a **@** mintát használtuk, hogy az

egyenlőség jobb oldalán egyaránt tudjunk hivatkozni a listák egészére, az `ls1`-re vagy az `ls2`-re, illetve a listák első elemére, a `k1`-re vagy `k2`-re és a listák végére, a `ve1`-re vagy a `ve2`-re.

5.6. feladat Írjunk egy Haskell-függvényt, amely két lista esetében, ahol az elemek növekvő sorrendben vannak, megvizsgálja, hogy van-e közös elemük.

```
eVizsgal :: (Ord a) => [a] -> [a] -> Bool
eVizsgal ls [] = False
eVizsgal [] ls = False
eVizsgal ls1@(k1: ve1) ls2@(k2: ve2)
    | k1 == k2 = True
    | k1 < k2 = eVizsgal ve1 ls2
    | otherwise = eVizsgal ls1 ve2

> eVizsgal [1,3,5,7,9] [2,4,6,8,11]
False
> eVizsgal [1,3,5,7,9] [2,4,7,8,11]
True
```

A fenti kódsor esetében is kényelmes hivatkozásokat tesz lehetővé a **@** minta alkalmazása. Algoritmikailag a függvény működése azon alapszik, hogy `True` lesz a függvény kimeneti értéke az első közös elem megtalálásakor, amely esetben a keresést nem is kell tovább folytatni. Ahhoz, hogy megállapítsuk, hogy a két bemeneti listának nincs közös eleme, annyi elem megvizsgálására van szükség, ahány elemből a rövidebbik lista áll. Ezért az `eVizsgal` függvény kimenetét csak a triviális esetekben lehet `False`-ra állítani.

5.4. Rendezési algoritmusok

Ebben a fejezetben három rendezési algoritmust mutatunk be: a beszúró rendezést, a gyorsrendezést és az összefésülő rendezést.

A **beszúró rendezést** (insertion sort) két függvénnyel valósítjuk meg, az első az `ins` függvény, amely beszúr egy elemet egy rendezett listába, a második az `insertionS`, amely az `ins` függvényt alkalmazva meghatározza a rendezett sorrendet. A bemeneti lista típusa tetszőleges, annyi a megkötés, hogy az `Ord` típusosztályhoz kell tartozzanak az elemek.

```
ins :: (Ord a) => a -> [a] -> [a]
ins x [] = [x]
```

```
ins x (k : ve)
   | x > k = k : ins x ve
   | otherwise = x : (k : ve)

> ins 'h' "abcdeklmno"
"abcdehklmno"
> ins "rozsnyoi" ["almasi", "budos", "sugo"]
["almasi","budos","rozsnyoi","sugo"]
```

Az alábbi `insertionS` kódsorából nagyon könnyedén leolvasható, hogy úgy rendezzük a lista elemeit, hogy az első elemet, a `k`-t beszúrjuk a rendezett `ve` listába, ahol a `ve` lista rendezését rekurzívan az `insertionS` függvénnyel végezzük.

```
insertionS :: (Ord a) => [a] -> [a]
insertionS [] = []
insertionS (k : ve) = ins k (insertionS ve)

> insertionS [3,12,7,8,4,5,11]
[3,4,5,7,8,11,12]
> insertionS ["rozsnyoi", "almasi", "budos", "sugo"]
["almasi","budos","rozsnyoi","sugo"]
```

A **gyorsrendezést** (quick sort) halmazkifejezésekkel végezzük. Kiválogatjuk a `k`-nál kisebb elemeket a `kLs`-be, illetve a `k`-nál nagyobb elemeket az `nLs`-be. Mindkét részlistát rendezzük rekurzívan a `quickS` függvénnyel, majd a kapott részlistákat összefűzzük, a két rendezett lista közé helyezve a `k` elemet.

```
quickS :: (Ord a) => [a] -> [a]
quickS [] = []
quickS (k : ve) = quickS kLs ++ [k] ++ quickS nLs
   where
   kLs = [x | x <- ve, x < k]
   nLs = [x | x <- ve, x >= k]

> quickS [7,5,9,10,13,2,3,11]
[2,3,5,7,9,10,11,13]
```

Az **összefésülő rendezést** (merge sort) is két függvénnyel oldjuk meg. Az első függvény, a `merge` két rendezett lista összefésülését végzi, amelyet korábban adtunk meg.

A második függvény a `mergeS`, a `merge` függvény segítségével rendezi a paraméterként megadott lista elemeit. Az eredeti listát itt is két részlistára

osztjuk, a `bLs` fogja tartalmazni a bemeneti lista első felét, a `jLs` pedig a második felét. Mindkét részlistát rekurzívan, a `mergeSort` függvénnyel rendezzük, majd a korábban megírt `merge` függvénnyel összefésüljük a két rendezett listát. Az eredeti lista felosztását a `take` és `drop` könyvtárfüggvényekkel végezzük.

```
mergeSort :: (Ord a) => [a] -> [a]
mergeSort [] = []
mergeSort [k] = [k]
mergeSort ls = merge bLs jLs
   where
   db = div (length ls) 2
   bLs = mergeSort (take db ls)
   jLs = mergeSort (drop db ls)

> mergeSort [3,12,6,7,5,9,10,2,10,1]
[1,2,3,5,6,7,9,10,10,12]
```

5.5. Hajtogatások

Korábban több magasabb rendű függvény is bemutatásra került, mint például a `map` és a `filter`, amelyek az explicit rekurziót helyettesítve, egy lista elemeinek a feldolgozását végezték, oly módon, hogy a kimeneti értékük típusa lista típus volt. Ebben a fejezetben további magasabb rendű függvényeket mutatunk be, pontosabban a *hajtogatásokat*, a *folding* műveleteket végző függvényekre térünk ki. Ezek a függvények, hasonlóan a korábban bemutatott magasabb rendű függvényekhez, listaelemeken végeznek kiértékeléseket explicit rekurzió nélkül.

A `foldl` függvény háromparaméteres, az első egy bináris operátor, a második egy kezdőérték, a harmadik pedig egy lista. Típusdeklarációja a következő:

```
foldl :: Foldable t => (a -> b -> a) -> a -> t b -> a
```

A függvény balról jobbra haladva, rendre feldolgozza a listaelemeket, ezért azt mondjuk, hogy balra asszociatív. A függvény rendre alkalmazza a bináris operátort úgy, hogy bal oldali operandusa a `foldl` függvény második paramétere, *jobb oldali operandusa az aktuális listaelem* lesz. A második paraméter minden listaelem feldolgozása után felülíródik a bináris művelet eredményével. A függvény által meghatározott érték a második argumentumban kiszámolt érték lesz, amely értékkel a triviális esetben lesz egyenlő

a függvény kimenete. Egyszerűen fogalmazva a `foldl` akkor számol, amikor *megy be* a rekurzióba, ezért a rekurzió legalsó szintjén már meg van határozva a végső eredmény.

A következő lekérdezésben a `foldl` alkalmazásával meghatározzuk a paraméterként megadott lista elemeinek összegét, az alkalmazott bináris operátor a `+`, és a helyes számítás végett a kezdőértéket `0`-ra állítottuk.

```
> foldl (+) 0 [1..10]
55
```

A `foldl` függvény implementációja segíthet jobban megérteni a függvény működését. A mintaillesztés alkalmazása miatt most is módosítottuk a típusdeklarációt, a harmadik paraméter típusát lista típusra cseréltük.

```
myFoldl :: (a -> b -> a) -> a -> [b] -> a
myFoldl op res [] = res
myFoldl op res (k : ve) = myFoldl op (op res k) ve
```

A következő kiértékelés esetében nem csak az eredményt tüntetjük fel, hanem a számítási sorozatot is:

```
> myFoldl (-) (-2) [34,6,12,65,8,11,23]
-161

((((((-2 - 34) - 6) - 12) - 65) - 8) - 11) - 23)
```

Ha a listaelemeket k_1, k_2, k_3, ..., k_n-nel jelöljük, és a `res` a kezdőérték, akkor általánosan a `foldl` hívásai során a következőképpen kerülnek sorra a műveletek, ahol az operátort infix formába írtuk:

$$(\ldots\ (((res\ op\ k_1)\ op\ k_2)\ op\ k_3)\ \ldots\ op\ k_n).$$

A `foldl` működésének megértéséhez úgy módosítjuk a `myFoldl` függvény kódsorát, hogy kiíratjuk a részeredményeket. Ennek érdekében módosul a triviális eset, az általános esetben pedig egy `do` blokk keretén belül fogjuk megadni azon műveletsorokat, amelyeket a függvény el kell végezzen.

```
myFoldlIr op res [] = return res
myFoldlIr op res (k : ve) = do
   putStr $ show (op res k) ++ " "
   temp <- myFoldlIr op (op res k) ve
   return temp
```

A `do` blokk keretén belül az `<-` operátort használva lehetőség van arra, hogy a `myFoldlIr` függvény által meghatározott értéket lekérjük. Figyeljük meg, hogy az `<-` operátort most másképp használjuk, mint ahogyan azt a

halmazkifejezések esetében tettük. A Haskellnek ez azonban nem okoz fennakadást, a kontextusból meg tudja állapítani, hogyan járjon el. Egy `do` blokk keretén belül az `<-` operátor használatáról elöljáróban annyit szeretnénk mondani, hogy a Haskellben így kell lekérni egy olyan függvény kimeneti értékét, amely *nem tiszta* függvény. Haskellben a *nem tiszta* függvények kiértékelések mellett utasításokat, mellékhatásokkal járó műveleteket is végrehajtanak.

A következő lekérdezések egy lista elemeinek összegét, szorzatát, illetve maximum elemét határozzák meg, ahol figyeljük meg a kiíratásra kerülő részeredményeket, és próbáljuk ki a függvényt más bemenetekre is:

```
> myFoldlIr (+) 0 [1..10]
1 3 6 10 15 21 28 36 45 55 55

> myFoldlIr (*) 1 [2,4..10]
2 8 48 384 3840 3840

> ls = [9,6,12,65,8,11,23]
> myFoldlIr max (head ls) ls
9 9 12 65 65 65 65 65
```

A `foldr` a `foldl` függvény párja, a listaelemeket azonban másképp dolgozza fel. Ez a függvény is egy bináris operátort alkalmaz egy kezdeti érték és a megfelelő listaelemek között. A függvény első paramétere a bináris operátor, második paramétere pedig a kezdeti érték lesz, ahol a listaelemeket a harmadik paraméterként megadott listából veszi. Típusdeklarációja a következő:

```
foldr :: Foldable t => (b -> a -> a) -> a -> t b -> a
```

A `foldr` a listaelemeket jobbról balra dolgozza fel, ezért azt mondjuk, hogy a függvény jobbról asszociatív. A kezdőérték csak a legutolsó rekurzív híváskor kerül feldolgozásra. A részértékek akkor kerülnek meghatározásra, amikor a függvény *jön vissza* a rekurzióból, így a végső kifejezés meghatározására csak akkor kerülhet sor, amikor minden rekurzív függvényhívás kiértékelődött. A bináris operátor *bal oldali operandusa az aktuális listaelem*, jobb oldali operandusa pedig a `foldr` függvény második paramétere lesz.

A következő lekérdezésekben a `foldr` függvény alkalmazásával meghatározzuk a listaelemek szorzatát:

```
> foldr (*) 1 [2,4..10]
3840
```

A `foldr` implementációjában a harmadik paraméter típusát most is lista típusra állítottuk, a függvénytörzsben alkalmazott mintaillesztés miatt.

```
myFoldr :: (b -> a -> a) -> a -> [b] -> a
myFoldr op res [] = res
myFoldr op res (k : ve) = op k $ myFoldr op res ve
```

A következő függvényhívás első ránézésre lehet, hogy meglepő eredményt ad, éppen ezért itt is érdemes végiggondolni a számítási sorozatot:

```
> myFoldr (-) (-2) [34,6,12,65,8,11,23]
 -3

 34 - (6 - (12 - (65 - (8 - (11 - (23 - (-2)))))))
```

Ha a listaelemeket k_1, k_2, k_3, ..., k_n-nel jelöljük, és a `res` a kezdőérték, akkor általános esetben a `foldr` a következőképpen értékeli ki a műveleteket:

$$k_1 \ op \ (k_2 \ op \ (k_3 \ op \ (\ldots \ (k_n \ op \ res) \ \ldots))).$$

Az összehasonlítás végett ideírjuk még egyszer a `foldl` függvény által végzett műveletsort:

$$(\ldots \ (((res \ op \ k_1) \ op \ k_2) \ op \ k_3) \ \ldots \ op \ k_n).$$

A következőkben hasonlóan járunk el, mint ahogy a `foldl` függvény esetében is tettük. A jobb megértés végett megadjuk a `foldr`-nek azt a változatát, amelyben kiíratjuk a részeredményeket:

```
myFoldrIr op res [] = return res
myFoldrIr op res (k : ve) = do
   temp <- myFoldrIr op res ve
   putStr $ show temp ++ " "
   return $ op k temp
```

A listaelemek összegének, szorzatának, illetve maximum elemének a meghatározására vonatkozó lekérdezések és az eredmények a következők lesznek, amelyeket hasonlítsunk össze azoknak a lekérdezéseknek az eredményeivel, amelyeket a `myFoldlIr`-nél kaptunk.

```
> myFoldrIr (+) 0 [1..10]
0 10 19 27 34 40 45 49 52 54 55

> myFoldrIr (*) 1 [2,4..10]
1 10 80 480 1920 3840
```

```
> ls = [9,6,12,65,8,11,23]
> myFoldrIr max (head ls) ls
9 23 23 23 65 65 65 65
```

Visszatérve a `foldl` függvényhez, egy fontos észrevételt kell vele kapcsolatban tegyünk. A lusta kiértékelési stratégia miatt a `foldl` hatékonysága közel sem elfogadható, ezt leginkább úgy tudjuk letesztelni, ha egy nagy elemszámú lista elemeinek összegét határozzuk meg. Egy `1` millió elemszámú lista elemei összegének a meghatározása több másodpercig is eltarthat:

```
> :set +s
> foldl (+) 0 [1..10000000]
50000005000000
(3.15 secs, 1,612,359,432 bytes)
```

Valamivel jobb időt kapunk, ha a `foldr` változattal dolgozunk:

```
> foldr (+) 0 [1..10000000]
50000005000000
(2.32 secs, 1,615,360,800 bytes)
```

A fenti hatékonysági problémák kiküszöbölésére a Haskell több `foldl` implementációt vezet be, amelyek esetében a kiértékelési stratégián is változtat, ezekben az esetekben szigorú (strict) kiértékelési stratégiát tesz lehetővé. Ilyen függvények a `foldl'`, `foldl1`, `foldl1'` ahol a `foldl'`, illetve `foldl1'` függvények a `Data.List` könyvtárcsomagban találhatók. Egy másik különbség, hogy a `foldl1`, illetve `foldl1'` változatok esetében nem kell megadni kezdőértéket, ez mindig a lista első eleme lesz. Összehasonlításképpen próbáljuk ki és mérjük le az időket a következő meghívások esetében is:

```
> import Data.List(foldl', foldl1')

> foldl' (+) 0 [1..10000000]
50000005000000
(0.23 secs, 880,062,880 bytes)

> foldl1 (+) [1..10000000]
50000005000000
(2.75 secs, 1,612,359,336 bytes)

> foldl1' (+) [1..10000000]
50000005000000
(0.43 secs, 880,059,088 bytes)
```

Következésképpen a `foldl` helyett mindig a `foldl'`, vagy amikor lehetséges a `foldl1'` változatokat használjuk.

5.7. feladat Írjunk egy Haskell-függvényt, amely a `foldl'`-t alkalmazva meghatározza a paraméterként megadott lista legnagyobb elemét, illetve a legnagyobb elem előfordulási pozícióit.

A feladatot korábban is megoldottuk, de az eredmény meghatározásához többször is bejártuk a bemeneti listát, a következő algoritmusokban egyszer fogunk végigmenni a listaelemeken.

```
import Data.List (foldl')
myMaximum :: (Num b, Ord a) => [a] -> (a, [b])
myMaximum ls = (max, mLs)
    where
    (mLs, _, max) = foldl' op res ls
    res = ([], 0, head ls)
    op tRes k
        | k == m = (p : pLs, p + 1, m)
        | k < m = (pLs, p + 1, m)
        | k > m = ([p], p + 1, k)
            where
            (pLs, p, m) = tRes
```

```
> myMaximum [3, 5, 6, 10, 3, 10, 7, 6, 10, 4, -10, 10]
(10,[11,8,5,3])
```

A `myMaximum`-nak egyetlen bemenete van, a feldolgozandó lista, kimenete pedig egy kételemű tuple típusú érték, ahol a tuple első eleme a maximum elem, míg a második a pozíciók listája lesz. Ezt a két értéket a `foldl'` függvény határozza meg. Figyeljük meg, hogy a `foldl'` függvény második paraméterének típusa egy háromelemű tuple, tehát a meghívásra kerülő bináris operátor két operandusa egy háromelemű tuple, illetve az aktuális listaelem lesz. A háromelemű tuple elemei rendre a következő értékeket jelölik: a maximum elem pozíciói, az aktuális pozíció, illetve az aktuális maximum elem.

A következő `myMaximum_` függvény is a legnagyobb elemet, illetve a legnagyobb elem pozícióit határozza meg, csak explicit rekurziót használ. A a kódsor segíthet az előző megértésében.

```
myMaximum_ :: (Num b, Ord a) => [a] -> (a, [b])
myMaximum_ ls = (max, mLs)
    where
```

```
    (mLs, _, max) = myMaxAux ([], 0, head ls) ls
    myMaxAux tRes [] = tRes
    myMaxAux tRes (k : ve)
        | k == m = myMaxAux (p : pLs, p + 1, m) ve
        | k < m = myMaxAux (pLs, p + 1, m) ve
        | k > m = myMaxAux ([p], p + 1, k) ve
            where
            (pLs, p, m) = tRes
```

A következőkben a hajtógató függvények segítségével számos, korábban már megadott függvény implementációját mutatjuk be. Egy lista elemeinek összegét a következő kódsorok adják:

```
mySumL :: (Foldable t, Num a) => t a -> a
mySumL = foldl (\ res k -> res + k) 0

mySumR :: (Foldable t, Num a) => t a -> a
mySumR = foldr (\ k res -> res + k) 0
```

```
> mySumL [1,2,3,4,5]
15
```

A `head` és `last` függvények implementációját kétféleképpen adjuk meg. A `myHead1` esetében a `foldr`-t használjuk, a `myLast1`-nál pedig a `foldl`-t, és mindkét esetben bevezetjük az `undefined` konstanst:

```
myHead1 :: [a] -> a
myHead1 = foldr (\ k res -> k) undefined

myLast1 :: [a] -> a
myLast1 = foldl (\ res k -> k) undefined
```

A lusta kiértékelési stratégia miatt az `undefined` egyik kódsorban sem kerül kiértékelésre a helyes eredmény meghatározásához, erre azonban nincs is szükség. Az `undefined` használata viszont azért szükséges, mert konkrét érték megadása esetében korlátoztuk volna a bemeneti lista típusát.

Vegyük észre, hogy sokkal kényelmesebb lenne, ha nem kellene kezdőértéket megadni, így a következő függvényekben a `foldr1`, illetve `foldl1'` függvényeket fogjuk használni:

```
myHead2 :: [a] -> a
myHead2 = foldr1 (\ k res -> k)
```

```
myLast2 :: [a] -> a
myLast2 = foldl1' (\ res k -> k)

> myHead2 "Lohavasi-vizeses"
'L'

> myLast2 [("lohavasi", 80), ("durrogo", 25)]
("durrogo", 25)
```

A következő példákban a `map` egy-egy implementációját adjuk meg, ahol figyeljük meg, hogy a `foldl'`-vel megadott kód kevésbé hatékony a `++` operátor alkalmazása miatt.

```
myMapL :: Foldable t => (a -> b) -> t a -> [b]
myMapL fg = foldl' (\ resLs k -> resLs ++ [fg k]) []

myMapR :: Foldable t => (a -> b) -> t a -> [b]
myMapR fg = foldr (\ k resLs -> fg k : resLs) []

> myMapL (\ x -> x * x) [1..10]
[1,4,9,16,25,36,49,64,81,100]
```

Hasonlóan megadhatjuk a `filter` egy-egy implementációját, ahol a `foldr`-el megadott kód itt is hatékonyabb a `:` operátor alkalmazása miatt.

```
myFilterL :: Foldable t => (a -> Bool) -> t a -> [a]
myFilterL fg = foldl' op []
   where
   op resLs k = if fg k then resLs ++ [k] else resLs

myFilterR :: Foldable t => (a -> Bool) -> t a -> [a]
myFilterR fg = foldr op []
   where
   op k resLs = if fg k then k : resLs else resLs
```

A következő lekérdezésekben a fenti függvényeket alkalmazzuk, az elsőnél meghatározzuk a listaelemek közül azokat, amelyek nem oszthatók hárommal, a második esetben pedig a megadott karakterláncból meghatározzuk azokat a betűket, amelyek nem magánhangzók.

```
> myFilterL (\ x -> mod x 3 /= 0) [1..10]
[1,2,4,5,7,8,10]
> myFilterR (`notElem` "aeiuo") "csurgoko"
"csrgk"
```

Az `elem` implementációját is kétféleképpen adjuk meg. Az egyikben a `foldl'`, a másikban a `foldr` függvényt fogjuk használni, ahol a bináris operátor egy `Bool` típusú értéket határoz meg. Figyeljük meg, hogy a két implementáció csak az `op` paraméterezési sorrendjében tér el egymástól:

```
myElemL :: (Foldable t, Eq a) => a -> t a -> Bool
myElemL x = foldl' op False
   where
    op resB k = (x == k) || resB

myElemR :: (Foldable t, Eq a) => a -> t a -> Bool
myElemR x = foldr op False
   where
    op k resB = (x == k) || resB

> myElemL 'o' "havasrekettyei vizeses"
False
> myElemR 'a' "havasrekettyei vizeses"
True
```

Az `any` függvény implementációját is megadjuk a `foldl`, illetve a `foldr` függvények segítségével.

```
myAnyL :: Foldable t => (a -> Bool) -> t a -> Bool
myAnyL fg = foldl' op False
   where
   op resB k = fg k || resB

myAnyR :: Foldable t => (a -> Bool) -> t a -> Bool
myAnyR fg = foldr op False
   where
   op k resB = fg k || resB
```

A következő lekérdezések, illetve a lekérdezéseket követő feladat azt fogják bemutatni, hogy a `foldr` függvény, a `foldl`, illetve `foldl'` függvényekkel szemben, bizonyos operátorokkal együtt használva alkalmazható végtelen listákon.

Az első két lekérdezésben a könyvtárfüggvény `any` kerül alkalmazásra, ez is kezeli a végtelen listaszerkezeteket. Abban az esetben, ha a bemeneti listaelemek között talál egy olyat, amelyre teljesül a feltétel, akkor `True` kimenettel le tud állni a kiértékelési folyamat, ellenkező esetben viszont nem. A következő három lekérdezésben a `foldl`,

illetve a `foldr` függvényekkel implementált változatokat használjuk.

```
> any even [1, 4 ..]
True
> any even [1, 3 ..]
...
```

```
> myAnyL even [1,4..]
...
> myAnyR even [1, 4 ..]
True
> myAnyR even [1, 3 ..]
...
```

A magyarázat előtt figyeljük meg a korábban megadott `myFoldl`, illetve `myFoldr` függvénytörzsek második sorait:

```
myFoldl op res (k : ve) = myFoldl op (op res k) ve
myFoldr op res (k : ve) = op k $ myFoldr op res ve
```

A `myAnyL` a `myFoldl` implementációja szerint *a második argumentumában* rendre meghatározza a következő kifejezések értékét:

```
even 1 || False -> False
even 4 || False -> True
even 7 || True -> True
...
```

A kiértékelési folyamat pedig akkor érhetne véget, ha a bemeneti lista minden elemét fel tudnánk dolgozni, de ez a végtelen lista bemenet miatt nem fog soha bekövetkezni.

A `myAnyR` a `myFoldr` implementációja alapján pedig a következő kifejezéseket értékeli ki:

```
even 1 || myFoldr op res ve
even 4 || myFoldr op res ve
```

Az `even 4` kifejezés kimenete `True`, ezért a `||` operátornak nem lesz szüksége arra, hogy meghatározza a második argumentumának is az értékét, mert az eredmény így is úgy is `True` lesz. Így a második lépésben a kiértékelési folyamat leáll. Ha azonban a `myAnyR`-ben a következőképpen definiáltuk volna az `op`-t szintén végtelen kiértékelési folyamatot generáltunk volna:

```
myAnyR :: Foldable t => (a -> Bool) -> t a -> Bool
myAnyR fg = foldr op False
   where
   op k resB = resB || fg k
```

5.8. feladat Írjunk egy Haskell-függvényt, amely megvizsgálja, hogy egy adott szám prímszám-e vagy sem, majd alkalmazzuk a függvényt a prímszámok listájának kigenerálására. Az implementáció során alkalmazzuk a `foldr` függvényt.

```
primTeszt_ :: Integer -> Bool
primTeszt_ n = n > 1 && foldr op True [3,5..]
   where
   op k resB = k * k > n || (mod n k /= 0 && resB)

> primTeszt_ 1789
True
```

A `foldr` harmadik paramétere a páratlan számok végtelen listája lesz, de ez nem eredményez végtelen számítási folyamatot, mert az `op`-ben a `||` operátort használtuk. Amikor a `k * k > n` feltétel eredménye `True` lesz, nincs már szükség a `||` jobb oldalán álló kifejezés kiértékelésére, az eredmény meghatározható. Ezután pedig elkezdődik a `k-2, k-4` stb. értékekkel (visszafelé haladva a páratlan számok listájában) való oszthatóságok vizsgálata.

A prímszámok listáját pedig a következőképpen adhatjuk meg, ahol a hatékonyság miatt további módosítást végeztünk a `primTeszt` függvényben:

```
primTeszt :: Integer -> Bool
primTeszt n = n > 1 && foldr op True primLista
   where
   op k resB = k * k > n || (mod n k /= 0 && resB)

primLista :: [Integer]
primLista = 2 : filter primTeszt [3,5..]

> take 10 primLista
[2,3,5,7,11,13,17,19,23,29]
> last $ take 10000 primLista
104729
(1.61 secs, 253,024,968 bytes)
```

Ha a `foldl'` függvény alkalmazásával szeretnénk egy hasonló kódsort írni, akkor az egy végtelen kiértékelési folyamatot eredményezne.

A `scanl` és `scanr` függvények a `foldl`, illetve `foldr` függvények általánosításai, olyan értelemben, hogy az általuk kiszámolt eredmények a `foldl`,

`foldr` számításait fogják tartalmazni különböző részlistákon. Típusdeklarációikból jól látható, hogy három bemeneti paramétert kell mindkét függvény esetében megadni, ahol az első paraméter egy bináris operátor, a második egy tetszőleges típusú érték, amit kezdőértékként kezel a függvény, és a harmadik egy lista típusú adat, aminek az elemeit fogja a függvény feldolgozni.

```
scanl :: (b -> a -> b) -> b -> [a] -> [b]
scanr :: (a -> b -> b) -> b -> [a] -> [b]
```

Természetesen használható a `scanl'` változat is, ami a szigorú kiértékelési stratégiára váltva hatékonyabbá teszi a függvénykiértékelést. Használatához a `Data.List` könyvtárcsomagot kell importálni. Ugyanúgy használható a `scanl1` függvény is, ami megengedi a függvény használatát kezdőérték nélkül. A hatékonyság miatt természetesen a `scanl'` változatot ajánlják.

A részlisták a `scanl` esetében az `inits` függvény által meghatározott listák lesznek, a `scanr` esetében pedig a `tails` határozza meg a részlistákat. Éppen ezért egy kis kitérőt teszünk, és megadjuk az `inits` és `tails` implementációit.

Az `inits` meghatározza a bemeneti lista kezdőszeleteiből létrehozható listák listáját, típusdeklarációja a következő:

```
inits :: [a] -> [[a]]
```

Előbb megadjuk a `preInits` függvény kódsorát, amely paraméterként egy elemet és egy listákból álló listát kap, ahol az elemet beszúrja minden egyes lista elejére.

```
preInits :: a -> [[a]] -> [[a]]
preInits k = map (k : )

> preInits 2 [[], [3], [3,4]]
[[2],[2,3],[2,3,4]]
```

A fenti elgondoláson alapszik az `myInits` implementációja:

```
myInits :: [a] -> [[a]]
myInits [] = [[]]
myInits (k : ve) = [] : map (k : ) (myInits ve)

> myInits "kofa"
["","k","ko","kof","kofa"]
```

A `tails` meghatározza a bemeneti lista alapján a listavégek listáját, típusdeklarációja a következő:

```
tails :: [a] -> [[a]]
```

A `tails` implementációját is mintaillesztéssel adjuk meg, ahol a második minta lesz a triviális eset, az első pedig az általános esetet tartalmazza. Ez utóbbinál **@**-mintát használtunk, hogy az egyenlőség jobb oldalán a teljes listára (`ls`) és annak a végére (`ve`) is lehessen hivatkozni. Vigyázzunk, a két minta sorrendjét nem szabad felcserélni, fordításkor figyelmeztetést, majd futtatáskor helytelen eredményt kapunk.

```
myTails :: [a] -> [[a]]
myTails ls@(_ : ve) = ls : myTails ve
myTails _ = [[]]

> myTails "kofa"
["kofa","ofa","fa","a",""]
```

A következő lekérdezésekben a `scanl'`-t alkalmazva meghatározzuk a bemeneti `[1..4]` lista minden kezdeti szegmensén az elemek összegét, illetve az elemek szorzatát, ahol feltüntetjük a számításokat is:

```
> import Data.List
> scanl' (+) 0 [1..4]
[0,1,3,6,10]
[0, 0+1, (0+1)+2, ((0+1)+2)+3, (((0+1)+2)+3)+4]

> scanl' (*) 1 [1..4]
[1,1,2,6,24]
[1, 1*1, (1*1)*2, ((1*1)*2)*3, (((1*1)*2)*3)*4]
```

A következő lekérdezés esetében az eredmény mellett feltüntetjük a számításokat:

```
> scanl' (\ x y -> (x + y) / 2) 2 [1,2,3]
[2.0,1.5,1.75,2.375]
foldl' (\ x y -> (x+y)/2) 2 [] -> 2 -> 2.0
foldl' (\ x y -> (x+y)/2) 2 [1] -> (2+1)/2 -> 1.5
foldl' (\ x y -> (x+y)/2) 2 [1,2] -> ((2+1)/2+2)/2 -> 1.75
foldl' (\ x y -> (x+y)/2) 2 [1,2,3] -> (((2+1)/2+2)/2+3)/2 -> 2.375
```

A `scanr` alkalmazása a fenti példákon a következőket eredményezi, ahol feltüntetjük a számításokat is:

```
> scanr (+) 0 [1..4]
[10,9,7,4,0]
[(1+(2+(3+(4+0)))), (2+(3+(4+0))), (3+(4+0)), (4+0), 0]
```

```
> scanr (*) 1 [1..4]
[24,24,12,4,1]
[(1*(2*(3*(4*1)))), (2*(3*(4*1))), (3*(4*1)), (4*1), 1]

> scanr (\ x y -> (x + y) / 2) 2 [1,2,3]
[1.625,2.25,2.5,2.0]
foldr (\ x y -> (x+y)/2) 2 [1,2,3] -> (((2+3)/2+2)/2+1)/2 -> 1.625
foldr (\ x y -> (x+y)/2) 2 [2,3] -> ((2+3)/2+2)/2 -> 2.25
foldr (\ x y -> (x+y)/2) 2 [3] -> (2+3)/2 -> 2.5
foldr (\ x y -> (x+y)/2) 2 [] -> 2 -> 2.0
```

A továbbiakban megadjuk mindkét függvény kétféle implementációját. Az első változatban a `foldl'`, illetve `foldr` függvények kerülnek meghívásra egy-egy `map` keretén belül. A `scanl'` esetében a bemeneti lista kezdőszeletein dolgozunk, amelyeket az `inits` állít elő, a `scanr` esetében a végeken dolgozunk, amelyeket a `tails` állít elő.

```
myScanl_ :: (b -> a -> b) -> b -> [a] -> [b]
myScanl_ op x ls = map (foldl' op x) $ inits ls

myScanr_ :: (a -> b -> b) -> b -> [a] -> [b]
myScanr_ op x ls = map (foldr op x) $ tails ls
```

A másik módszer szerinti implementációk a könyvtármodulban található változatokat követik, explicit rekurziót használnak:

```
myScanl :: (b -> a -> b) -> b -> [a] -> [b]
myScanl op x [] = [x]
myScanl op x (k : ve) =  x : myScanl op (op x k) ve

myScanr :: (a -> b -> b) -> b -> [a] -> [b]
myScanr op x [] = [x]
myScanr op x (k : ve) = op k (head temp): temp
   where
   temp = myScanr op x ve
```

A `scanl` függvénnyel különböző számsorozatok előállítását lehet nagyon kompakt formában megadni, ahogyan ezt a következő példák szemléltetik, ahol a hatékonyság miatt természetesen a `scanl'` változatot fogjuk használni.

5.9. feladat Írjunk egy Haskell-függvényt, amely a `scanl'` függvényt használva kigenerálja egy listába az első `n` Fibonacci-számot.

```
import Data.List (scanl')
fibonacci :: [Integer]
fibonacci = 1 : scanl' (+) 1 fibonacci

fibonacciLs :: Int -> [Integer]
fibonacciLs n = take n fibonacci

> fibonacciLs 10
[1,1,2,3,5,8,13,21,34,55]
```

Figyeljük meg, hogy a `fibonacci` függvény meghívása `take` nélkül végtelen kiértékelési folyamathoz vezetne.

A kódsor jobb megértése végett próbáljuk ki a következő meghívásokat:

```
> scanl' (+) 1 []
> scanl' (+) 1 [1]
> scanl' (+) 1 [1,1]
> scanl' (+) 1 [1,1,2]
> scanl' (+) 1 [1,1,2,3]

> map (foldl' (+) 1) $ inits []
> map (foldl' (+) 1) $ inits [1]
> map (foldl' (+) 1) $ inits [1,1]
> map (foldl' (+) 1) $ inits [1,1,2]
> map (foldl' (+) 1) $ inits [1,1,2,3]
...
```

5.10. feladat Írjunk egy Haskell-függvényt, amely a `scanl'` függvényt használva kigenerálja az első `n` háromszögszámot egy listába, ahol háromszögszámoknak nevezzük azon számsorozat elemeit, ahol az elemek felírhatók az első valahány természetes szám összegeként.

Az első `5` háromszögszám a következő:

```
1  = 1 + 0
3  = 2 + 1
6  = 3 + 2 + 1
10 = 4 + 3 + 2 + 1
15 = 5 + 4 + 3 + 2 + 1
```

```
haromszogSz :: [Integer]
haromszogSz = scanl' (+) 1 [2..]

haromszogSzLs :: Int -> [Integer]
haromszogSzLs n = take n haromszogSz
```

```
> haromszogSzLs 10
[1,3,6,10,15,21,28,36,45,55]
```

A `haromszogSz` függvény `take` nélküli meghívása most is egy végtelen lista kigenerálásához vezetne. A kódsor megértése végett itt is alternatív meghívásokkal próbálkozhatunk:

```
>  scanl' (+) 1 []
>  scanl' (+) 1 [2]
>  scanl' (+) 1 [2,3]
>  scanl' (+) 1 [2,3,4]
```

5.11. feladat Írjunk egy Haskell-függvényt, amely a `scanl'` függvényt használva kigenerálja az első `n` harmonikus számot egy listába, ahol az `n`-edik harmonikus szám, a H_n egyenlő az első `n` pozitív egész szám reciprokának az összegével, azaz: $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = \sum_{k=1}^{n} \frac{1}{k}$.

A feladatot kétféleképpen oldjuk meg, az első megoldás során a harmonikus számokat egy kételemű tuple elemtípusú listába generáljuk ki, és megírjuk azt az `op` függvényt, amely két racionális szám összeadását úgy határozza meg, hogy az eredmény is egy kételemű tuple típusú érték lesz. Az `op` függvényben használni fogjuk a `gcd` beépített függvényt, amely két szám legnagyobb közös osztóját határozza meg. A pozitív egész számok reciprok sorozatának az előállítását a `zip (repeat 1) [2..]`-el végezzük, éppen ezért a `harmonikusSz` függvény meghívásakor vigyázni kell, hogy a kimenet ne egy végtelen elemszámú lista legyen.

```
harmonikusSzLs :: Int -> [(Integer, Integer)]
harmonikusSzLs n = take n harmonikusSz

harmonikusSz :: [(Integer, Integer)]
harmonikusSz = scanl' op (1,1) $ zip (repeat 1) [2..]
  where
  op (x1, x2) (y1, y2) = (x, y)
       where
       a = x1 * y2 + x2 * y1
       b = x2 * y2
       g = gcd a b
       x = div a g
       y = div b g

> harmonikusSzLs 50
[(1,1),(3,2),(11,6)...(13943237577224054960759,30990...)]
```

A második módszernél a `Data.Ratio` könyvtárcsomagban található `Rational` típussal dolgozunk. Ebben az esetben nem kell megírni a két racionális szám összeadását meghatározó függvényt, mert a + operátor felül van írva, ahogyan azt korábban már mutattuk.

Algoritmikailag a `harmonikusSzLs_` hasonló elgondoláson alapszik, mint az előző változat, de a `Rational` típus alkalmazásával kompaktabb kódot kapunk.

```
import Data.Ratio ( Ratio, (%) )

harmonikusSzLs_ :: Int -> [Ratio Integer]
harmonikusSzLs_ n = take n harmonikusSz_

harmonikusSz_ :: [Ratio Integer]
harmonikusSz_ = scanl' op (1 % 1) $ zip (repeat 1) [2..]
  where
   op x (y1, y2) = x + (y1 % y2)

> harmonikusSzLs_ 50
[1 % 1,3 % 2,11 % 6,...,13943237577224054960759 % 3099...]
```

5.6. Kitűzött feladatok

5.1. feladat Implementáljuk két különböző explicit rekurziót alkalmazva, könyvtárfüggvények használata nélkül a következő Haskell-függvényeket:

`length, product, minimum, maximum, (!!), notElem, concat, cycle, repeat, replicate, iterate, all`

5.2. feladat Írjunk egy Haskell-függvényt, amely könyvtárfüggvények használata nélkül egy tetszőleges elemtípusú lista első elemét elköltözteti a lista végére.

5.3. feladat Írjunk egy Haskell-függvényt, amely meghatározza az `a` és `b` közötti Fibonacci-számokat.

5.4. feladat Írjunk egy Haskell-függvényt, amely meghatározza az első `n` elemét a következő szerkezetű listának:
`[1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, ...]`.

5.5. feladat Írjunk egy Haskell-függvényt, amely meghatározza az első `n` elemét a következő szerkezetű listának:

`[2, 4, 4, 6, 6, 6, 8, 8, 8, ...]`.

5.6. feladat Írjunk egy Haskell-függvényt, amely egy adott `x` értékre meghatározza az első `n` elemét a következő szerkezetű listának:

$[x, x^2, x^2, x^3, x^3, x^3, x^4, x^4, x^4, x^4, x^5 \ldots]$.

5.7. feladat Írjunk egy Haskell-függvényt, amely meghatározza az első `n` elemét annak a listának, amely váltakozva tartalmazza a `0, 1, -1` értékeket.

5.8. feladat Írjunk egy-egy Haskell-függvényt, amely könyvtárfüggvények használata nélkül

- meghatározza egy numerikus elemtípusú lista pozitív elemeinek átlagát,
- meghatározza azt a listát, amely tartalmazza egy tetszőleges elemtípusú lista minden `n`-edik elemét,
- meghatározza egy lista leggyakrabban előforduló elemét.

5.9. feladat Írjunk egy Haskell-függvényt, amely könyvtárfüggvények használata nélkül megállapítja egy tetszőleges elemtípusú listáról, hogy az palindrom-e vagy sem. Egy lista palindrom, ha az elemek fordított sorrendje megegyezik az elemek eredeti sorrendjével.

5.10. feladat Írjunk egy Haskell-függvényt, amely egy természetes számokat tartalmazó lista esetében szétválogatja a Fibonacci-számokat és a nem Fibonacci-számokat.

5.11. feladat Adva van egy `pP` pont koordinátája a kétdimenziós síkban, és adott az `lsP`, amely pontok koordinátáinak a listája. Írjunk egy `kiirPont` Haskell-függvényt, amely meghatározza annak az `lsP`-beli pontnak a koordinátáját, amely legközelebb van a `pP` ponthoz, illetve kiírja a kapott minimális távolságot.

A következő sorok példát adnak az értékadásra, a függvényhívásra és a kiértékelés eredményére:

```
> pP = (0.0, 0.0)
> lsP = [(1.9, 4.5), (1.4, 2.9), (3.2, 6.2), (17.9, 1.2)]
> kiirPont pP lsP
```

```
a pont koordinatai: (1.4,2.9)
a minimalis tavolsag: 3.2202484376209237
```

5.12. feladat Írjunk egy Haskell-függvényt, amely egy `String` típusú listából meghatározza azokat a szavakat, amelyek karakterszáma a legkisebb. Például ha a lista a következő szavakat tartalmazza:

```
function class Float higher-order monad tuple variable
Maybe recursion
```

akkor az eredmény-lista a következőkből áll:

```
class Float monad tuple Maybe
```

5.13. feladat Írjunk egy `talalat` Haskell-függvényt, amely meghatározza azt a listát, amely a bemeneti listában megkeresi egy megadott elem előfordulási pozícióit.

A következő sorok példát adnak a függvényhívásra és a kiértékelés eredményére, ahol az első az **5**-ös előfordulási pozícióinak, míg a második az **e** előfordulási pozícióinak listája lesz.

```
> talalat 5 [3, 13, 5, 6, 7, 12, 5, 8, 5]
[2, 6, 8]
> talalat 'e' "Bigeri-vizeses"
[3,10,12]
```

5.14. feladat Írjunk egy `osszegT` Haskell-függvényt, amely meghatározza egy `(String, Int)` értékpárokból álló lista esetében az értékpárok második elemeiből képzett összeget.

A következő sorok példát adnak az értékadásra, a függvényhívásra és a kiértékelés eredményére:

```
> ls = [("golya",120),("fecske",85),("cinege",132)]
> osszegT ls
337
```

5.15. feladat Írjunk egy `atlagTu` Haskell-függvényt, amely egy kételemű, tuple elemtípusú lista esetében átlagértékeket számol a második elem szerepét betöltő listaelemeken. Az eredmény egy tuple elemtípusú lista legyen, amelynek kiíratása során a tuple-elemeket formázzuk, és külön sorba írjuk őket.

A következő sorok példát adnak az értékadásra, a függvényhívásra és a kiértékelés eredményére:

```
> :set +m
> ls = [("mari",[10, 6, 5.5, 8]), ("feri",[8.5, 9.5]),
| ("zsuzsa",[4.5, 7.9, 10]),("levi", [8.5, 9.5, 10, 7.5])]
> atlagTu ls
mari 7.375
feri 9.0
zsuzsa 7.466666666666666
levi 8.875
```

5.16. feladat Írjunk egy Haskell-függvényt, amely szövegállományban levő számok mindegyikére meghatározza a szám előtti legnagyobb, illetve a szám utáni legkisebb prímszámot. Az eredményeket a képernyőre írjuk, minden sorba három számot tegyünk, és válasszuk el őket szóközökkel.

5.17. feladat Írjunk egy Haskell-függvényt, amely meghatározza a $P(x) = a_n \cdot x_n + a_{n-1} \cdot x_{n-1} \ldots a_2 \cdot x_1 + a_0$ polinom adott x_0 értékre való behelyettesítési értékét. Adjuk meg a függvénynek azokat a változatát is, amelyben a `foldl'`, illetve `foldr` függvényeket használjuk.

5.18. feladat Írjunk Haskell-függvényeket, amelyek a `foldr` vagy `foldl'` függvény segítségével rendre implementálják a következő függvényeket:

```
length, product, minimum, reverse, (++), (!!),
notElem, all.
```

5.19. feladat Írjunk Haskell-függvényt, amely a `foldl'`, majd a `foldr` függvényt alkalmazva meghatározza egy lista pozitív elemeinek összegét.

5.20. feladat Írjunk Haskell-függvényt, amely a `foldl'`, majd a `foldr` függvényt alkalmazva meghatározza egy lista páros elemeinek szorzatát.

5.21. feladat Írjunk Haskell-függvényt, amely a `foldl'`, majd a `foldr` függvényt alkalmazva meghatározza `n`-ig a négyzetszámokat.

5.22. feladat Írjunk Haskell-függvényt, amely megadott `n`-ig, a `scanl` függvényt alkalmazva kigenerálja a négyzetszámok listáját. Alkalmazzuk azt az összefüggést amely szerint a `k`-adik négyzetszám megegyezik az első `k` páratlan szám összegével.

6. fejezet

Írás, olvasás

6.1. A `Show` és a `Read` típusosztályok

A `Show` és a `Read` típusosztályokhoz tartozó `show`, illetve `read` függvényeket az egyszerű típusú adatoktól kezdve a bonyolultabb, a felhasználók által definiált adatszerkezetekig használják, elsősorban írás, illetve olvasás műveleteknél. Tulajdonképpen egy adott típus és egy `String` típus közötti átalakítást tesznek lehetővé. Éppen ezért a különböző adatbeviteli és kiíratási függvények tárgyalása előtt ezekről lesz szó.

A **`Show`** azokat a típusokat tartalmazza, amelyek átalakíthatók karakterlánccá (`String`-gé). Magába foglalja az alaptípusokat, azaz a `Bool`, `Char`, `Int`, `Integer`, `Float`, `Double`, `List`, `Tuple` típusokat, de ez bővíthető aszerint, hogy egy új típus definiálásakor a `deriving` kulcsszóval specifikáljuk-e ezt vagy sem. Leggyakrabban használt függvénye a `show`, amelyet korábban már többször is használtunk.

6.1. feladat Írjunk Haskell-függvényt, amely külön sorokba írja egy `String` típusú elemekből álló lista elemeit.

```
ls1 = ["Dell", "HP", "ASUS", "Lenovo", "Toshiba"]

myShow1 :: [String] -> String
myShow1 [] = ""
myShow1 (k : ve) = k ++ "\n" ++ myShow1 ve

> putStr $ myShow1 ls1
Dell
```

```
HP
ASUS
Lenovo
Toshiba
```

A képernyőre való kiíratást a putStr könyvtárfüggvénnyel végeztük, amely bemenetként egy String típusú értéket kapott. Ezt a String típusú értéket a myShow1 függvényben úgy hozzuk létre, hogy a ++ operátor segítségével a listaelemek közé '\n' karaktert tettünk. A rendezett sorrend meghatározásához alkalmazhatjuk a sort beépített függvényt:

```
> import Data.List (sort)
> putStr $ (myShow1 . sort) ls1
```

6.2. feladat Írjunk Haskell-függvényt, amely kiírja külön sorokba azokat az elemeket egy (String, Int, Int) típusú elemhármasokból álló lista esetében, ahol a harmadik elem pozitív.

```
ls2 = [("Samsung",1,200), ("Apple",2,100),
       ("Huawei",3,-300), ("BlackBerry",4,-400),
       ("HTC",5,250), ("Nokia",6,600)]

myShow2 :: [(String, Integer, Integer)] -> String
myShow2 [] = ""
myShow2 (k : ve)
   | k3 > 0 = temp ++ myShow2 ve
   | otherwise = myShow2 ve
       where
       temp = k1 ++" "++ show k2 ++" "++ show k3 ++"\n"
       (k1, k2, k3) = k

> (putStr . myShow2) ls2
Samsung 1 200
Apple 2 100
HTC 5 250
Nokia 6 600
```

A myShow2 függvényben a temp jelöli azt a karakterláncot, amelyet a tupleelemek értékei alapján hoztunk létre. A tuple második és harmadik értékét a show függvénnyel átalakítottuk String típussá, és a ++ operátort alkalmazva egymás után fűztük az értékeket és szóközöket.

Rendezett sorrendben is kiírhatjuk az adatokat, például a harmadik elem szerinti rendezéshez felhasználjuk a `myThd`, a `sortBy`, illetve a `comparing` függvényeket.

```
import Data.List (sortBy)
import Data.Ord (comparing)

myThd :: (a, b, c) -> c
myThd(t1, t2, t3) = t3

> putStr $ myShow2 $ sortBy (comparing myThd) ls2
Apple 2 100
Samsung 1 200
HTC 5 250
Nokia 6 600
```

A rendezést megoldhatjuk a `sortOn` függvénnyel is:

```
> import Data.List(sortOn)
> putStr $ myShow2 $ sortOn myThd ls2
Apple 2 100
...
```

6.3. feladat Írjunk Haskell-függvényt, amely kiírja külön sorokba rendezve egy `(String, Integer, [Double])` típusú elemhármasokból álló lista esetében a `String` értékeket és a `Double` elemtípusú lista elemei közül a maximum elemet.

```
ls3 = [("Ferenc", 1, [7.5, 9.25]),
       ("Katalin", 2, [7.75, 6.25, 10, 7.55]),
       ("Zsuzsa", 4, [7.33, 8.25, 9.75]),
       ("Maria", 3, [6.5, 8.25, 9.33]),
       ("David", 5, [7.90, 9.85, 9.95, 8.55])]

myShow3 :: [(String, Integer, [Double])] -> String
myShow3 [] = ""
myShow3 (k : ve) = temp ++ myShow3 ve
       where
       temp = k1 ++ " " ++ show (maximum k3) ++ "\n"
       (k1, k2, k3) = k

> putStr $ myShow3 ls3
Ferenc 9.25
Katalin 10.0
```

```
Zsuzsa 9.75
Maria 9.33
David 9.95
```

A maximum meghatározásához a beépített `maximum` függvényt használtuk, amely értéket a `show` függvénnyel `String` típusú adattá alakítottunk azért, hogy a `++` operátorral össze tudjuk fűzni a tuple első elemével, a szóközzel és az új sor jellel.

A **Read**, a `Show` típusosztály párja, a hozzá tartozó `read` függvény paraméterként egy `String` típusú értéket vár, és azt átalakítja egy olyan típussá, ami a `Read` osztályhoz tartozik, ahogy az a típusdeklarációból is látható:

```
read :: Read a => String -> a
```

A következő lekérdezésekben rendre a paraméterként megadott `String` típusú értékeket alakítjuk át `Bool`, `[Int]`, `Double` típusú értékekké, illetve adunk olyan példákat is, ahol nem sikerül az átalakítás. Figyeljük meg, hogy a `read` alkalmazásakor mindig specifikálni kell, hogy a kimenet milyen típusú érték legyen.

```
> read "False" :: Bool
False

> read "5.12" :: Double
5.12

> read "[10, 11, 12, 13]" :: [Int]
[10, 11, 12, 13]

> read "5.12" :: Int
*** Exception: Prelude.read: no parse...

> read ["1", "2", "3"] :: [Int]
read ["1", "2", "3"] :: [Int]
... error:
Couldn't match expected type...
```

Vegyük észre, hogy `String`-ként, de az `Int` lista szintaxisa szerint megadott bemenetet át tudtuk alakítani `[Int]` típussá, azonban ha olyan formában adjuk meg a `read` függvény bemenetét, hogy az nem alakítható át, akkor futási hibát kapunk.

A következő példákban figyeljük meg a bemenet szerkezetét. Az első két esetben `String`-ekből álló listát adunk meg bemenetnek, ahol mindegyik

`String` átalakítható `Int` típusú értékké, így a kimenet egy `[Int]` típusú érték lesz. A harmadik lekérdezésben egy olyan `String`-et adunk meg, amely egész számokat tartalmaz, és amelyek mindegyike átalakítható `Int` típusú értékké. A negyedik lekérdezés hasonló a harmadikhoz, de most `[Double]` típusú lesz a kimenet.

```
myReadStr :: [String] -> [Int]
myReadStr = map read

> myReadStr ["101", "41", "1789"]
[101,41,1789]

> map (read :: String -> Int) ["101", "41", "1789"]
[101,41,1789]

> map (read :: String -> Int) $ words "12 34 56"
[12,34,56]

> map (read :: String -> Double) $ words "1.2 3.14 2.72"
[1.2,3.14,2.72]
```

6.2. Haskell-monádok

Az eddig megírt Haskell-függvényeket konstans bemenetekre hívtuk, de természetesen felmerül az igény, hogy az adatokat billentyűzetről vagy állományból olvassuk be, és a képernyőre vagy egy állományba írjuk ki. A Haskell, hasonlóan más programozási nyelvekhez, biztosítja az adatbevitelhez és az adatkiíráshoz szükséges eszközöket, azonban ezt korántsem volt olyan egyszerű megoldani.

Az olvasás/írás (`I/O`) műveletek mechanizmusa nem illik bele a funkcionális paradigmába, ezért egy olyan eszközt, olyan struktúrát kellett kidolgozni, amely alkalmas az `I/O` műveletek hatékony használatára, anélkül, hogy az megsértené a funkcionális gondolkodásmódot. A Haskell esetében ezt egy *monád*-nak nevezett struktúra bevezetésével oldották meg. A monádok alkalmazásával az `I/O` műveletek mellett a tömbök, a hibakezelések hatékony használatát is megvalósították, azaz minden olyan programozási művelet, amely mellékhatással jár, alkalmazható lett a Haskellben.

A monád fogalma a kategóriaelméletből származik. A kategóriaelmélet a legáltalánosabb matematikai struktúrák közötti kapcsolatokat írja le, az

ezzel kapcsolatos értelmezések, fogalmak pontos leírását pedig megtaláljuk Awodey könyvében[1]. A monád egy számítási adattípust definiál, ami azt jelenti, hogy megadjuk, hogy egy adattípus értékein milyen számításokat végezhetünk, és ezek a számítások hogyan kombinálhatók.

Mint ahogyan eddig is megfigyelhettük, egy Haskell-függvény kiértékelése nem a programozó által definiált műveletek egymás után való végrehajtásából áll, azonban az `I/O` műveletek elvégzése nem oldható meg másként. Monádok segítségével megadhatjuk, hogy az `I/O` műveleteket hogyan kombináljuk, azaz milyen sorrendben végezzük el őket, ilyenformán az imperatív nyelvekhez hasonló módon lehet megadni ezen műveletek kiértékelési sorrendjét.

Ebben a fejezetben először a `Monad m` típusosztályról lesz szó, majd az `IO` monádot mutatjuk be, amelynek segítségével a Haskellben olvasás, írás műveleteket végezhetünk. Későbbi fejezetekben bevezetésre fog kerülni még a `Maybe` monád.

Az `IO` monád és a `Maybe` monád is a `Monad m` típusosztálynak a példányai, ahol a `Monad m` típusosztály a standard Prelude-ben a következőképpen van definiálva:

```
class Monad m where
    (>>=) :: m a -> (a -> m b) -> m b
    return :: a -> m a
```

A fenti típusosztályban két művelet, azaz két függvény/operátor típusdeklarációját láthatjuk.

A `>>=` függvény számítások (akciók) láncolását teszi lehetővé. Többször fogjuk használni a későbbiekben, csak egyszerűbben, egy `do` kifejezést (blokkot, jelölést) fogunk használni helyette, ahogyan azt korábban is már mutattuk. A `>>=` függvény például `do` jelölésre átírva a következőképpen néz ki:

```
m >>= f = do
            x <- m
            f x
```

Ez azt jelenti, hogy először végrehajtjuk az `m` számítási sorozatot, amelynek eredménye az `x`-be kerül, majd meghívjuk az `f` függvényt az `x` bemeneten, amelynek eredménye végül a `do` blokk eredménye lesz. A következő `olvasIr_` függvény egy karakterláncot vár a billentyűzetről, amelyet rögtön a beolvasás után ki is ír a képernyőre, ahol az általunk beolvasott értéket most is és a következőkben is mindig dőlt betűkkel fogjuk megjeleníteni. A

kiíratáshoz a `putStrLn` függvényt alkalmaztuk, amely hasonló a `putStr`-hez, csak használatakor a kiíratást automatikusan egy *new line*, azaz egy új sor karakter is követi.

```
olvasIr_ :: IO ()
olvasIr_ = getLine >>= putStrLn

> olvasIr_
Hello Haskell!
Hello Haskell!
```

A `do` jelölést használva a következőképpen módosul a függvény:

```
olvasIr :: IO ()
olvasIr = do
   x <- getLine
   putStrLn x
```

A `>>` függvény két akció együttes elvégzését teszi lehetővé, azaz először az első, és utána a második akció kerül végrehajtásra, ahol az első akció által meghatározott érték nem lesz releváns. A következő függvény két karakterlánc kiírását végzi:

```
ir_ :: IO ()
ir_ = putStr "Hello " >> putStrLn " Haskell!"

> ir_
Hello Haskell!
```

Ennek a függvénynek is megadjuk a `do` jelöléssel megírt változatát:

```
ir :: IO ()
ir = do
   putStr "Hello "
   putStrLn " Haskell!"
```

A következőkben az akciók megadását, illetve az egymás után való láncolásukat, ahogy eddig is tettük, ezután is egy `do` blokkban fogjuk megadni. Megjegyezzük még, hogy minden akciót külön sorba kell írni.

A `return` függvény segítségével elérjük, hogy a paramétereként megadott adatot *becsomagoljuk* egy `IO` monádba. A következő függvény egy karakterláncot olvas be a billentyűzetről, majd a beolvasott értéket a `return` alkalmazásával becsomagolja egy `IO` monádba, ezért a függvény kimenetének típusa `IO String` lesz.

```
olvasInt_ :: IO String
olvasInt_ = do
   temp <- getLine
   return temp

> olvasInt_
12
"12"
```

A következőkben az `olvasInt_` által meghatározott értéket egy `<-` művelet segítségével lekérjük, mondhatjuk azt is, hogy *kicsomagoljuk*, és a kapott értéket `Int` típusú adattá alakítjuk:

```
> x <- olvasInt_
12
> read x :: Int
12
```

Az `Int` típusú adattá való átalakítás elvégezhető a függvénytörzsben is, ahogyan ezt a következő `olvasInt` függvényben láthatjuk. A `read` függvény által meghatározott érték lekérését `let ... =` jelölés használatával fogjuk megoldani, mert így jelezzük a Haskellnek, hogy most egy tiszta függvény kiértékelése következik, nem egy akció végrehajtása. Figyeljük meg, hogy a `let`-et ebben az esetben nem lokális definíció megadására használjuk, ahogyan azt korábban tettük. A `return` alkalmazásával az `olvasInt` függvény kimenetét most is becsomagoljuk egy `IO` monádba, a különbség azonban az, hogy az `olvasInt` kimenete most `IO Int` típusú érték lesz.

```
olvasInt :: IO Int
olvasInt = do
   temp <- getLine
   let x = read temp :: Int
   return x

> olvasInt
12
12
```

Figyeljük meg, hogy az `olvasIr_`, `olvasIr`, `ir_`, illetve `ir` függvények kimeneti értékének típusa `IO ()`, ami azt jelenti, hogy *semmit*, azaz `()`-t eredményeznek a függvényhívások. Haskellben a *tiszta* függvények mindig meghatároznak egy értéket. Az `I/O` műveletek során azonban ez nem mindig lehetséges, ezért erre az esetre definiálták az `IO ()` típust. Egy függvény kimeneti értékének típusa akkor `IO ()`, ha az akciók sorát egy olyan `I/O`

függvény zárja, amely kimeneti értékének típusa szintén `IO ()`, vagy az utolsó akció `return ()`.

6.3. A standard bemenet és kimenet

A következőkben példákon keresztül mutatjuk be, hogy a Haskellben hogyan használjuk a standard bemenetet és kimenetet.

6.4. feladat Írjunk egy Haskell-függvényt, amely meghatározza a billentyűzetről beolvasott `n` szám páros osztóinak listáját.

```
foParosO :: IO ()
foParosO = do
   putStr "n = "
   temp <- getLine
   let n = read temp :: Int
   let res = [i | i <- [2, 4 .. n], mod n i == 0]
   print res

> foParosO
n: 60
[2,4,6,10,12,20,30,60]
```

A `foParosO` függvénynek nincs bemeneti paramétere, kimeneti értékének típusa `IO ()`. A `getLine` függvény, ahogy ezt már megfigyelhettük, a billentyűzetről történő adatbevitelt teszi lehetővé, típusdeklarációjából jól látszik, hogy nem vár bemeneti értéket, kimenetének típusa pedig `IO String`, azaz egy `IO` monád lesz, ami egy olyan számítást definiál, ahol a végeredmény típusa `String`. Hasonlóan a `putStr` és a `print` is egy-egy `IO` monád, amelyek olyan számításokat definiálnak, ahol a végeredmény típusa `()`, a függvények kimenetének típusa pedig `IO ()`.

```
getLine :: IO String
putStr :: String -> IO ()
print :: Show a => a -> IO ()
```

A `foParosO` függvény törzse egy `do` blokk keretén belül több, összesen öt `akció`-nak nevezett műveletet hajt végre, abban a sorrendben, ahogyan megadtuk. Szerepük a következő:

1. a `putStr` függvénnyel kiírunk a képernyőre,

2. a `getLine` függvénnyel adatot olvasunk be billentyűzetről, amelyet a **`<-`** használatával kicsomagolunk és `temp`-pel jelölünk,
3. a billentyűzetről beolvasott `String` típusú értéket átalakítjuk `Int` típussá, amelyet a **`let ... =`** jelölést használva `n`-el jelölünk,
4. a kért lista előállításához halmazműveletet használunk, az eredményt a `res`-sel jelöljük, megint a `let ... =` jelölést használjuk,
5. a `print` függvénnyel kiírunk a képernyőre.

A fenti akciók közül a harmadik és negyedik akció olyan függvények kiértékelését valósítják meg, amelyeknek nincs mellékhatásuk, ezért használjuk a `let ... =` jelölést, azaz jelezzük, hogy tiszta funkcionális programozási stílusban megírt függvényeket kell kiértékelni. A Haskell tehát külön jelölési rendszert dolgozott ki arra, amikor egy tiszta függvényt akar kiértékelni, és mást használ, amikor egy mellékhatással járó függvényt hajt végre. Élesen elkülöníti a tisztán funkcionális paradigmában írt függvényeket a mellékhatással járó függvényektől.

A következő függvény egy karakterláncot olvas be a billentyűzetről, majd meghatározza azt a két karakterláncot, amelyben csak a kis- és nagybetűk, illetve azt, amelyben csak a számjegyek szerepelnek. Figyeljük meg, hogy a `return` használatakor nem következik be az `olvasStr_`-ből való kilépés.

```
import Data.Char (isAlpha, isNumber)

olvasStr_ :: IO ()
olvasStr_ = do
   putStr "kerek egy karakterlancot: "
   str <- getLine
   x <- return (filter isAlpha str)
   y <- return (filter isNumber str)
   putStrLn $ "Az alfanumerikus karakterek: " ++ x
   putStrLn $ "A szamok: " ++ y
```

A következő függvény ugyanazt végzi, mint az előző, csak az implementáció során a `filter` által meghatározott értékeket másképpen kérdezzük le. A jegyzet további részében ezen utóbbi módszer szerint fogunk eljárni.

```
olvasStr :: IO ()
olvasStr = do
   putStr "kerek egy karakterlancot: "
   str <- getLine
```

```
    let x = filter isAlpha str
    let y = filter isNumber str
    putStrLn $ "Az alfanumerikus karakterek: " ++ x
    putStrLn $ "A szamok: " ++ y

> olvasStr
kerek egy karakterlancot: Marosvasarhely 2001 oktober 1
Az alfanumerikus karakterek: Marosvasarhelyoktober
A szamok: 20011
```

6.5. feladat Írjunk egy Haskell-függvényt, amely meghatározza a billentyűzetről beolvasott számok rendezett sorrendjét. Használjuk a `sort` könyvtárfüggvényt.

```
import Data.List (sort)

foRendez :: IO ()
foRendez = do
   putStr "szamokat kerek: "
   ls <- olvasSzamok1
   let sortLs = sort ls
   putStr "rendezve: "
   print sortLs

olvasSzamok1 :: IO [Int]
olvasSzamok1 = do
   temp <- getLine
   let ls = read temp :: [Int]
   return ls

> foRendez
szamokat kerek: [12, 4, 67, 8, 9]
rendezve: [4,8,9,12,67]
```

Hasonlóan az előző példához, egy `do` blokk keretén belül a `foRendez` függvényben több akció kerül végrehajtásra.

Az `olvasSzamok1` függvény számok beolvasását biztosítja, amelyben a `return` segítségével az `ls` listát becsomagoljuk, azaz létrehozunk egy `IO` monádot, így az `olvasSzamok1` kimenetének típusa `IO [Int]` lesz. Itt vigyáznunk kell arra, hogy helyes formátumban olvassuk be a bemenetet, azaz ha nem megfelelő helyen használunk szögletes zárójeleket, illetve vesszőket, akkor futási hibát kapunk, ahogy a következő lekérdezésből ez kitűnik:

```
> foRendez
szamokat kerek: 12, 4, 7, 8
rendezve: *** Exception: Prelude.read: no parse...
```

6.6. feladat Írjunk egy Haskell-függvényt, amely valós számokat olvas be a billentyűzetről, majd a számokat az egészrészük alapján különböző csoportokba, azaz listákba teszi.

```
import Data.List (sort, groupBy)

foCsoportosit :: IO ()
foCsoportosit = do
   putStr "szamokat kerek: "
   ls <- olvasSzamok2
   let gLs = groupBy (\x y -> truncate x == truncate y)
                              $ sort ls
   putStr "csoportositva: "
   print gLs

olvasSzamok2 :: IO [Double]
olvasSzamok2 = do
   temp <- getLine
   let ls = map (read :: String -> Double) $ words temp
   return ls

> foCsoportosit
szamokat kerek:1.77 5.6 1.4 5.34 2.7 2.9 1.9 2.4 1.33
csoportositva:
[[1.33,1.4,1.77,1.9],[2.4,2.7,2.9],[5.34,5.6]]
```

Ahogy a korábbi példáknál láttuk, a `foCsoportosit` függvény itt is akciók egymásutánjából áll, amelyek adatbevitelt, illetve adatfeldolgozást tesznek lehetővé. Az előző példához képest a `words` és `map` függvények használata miatt a számok bevitelét azonban más formában kell megadni. Most a számokat egy sorban, szóközöket téve közéjük kell beírni, mert ellenkező esetben futási hibát kapunk. Az `olvasSzamok2` függvényben a `temp`-be beolvasott karakterláncot a `words` függvénnyel előbb szavakra bontjuk, azaz létrehozunk egy `String` elemekből álló listát, majd a `map` segítségével egyenként átalakítjuk őket `Double` típusúvá, így a függvény kimenete `IO [Double]` lesz.

Az egészrész alapján történő csoportosításhoz először a `sort` függvénnyel rendezzük az `ls` listát, majd a `groupBy` függvénnyel csoportosítjuk

az egymás után következő, azonos egész résszel rendelkező elemeket. Az egész részt a `truncate` könyvtárfüggvénnyel határozzuk meg.

A csoportok kiíratását elegánsabb formában is megoldhatjuk, ha minden sorba csak az azonos csoportbeli elemeket írjuk ki, nem mint listákat, hanem egyenként. Ennek megfelelően módosítjuk a `foCsoportosit` utolsó sorát, és egy saját kiíró függvényt adunk meg, a `myPrintList`-et:

```
foCsoportosit_ :: IO ()
foCsoportosit_ = do
   putStr "szamokat kerek: "
   ls <- olvasSzamok2
   let gLs = groupBy (\x y -> truncate x == truncate y)
                              $ sort ls
   putStr "csoportositva: \n"
   myPrintList gLs

myPrintList :: [[Double]] -> IO ()
myPrintList = mapM_ auxF1
   where
   auxF1 :: [Double] -> IO ()
   auxF1 kLs = do
       mapM_ auxF2 kLs
       putStrLn ""

   auxF2 :: Double -> IO ()
   auxF2 k = putStr $ show k ++ " "
```

```
> foCsoportosit_
szamokat kerek: 1.77 5.6 1.4 5.34 2.7 2.9 1.9 2.4 1.33
csoportositva:
1.33 1.4 1.77 1.9
2.4 2.7 2.9
5.34 5.6
```

A `myPrintList` függvényben kétszer is használatra került a már korábban is alkalmazott `mapM_` beépített függvény. Az első `mapM_` a sorok kiíratásáért felelős, a második a sorokon belüli értékeket írja ki, szóközt téve az elemek közé.

Haskellben nincs egész, illetve valós számokat beolvasó könyvtárfüggvény, de ez könnyen megoldható. Korábban megadtuk az `olvasInt` függvényt, amelynek visszatérési értéke `Int` volt, a következő `olvasDouble` pedig `Double` típusú érték beolvasását teszi lehetővé.

```
olvasDouble :: IO Double
olvasDouble = do
    str <- getLine
    return (read str :: Double)
```

A következő függvény több valós számot olvas be a billentyűzetről, ahol minden szám beolvasása után `enter`-t kell nyomni. Paraméterként meg kell adni, hogy hány számot szeretnénk beolvasni, a függvény pedig egy `IO [Double]` típusú értéket határoz meg. A kimeneti listában a számok sorrendje megegyezik a beolvasási sorrenddel.

```
olvasNszam :: Int -> IO [Double]
olvasNszam n = do
    k <- olvasDouble
    if n == 1 then return [k]
    else do
        ve <- olvasNszam (n-1)
        return (k : ve)

> olvasNszam 3
12.6
34.3
5.7
[12.6,34.3,5.7]
```

A következő `olvasSzam` függvény egész számok beolvasását végzi, és addig kéri a számokat, amíg üres sort nem adunk. A függvény által meghatározott kimenet típusa `IO [Int]` lesz, és megadja a beolvasott számokat, fordított sorrendben, mint ahogy beolvastuk őket a billentyűzetről.

```
olvasSzam :: IO [Int]
olvasSzam = auxOlvas []
    where
    auxOlvas :: [Int] -> IO [Int]
    auxOlvas res = do
        temp <- getLine
        if null temp then return res
        else do
            let k = read temp :: Int
            auxOlvas (k : res)

> olvasSzam
3
6
```

```
7

[7,6,3]
```

A következő példák számrendszerek közötti átalakításokat fognak bemutatni, amelyeket további `I/O` műveleteket végző példáknál fogunk felhasználni.

6.7. feladat Írjunk egy Haskell-függvényt, amely meghatározza egy szám tetszőleges számrendszerbeli alakja alapján a szám `10`-es számrendszerbeli alakját.

```
convToNr :: (Integral a) => [a] -> a -> a
convToNr ls b = auxConv ls b 0
   where
   auxConv :: (Num a) => [a] -> a -> a -> a
   auxConv [] b res = res
   auxConv (k : ve) b res = auxConv ve b (k + b * res)

> convToNr [1,1,1,0,1] 2
29
```

A `convtToNr` függvény első paramétere egy lista, amely a `b` számrendszerbeli számjegyeket fogja tartalmazni. Kimenete a `10`-es számrendszerbeli szám. A függvény egy `auxConv` segédfüggvényt használ, ami lehetővé teszi a kezdeti értékadást, illetve hogy az eredményt a harmadik paraméterében akkor számoljuk, amikor megyünk be a rekurzióba, ahol az eredmény meghatározása azt jelenti, hogy a megadott számjegyekből kiszámítjuk a megfelelő hatványok összegét. Első ránézésre lehet, hogy nem egyértelmű a hatványösszeg meghatározása, ezért a fenti bemenetre bemutatjuk a lépésenkénti műveleteket:

```
   k : ve          res --> k + 2 * res
[1,1,1,0,1]          0
[1,1,0,1]         -> 1 + 2·0
[1,0,1]           -> 1 + 2·(1 + 2·0)
[0,1]             -> 1 + 2·(1 + 2·(1 + 2·0))
[1]               -> 0 + 2·(1 + 2·(1 + 2·(1 + 2·0)))
[]                -> 1 + 2·(0 + 2·(1 + 2·(1 + 2·(1 + 2·0))))

-> 1 + 2^1·0 + 2^2·1 + 2^3· 1 + 2^4·1 + 2^5·0 -> 29
```

A fordított műveletet a `convFromNr` függvény végzi, amelynek bemeneti paraméterként meg kell adni egy tízes számrendszerbeli számot és egy

b számrendszert, hogy eredményként meghatározza egy Int elemtípusú listába a b számrendszerbeli számjegyeket.

6.8. feladat Írjunk egy Haskell-függvényt, amely meghatározza egy 10-es számrendszerbeli szám tetszőleges b számrendszerbeli alakját.

```
convFromNr :: Integral a => a -> a -> [a]
convFromNr nr b = auxConv nr b []
   where
   auxConv :: Integral a => a -> a -> [a] -> [a]
   auxConv nr b res
       | nr < b = nr : res
       | otherwise = auxConv d b (r : res)
       where
       r = rem nr b
       d = div nr b

> convFromNr 256 2
[1,0,0,0,0,0,0,0,0]
```

6.9. feladat Olvassuk be a billentyűzetről egy szám számjegyeit és a megfelelő számrendszer értékét. Alakítsuk át a számot 10-es számrendszerbe, majd ellenőrizzük az eredményt, használjuk a korábban megírt convToNr és convFromNr függvényeket.

```
foAlakit :: IO ()
foAlakit = do
   putStr "szamjegyek: "
   temp <- getLine
   let ls = map (read :: String -> Integer) $ words temp
   putStr "szr alap: "
   str <- getLine
   let b = read str :: Integer
   let nr = convToNr ls b
   putStr "eredmeny: "
   print nr
   let ls = convFromNr nr b
   putStr "ellenorzes: "
   print ls

> foAlakit
szamjegyek: 11 6 14 14 1
szr alap: 16
```

```
eredmeny: 749281
ellenorzes: [11,6,14,14,1]
```

Megjegyzés: az eredmény helyességét a következő összefüggés alapján tudjuk leellenőrizni: $749281 = 11 \cdot 16^4 + 6 \cdot 16^3 + 14 \cdot 16^2 + 14 \cdot 16 + 1$.

Összegzésképpen elmondhatjuk, hogy a Haskell két szempont szerint figyeli a programkódot: mikor kell kifejezést, azaz tiszta függvényt kiértékelni, illetve mikor kell egy adott akciót, azaz mellékhatással járó műveletet végrehajtani. A kifejezések kiértékelésekor nem parancsvégrehajtás történik, ezért ezekben a függvényekben nem léphetnek fel mellékhatások, ez a tulajdonképpeni tiszta funkcionális stílusban írt programrészek megadását jelenti. Az `I/O` műveletek, azaz akciók végrehajtásakor előállhatnak mellékhatások, ez az imperatív stílusú programrészek megadását jelenti.

6.4. Állománykezelés

Ebben a fejezetben a Haskell állománykezelő függvényeit mutatjuk be. Az első példák szövegállományokkal dolgoznak, a későbbiekben pedig bemutatjuk a bináris állományok feldolgozását.

6.10. feladat Írjunk egy Haskell-függvényt, amely Eratoszthenész szitájával kigenerálja az első `n` prímszámot, és az eredményt kiírja a `prim.txt` szövegállományba.

```
eSzita :: (Integral a) => Int -> [a]
eSzita n = take n (2: szita [3, 5..])

szita :: Integral a => [a] -> [a]
szita (k : ve) = k : szita [x | x <- ve, mod x k /= 0]
```

Az `eSzita` függvény egy listába generálja a prímszámokat a `szita` függvény segítségével. Kezdetben a `szita` függvény paraméterként a `3`-mal kezdődő páratlan számok egy végtelen listáját kapja meg. A `szita` függvény minden egyes rekurzív híváskor más és más listát fog kapni paraméterként, amelyben nem lesznek benne a lista első elemének többszörösei. Első körben a `3`, majd az `5`, majd a `7` stb. számok többszöröseit szitáljuk ki, azaz a prímszámok igen, de a többszöröseik nem kerülnek át az eredmény listába.

Az állományba való írást a `primIr` függvény oldja meg, a `writeFile` könyvtárfüggvény segítségével.

```
primIr :: IO ()
primIr = do
   putStr "n = "
   temp <- getLine
   let n = read temp :: Int
   let lPrim = eSzita n
   writeFile "prim.txt" (show lPrim)
```

A `writeFile` segítségével karakterláncokat tudunk állományba írni, ahol paraméterként meg kell adni egy állománynevet, adott esetben az útvonallal együtt, ahol az állomány található. Második paraméterként azt a `String` típusú értéket kell megadni, amit be szeretnénk írni az állományba. Típusdeklarációja a következő:

```
writeFile :: FilePath -> String -> IO ()
```

Vegyük észre, hogy a prímszámok listáját a `show` függvénnyel előbb átalakítottuk `String` típusú adattá, és a `writeFile` függvénynek az így kapott értéket adtuk meg második paraméterként.

6.11. feladat Írjunk egy Haskell-függvényt, amely beolvassa a `prim.txt` állomány tartalmát, majd kiírja a képernyőre a prímszámokat, illetve meghatározza a prímszámok számát.

```
primOlvas :: IO ()
primOlvas = do
   inf <- readFile "prim.txt"
   let lPrim = read inf :: [Int]
   print lPrim
   let nr = length lPrim
   putStr "primek szama: "
   print nr
```

A `readFile` a `writeFile` társa, a megadott állományból beolvassa az adatokat, típusdeklarációja a következő:

```
readFile :: FilePath -> IO String
```

A típusdeklarációból jól látható, hogy a függvény kimenetének típusa egy `IO String`, éppen ezért az adatok feldolgozása előtt a `read` függvénnyel először `Int` elemtípusú listává alakítjuk a kiolvasott értéket, majd az így meghatározott listának a `length` függvénnyel kiszámítjuk az elemszámát.

A beolvasáskor feltételeztük, hogy a `prim.txt` tartalma a következő struktúrájú, tulajdonképpen az előző példánál létrehozott állományt dolgozzuk fel:

```
[2,3,5,7,11,13,17,19]
```

A számok tehát szögletes zárójelek között vannak, és vessző az elválasztójel. Más állományszerkezet esetén, hogy ne kapjunk futási hibát, módosítani kell a beolvasási algoritmuson.

A `readFile` és `writeFile` függvényekhez hasonlóan működik az `appendFile`, amely a paraméterként megadott állomány tartalmához hozzáfűz egy `String` típusú adatot. Ha az állomány nem létezik, akkor előbb létrehozza, ha már létezik, akkor a tartalmát meghagyja, és a végéhez fog hozzáfűzni, típusdeklarációja a következő:

```
appendFile :: FilePath -> String -> IO ()
```

A következőkben módosítjuk az Eratosztenész szitájának algoritmusát, alkalmazni fogjuk a `dropWhile` és `takeWhile` függvényeket. Az `n` és `m` közötti prímszámokat generáljuk ki, az új függvény az `eSzitaDT` lesz. A `primHozzafuz`-ben háromszor fogjuk alkalmazni az `eSzitaDT`-t, és a függvény által meghatározott listákat a `writeFile`, majd az `appendFile` segítségével a `prim1.txt` állományban egymás után fűzzük. A kiíratást a `myShow` függvény segítségével formázzuk, olyan értelemben, hogy egyenként alakítjuk át az értékeket `String` típusú adattá, és a számok közé szóközt, a *blokkok* közé két új sor jelet teszünk.

```
eSzitaDT :: (Integral a) => a -> a -> [a]
eSzitaDT n m = dropWhile (< n) $
   takeWhile ( <= m) (2 : szita [3, 5..])

myShow :: Show a => [a] -> String
myShow [] = "\n\n"
myShow (k : ve) = show k ++ " " ++ myShow ve

primHozzafuz :: IO ()
primHozzafuz = do
   let ls1 = eSzitaDT 100 200
   writeFile "prim1.txt" $ myShow ls1
   let ls2 = eSzitaDT 1100 1200
   appendFile "prim1.txt" $ myShow ls2
   let ls3 = eSzitaDT 2100 2200
   appendFile "prim1.txt" $ myShow ls3
```

A következő példák a `System.IO` könyvtárcsomag állománykezelő függvényeit fogják használni, segítségükkel bonyolultabb szerkezetű szövegállományok, bináris állományok is feldolgozhatók.

6.12. feladat Írjunk egy Haskell-függvényt, amely kigenerálja az első `n` Hamming-számot növekvő sorrendben, majd írjuk ki a számokat a `hamming.txt` állományba, minden sorba kiírva a szám sorszámát, majd szóközzel elválasztva a Hamming-számot.

A megoldás első felében az adatbeviteli és kiíratási lehetőségeket mutatjuk be, és csak azután térünk rá a Hamming-számok értelmezésére, illetve a Hamming-számok előállításához használt algoritmus magyarázatára.

```
import System.IO

foHamming :: IO ()
foHamming = do
   putStr "n = "
   temp <- getLine
   let n = read temp :: Int
   let lsH = take n hammingF
   outf <- openFile "hamming.txt" WriteMode
   myPutLs outf lsH 1
   hClose outf

myPutLs :: (Show a, Show b, Num b)
           => Handle -> [a] -> b -> IO ()
myPutLs outf [] _ = return ()
myPutLs outf (k : ve) i = do
       hPutStrLn outf $ show i ++ " " ++ show k
       myPutLs outf ve (i + 1)
```

A `foHamming` függvény keretén belül végezzük az adatok beolvasását és kiíratását, ez lesz a fő függvény. A `hammingF` függvény egy `Integer` típusú elemekből álló, végtelen listába kigenerálja a Hamming-számokat, amiből a `take` függvénnyel lekérünk `n` darabot. A kigenerált számokat a `myPutLs` függvény fogja kiírni formázva a `hamming.txt` állományba. Írás előtt az állományt meg kell nyitni, majd írás után be kell zárni. Ezeket az `openFile`, `hClose` könyvtárfüggvények végzik, amelyek a `System.IO` könyvtármodulban találhatók.

```
openFile :: FilePath -> IOMode -> IO Handle
hClose :: Handle -> IO ()
```

A típusdeklarációkból jól látszik, hogy az `openFile` esetében két bemeneti paraméternek kell értéket adni. Meg kell adni az állomány nevét/útvonalát, és meg kell adni a megnyitási módot, ami a fenti kódsor esetében a `WriteMode` konstans értéket jelentette, azaz írásra nyitottuk meg az állományt. Más lehetséges megnyitási módok esetében a következő konstansokkal dolgozhatunk: `ReadMode, AppendMode`, ahol az első az olvasási lehetőséget, a második a hozzáfűzést biztosítja az állományban. Kimenetként egy állományazonosítót kapunk.

A `myPutLs` rekurzívan dolgozza fel a Hamming-számokból álló listát, az állományba soronként ír, ahol a sorszámot, amelyet az `i` jelöl, és a Hamming-számot, amelyet a `k` jelöl, a `show` függvénnyel előbb átalakítjuk `String` típusú adattá. Az állományba való soronkénti írást a `hPutStrLn` függvénnyel végeztük, amelynek bemeneti paraméterként meg kell adni az állományazonosítót és a beírandó `String` típusú adatot, típusdeklarációja a következő:

```
hPutStrLn :: Handle -> String -> IO ()
```

A következő függvény hasonló a `hPutStrLn`-hez, csak nem tesz új sor jelet a kiírt szöveg végére. Mindkét függvény használatához szükséges a `System.IO` importálása.

```
hPutStr :: Handle -> String -> IO ()
```

A `myPutLs` függvényt megírhatjuk explicit rekurzió nélkül is, a `mapM_` függvény alkalmazásával, így tömörebb és átláthatóbb lesz a kód:

```
myPutLs_ :: (Show a, Show b)
             => Handle -> [a] -> [b] -> IO ()
myPutLs_ outf ls lsT = mapM_ auxF $ zip ls lsT
   where
   auxF (k, i) = hPutStrLn outf $ show i ++ " " ++ show k
```

A `myPutLs_` függvény bemeneti paraméterként az `1`-gyel kezdődő természetes számok végtelen listáját és a Hamming-számok listáját kapja, ahol a természetes számok a megfelelő sorszámok kiíratásához kellenek. A sorszámok és Hamming-számok összekapcsolását a `zip` függvénnyel oldottuk meg, így az `auxF` függvény paraméterként már csak egy számpárt kap, ahol az első érték a sorszám, a második érték a megfelelő Hamming-szám lesz.

Ezek alapján a `foHamming` függvényt is módosítjuk, ahol a `myPutLs_` függvényt fogjuk meghívni:

```
foHamming_ :: IO ()
foHamming_ = do
```

```
    putStr "n = "
    temp <- getLine
    let n = read temp :: Int
    let lsH = take n hammingF
    outf <- openFile "hamming.txt" WriteMode
    myPutLs_ outf lsH [1..]
    hClose outf
```

A `hammingF` függvény fogja meghatározni a Hamming-számok végtelen listáját. Egy Hamming-szám általános alakja: $2^n \cdot 3^n \cdot 5^n$, azaz a `2, 3, 5` és ezek többszöröseiből képzett számok növekvő sorrendje adja a Hamming-számok végtelen listáját. Az első `10` Hamming-szám a következő: `[1, 2, 3, 4, 5, 6, 8, 9, 10]`

Az algoritmus elvi működése, amely kigenerálja a Hamming-számokat, a következő:

- Az első Hamming-szám az `1`, ez alapján a `hammingF` függvény létrehozza az `1 : lsH` listát, amelyet minden egyes rekurzív híváskor további elemekkel bővít.
- Három listát határozunk meg, amelyeknek elemeit rendre a `2·h, 3·h`, illetve `5·h` összefüggéssel adjuk meg, ahol `h` a `hammingF` függvény által generált lista értékeit jelöli. Vegyük észre, hogy a három lista elemei növekvő sorrendben lesznek.
- Az így generált három listát összefésüljük úgy, hogy a többször előforduló elemeket csak egyszer tesszük be az eredménylistába. Az eredményt az `lsH` fogja jelölni.

```
hammingF :: [Integer]
hammingF = 1 : lsH
    where
    temp = hammingF
    lsH2 = [ 2 * h | h <- temp ]
    lsH3 = [ 3 * h | h <- temp ]
    lsH5 = [ 5 * h | h <- temp ]
    lsH = mergeH lsH2 lsH3 lsH5
```

A kigenerált listák összefésülését a `mergeH` végzi, ami először az `ls2` és `ls3` listákat fésüli össze, majd az `ls1`-et fésüli össze a kapott eredménylistával, ahol két lista összefésüléséhez a `merge` függvényt használtuk.

```
mergeH :: (Integral a) => [a] -> [a] -> [a] -> [a]
mergeH ls1 ls2 ls3 = merge ls1 $ merge ls2 ls3
    where
```

```
merge :: (Integral a) => [a] -> [a] -> [a]
merge ls1 [] = ls1
merge [] ls2 = ls2
merge ls1@(k1 : ve1) ls2@(k2 : ve2)
    | k1 < k2 = k1 : merge ve1 ls2
    | k1 == k2 = merge ls1 ve2
    | otherwise = k2 : merge ls1 ve2
```

6.13. feladat Írjunk egy Haskell-függvényt, amely a `hamming.txt` állományt beolvassa, majd kiírja tartalmát a képernyőre.

```
import System.IO

foHammingOlvas1 :: IO ()
foHammingOlvas1 = do
   inf <- openFile "hamming.txt" ReadMode
   myRead1 inf
   hClose inf

myRead1 :: Handle -> IO ()
myRead1 inf = do
   heof <- hIsEOF inf
   if heof then return ()
   else do
       temp <- hGetLine inf
       putStrLn temp
       myRead1 inf
```

A `foHammingOlvas1` függvény a `myRead1` függvény meghívásával az előző példánál létrehozott `hamming.txt` állomány tartalmát dolgozza fel. A `myRead1` függvény soronként olvassa be az állomány tartalmát, és a beolvasott sorokat kiírja a képernyőre, ahol az állományazonosító lesz a függvény bemeneti paramétere, kimeneti paraméterének típusa pedig `IO ()`. A feldolgozás során végig karakterláncokkal dolgozunk, nincs szükség átalakításra, mert az adatok feldolgozása csak annyiból áll, hogy kiíratjuk őket a képernyőre.

A `myRead1` függvényben a `hIsEOF` minden egyes sor beolvasása után megvizsgálja, hogy a paraméterként megadott állománynak nem az utolsó sorát olvastuk-e be. Ez a függvény `True`-t határoz meg, ha az állomány végén vagyunk, másképp `False`-t, típusdeklarációja a következő:

```
hIsEOF :: Handle -> IO Bool
```

A soronkénti beolvasást a `hGetLine` függvénnyel végeztük, amely bemeneti paraméterként egy állományazonosítót vár, és egy `IO String` értéket határoz meg. Figyeljük meg, hogy híváskor a `<-` műveleti jelet használtuk, mert a függvény által meghatározott érték típusa `IO String`.

```
hGetLine :: Handle -> IO String
```

6.14. feladat Írjunk egy Haskell-függvényt, amely a `hammming.txt` állományt beolvassa, és átteszi egy listába a Hamming-számokat, majd kiírja a képernyőre a lista tartalmát.

```
import System.IO

foHammingOlvas2 :: IO ()
foHammingOlvas2 = do
   inf <- openFile "hamming.txt" ReadMode
   hLs <- myRead2 inf
   print hLs
   hClose inf

myRead2 :: Handle -> IO [Integer]
myRead2 inf = do
   heof <- hIsEOF inf
   if heof then return []
   else do
       temp <- hGetLine inf
       let ls = words temp
       let [i, k] = map (read :: String -> Integer) ls
       ve <- myRead2 inf
       return (k : ve)
```

A `myRead2` függvény soronként olvassa be az állomány tartalmát, és annak érdekében, hogy létre tudja hozni a Hamming-számokból álló listát a sorszám-érték párokból álló sorokból, meghatározza `k`-ba a tulajdonképpeni Hamming-számot (ne felejtsük el, hogy az előző példánál az állományba sorszám-érték párokat írtunk ki), amely értékkel aztán a listaépítést végzi.

A `foHammingOlvas3` függvényben az állomány soronkénti feldolgozását a `lines` függvényt használva oldjuk meg.

```
foHammingOlvas3 :: IO ()
foHammingOlvas3 = do
   hLista <- readFile "hamming.txt"
   mapM_ (putStr . auxF) $ lines hLista
```

```
    putStrLn ""
    where
    auxF temp =
        let ls = map (read :: String -> Integer)
                     $ words temp
        in show (ls !! 1) ++ " "
```

Megjegyezzük, hogy a `lines` bemeneti paramétere egy `String` típusú érték, amelyet a függvény az új sor jelek menetén feldarabol, ahol az eredmény egy `String` elemtípusú lista lesz. Működését az alábbi lekérdezés is mutatja:

```
> lines "12\n123\n1234\n12345"
["12","123","1234","12345"]
```

6.15. feladat Írjunk egy Haskell-függvényt, amely nagybetűsíti az `input.txt` tartalmát, és az eredményt átteszi az `output.txt` állományba.

A feladatot háromféle módszerrel is megoldjuk, ahol mindhárom esetben egy sornak a nagybetűsítése a `map` és `toUpper` függvények segítségével történik. Mielőtt lekérdeznénk a függvényeket, ellenőrizzük, hogy az aktuális mappában szerepel-e egy `input.txt` állomány, nehogy futási hibát kapjunk. A Haskell kivételkezelési lehetőségeiről egy későbbi fejezetben lesz szó.

Az első módszer során az `input.txt` állomány tartalmát a Haskell automatikusan dolgozza fel.

```
import Data.Char (toUpper)
foNagybetusit1 :: IO ()
foNagybetusit1 = do
       inf <- readFile "input.txt"
       writeFile "output.txt" $ map toUpper inf
```

A második módszernél az állomány tartalmát soronként dolgozzuk fel.

```
import Data.Char (toUpper)
import System.IO
foNagybetusit2 :: IO ()
foNagybetusit2 = do
   inf <- openFile "input.txt" ReadMode
   outf <- openFile "output.txt" WriteMode
   feldolgoz inf outf
   hClose inf
   hClose outf
```

```
feldolgoz :: Handle -> Handle -> IO ()
feldolgoz inf outf = do
   feof <- hIsEOF inf
   if feof then return ()
   else do
       temp <- hGetLine inf
       hPutStrLn outf (map toUpper temp)
       feldolgoz inf outf
```

A harmadik változatban az adatok beolvasását szintén soronként végezzük, de használva a `lines` függvényt, sokkal tömörebb és átláthatóbb kódot kapunk. Az állományba az `appendFile` segítségével írunk, amelyet egy `mapM_` keretén belül hívunk meg, előtte azonban meghívjuk a `writeFile` függvényt, hogy az `output.txt` állomány tartalma minden egyes új futtatáskor üres legyen.

```
import Data.Char (toUpper)
foNagybetusit3 :: IO ()
foNagybetusit3 = do
   hLista <- readFile "input.txt"
   writeFile "output.txt" ""
   let ls = map auxF $ lines hLista
   mapM_ (appendFile "output.txt") ls
       where
       auxF temp = map toUpper temp ++ "\n"
```

A következő példákban bináris állományokkal végzünk műveleteket, elöljáróban azonban felhívjuk a figyelmet arra, hogy az operációs rendszerek közül a Windows másképp kezeli a szövegállományokat és másképp a bináris állományokat. Éppen ezért Windows alatt az `openFile` helyett bináris állomány megnyitására az `openBinaryFile` függvényt fogjuk használni. Egy Linux alapú rendszer esetében mind az `openFile`, mind az `OpenBinaryFile` hasonló feldolgozást tesz lehetővé, de a programkód hordozhatósága miatt a bináris állományok esetében itt is inkább az `openBinaryFile` függvényt használjuk. Típusdeklarációja a következő:

```
openBinaryFile :: FilePath -> IOMode -> IO Handle
```

6.16. feladat Írjunk egy Haskell-függvényt, amely kiírja egy szövegállományba egy tetszőleges típusú állomány bájtjainak hexa értékét.

```
import Data.Char (intToDigit, digitToInt, ord)
import System.IO
```

```
foHexa :: IO ()
foHexa = do
   infB <- openBinaryFile "file.pdf" ReadMode
   outT <- openFile "fileHexa.txt" WriteMode
   bStr <- hGetContents infB
   let bHexa = alakitHexa bStr
   hPutStr outT bHexa
   hClose infB
   hClose outT
```

A `foHexa` függvény a `file.pdf` binárisállomány hexa értékeit írja ki a `fileHexa.txt` szövegállományba. Természetesen a `file.pdf` helyett megadhatunk más állományt is. Futtatás előtt most is ellenőrizzük, hogy az aktuális mappában szerepel-e egy `file.pdf` állomány.

A kódsorban a Haskellre bíztuk az adatok beolvasását, azaz a `hGetContents` függvényt használtuk, amely mindig annyi bájtot olvas be az állományból, amennyire szükség van, típusdeklarációja a következő:

```
hGetContents:: Handle -> IO String
```

Az `alakitHexa` függvény fogja a beolvasott bájtokat, azaz a karakterek listáját átalakítani, felhasználva a korábban megírt `convFromNr` függvényt, illetve az `intToDigit` könyvtárfüggvényt. Az `intToDigit` a `Data.Char` könyvtárban található, és a bemeneti egész számot, ha az kisebb mint `16`, átalakítja hexa szimbólummá, ellenkező esetben hibaüzenetet ad. A függvénynek a `digitToInt` lesz a párja, amely a bemeneti karaktert, ha az egy hexa szimbólum, átalakítja a megfelelő egész számmá, ellenkező esetben hibaüzenetet ad.

```
> intToDigit 15
 'f'

> intToDigit 20
*** Exception: Char.intToDigit: not a digit 20...
```

```
alakitHexa :: [Char] -> [Char]
alakitHexa (k : ve) =
   if null ve then tempK
   else newK ++ alakitHexa ve
        where
        tempK = map intToDigit $ convFromNr (ord k) 16
        newK = tempK ++ " "
```

Az `alakitHexa` függvény forráskódját módosíthatjuk úgy, hogy egy másik könyvtárfüggvényt használunk a hexa szimbólumok meghatározásához. Ez a függvény a `showHex` lesz, amely a `Numeric` könyvtárcsomagban található. A `showHex` két értéket vár bemenetként, ahol az első egy `String` formátumban megadott nem negatív egész szám kell legyen, amit átalakít hexa alakba, a másik paraméter pedig üres `String` kell legyen.

```
> showHex 1024 ""
"400"
```

```
> showHex 10 ""
"a"
```

```
import Numeric (showHex)
alakitHexa_ :: [Char] -> [Char]
alakitHexa_ (k : ve) =
   if null ve then tempK
   else newK ++ alakitHexa_ ve
   where
   tempK = showHex (ord k) ""
   newK = tempK ++ " "
```

6.17. feladat Írjunk egy Haskell-függvényt, amely meghatározza egy állomány bájtméretét.

```
import System.IO
foMeret :: IO ()
foMeret = do
   putStr "Kerek egy allomany nevet: "
   fName <- getLine
   inf <- openBinaryFile fName ReadMode
   size <- hFileSize inf
   putStr "az allomany bajtmerete: "
   print size
   hClose inf
```

A `hFileSize` az állomány bájtméretét adja meg, szignatúrája a következő:

```
hFileSize :: Handle -> IO Integer
```

6.18. feladat Írjunk egy Haskell-függvényt, amely másolatot készít egy tetszőleges típusú állományról.

```
import System.IO
foMasol1 :: IO ()
foMasol1 = do
```

```
    inf <- openBinaryFile "file.pdf" ReadMode
    outf <- openBinaryFile "cFile.pdf" WriteMode
    bLs <- beolvasBajtok inf
    kiirBajtok outf bLs
    hClose inf
    hClose outf
```

A `beolvasBajtok` függvény bájtonként dolgozza fel a `file.pdf` állomány tartalmát. A program működéséhez szükséges tehát, hogy legyen egy ilyen nevű állomány az aktuális mappában. A függvény a beolvasott bájtokból létrehoz egy `bLs` listát, amit a `kiirBajtok` bemeneti paraméterként fog megkapni. A `kiirBajtok` a listaelemeket egyenként kiírja az `cFile.pdf` állományba, ahol a bájtok beolvasását, illetve az állományba való kiírást a `hGetChar, hPutChar` könyvtárfüggvények végzik. A típusdeklarációk a következők:

```
    hGetChar :: Handle -> IO Char
    hPutChar :: Handle -> Char -> IO ()

beolvasBajtok :: Handle -> IO [Char]
beolvasBajtok inf = do
    heof <- hIsEOF inf
    if heof then return []
    else do
        k <- hGetChar inf
        ve <- beolvasBajtok inf
        return (k : ve)

kiirBajtok :: Handle -> [Char] -> IO()
kiirBajtok out [] = return ()
kiirBajtok outf (k : ve) = do
    hPutChar outf k
    kiirBajtok outf ve
```

A `kiirBajtok` függvény megadható tömörebben is, a `mapM_` függvény segítségével:

```
kiirBajtok_ :: Handle -> [Char] -> IO()
kiirBajtok_ outf = mapM_ (hPutChar outf)
```

A feladat hatékonyabb megoldása azonban az, amikor az adatok beolvasását a Haskellre bízzuk:

```
foMasol2 :: IO ()
foMasol2 = do
```

```
    inf <- openBinaryFile "file.pdf" ReadMode
    outf <- openBinaryFile "cFile.pdf" WriteMode
    bLs <- hGetContents inf
    hPutStr outf bLs
    hClose inf
    hClose outf
```

6.19. feladat Írjunk egy Haskell-függvényt, amely összehasonlítja két bináris állomány tartalmát, azaz megállapítja azt az első pozícióértéket, ahol különbözik egy-egy bájt a két állományban.

```
import System.IO
foHasonlit1 :: IO ()
foHasonlit1 = do
   inf1 <- openBinaryFile "file1.pdf" ReadMode
   inf2 <- openBinaryFile "file2.pdf" ReadMode
   bInd <- hasonlitBajtok1 inf1 inf2 (-1) 0
   if bInd == -1 then
       putStrLn "OK, egyformak!"
   else do
       putStr "kulonboznek, pozicio: "
       print bInd
   hClose inf1
   hClose inf2

hasonlitBajtok1 :: Num a =>
                   Handle -> Handle -> a -> a -> IO a
hasonlitBajtok1 inf1 inf2 bInd bInd1 = do
   heof1 <- hIsEOF inf1
   heof2 <- hIsEOF inf2
   if heof1 && heof2 then return bInd
   else do
       if heof1 || heof2 then return bInd1
       else do
           nr1 <- hGetChar inf1
           nr2 <- hGetChar inf2
           if nr1 /= nr2 then return bInd1
           else
               hasonlitBajtok1 inf1 inf2 bInd (bInd1 + 1)
```

`A foHasonlit1` a feladat első módszer szerinti implementációjának a főfüggvénye, amelyben `hasonliBajtok1` a két bemeneti állományból beolvas egy-egy bájtot, majd összehasonlítja ezt a két értéket. A függvény bemeneti

paraméterként a két állomány azonosítóját és két további paramétert kap, amelyeket `bInd` és `bInd1`-el jelöltünk. A `bInd` kezdeti értéke `-1`, és ha ez az érték a vizsgálat során megmarad `-1`-nek, akkor az azt jelenti, hogy a két állomány egyforma. A `bInd1` azt az első pozícióértéket jelöli, ahol a két állomány bájtjai különböznek.

A második implementáció a fájlok tartalmának a beolvasására a `hGetContents` könyvtárfüggvényt használja, ahol a főfüggvény a `foHasonlit2` lesz.

```
foHasonlit2 :: IO ()
foHasonlit2 = do
   inf1 <- openBinaryFile "file1.pdf" ReadMode
   inf2 <- openBinaryFile "file2.pdf" ReadMode
   bstr1 <- hGetContents inf1
   bstr2 <- hGetContents inf2
   let bInd = hasonlitBajtok2 bstr1 bstr2 0
   if bInd == -1 then
       putStrLn "OK, egyformak!"
   else do
       putStr "kulonboznek, pozicion: "
       print bInd
   hClose inf1
   hClose inf2

hasonlitBajtok2 :: (Eq a1, Num a) =>
                    [a1] -> [a1] -> a -> a
hasonlitBajtok2 [] [] bInd = -1
hasonlitBajtok2 [] ve bInd = bInd
hasonlitBajtok2 ve [] bInd = bInd
hasonlitBajtok2 (k1: ve1) (k2: ve2) bInd
   | k1 /= k2 = bInd
   | otherwise = hasonlitBajtok2 ve1 ve2 (bInd + 1)
```

A `hasonlitBajtok2` függvény első két paramétere egy-egy `String` típusú érték lesz, azaz bemeneti paraméterként a két állományban levő bájtok egy-egy listáját adjuk meg. A harmadik paraméter kezdetben `0`, és az aktuális pozíció értékét jelöli. A `hasonlitBajtok2` függvény meghatározza azt a pozícióértéket, ahol eltér egymástól a két fájl, illetve ha a visszatérési érték `-1`, akkor az azt fogja jelenteni, hogy a két állomány tartalma egyforma.

6.20. feladat Írjunk egy Haskell-függvényt, amely titkosítja egy adott állomány tartalmát, alkalmazva a `xor` műveletet, majd fejtsük is vissza a rejtjelezett állományt.

A titkosításhoz egy titkos információt, egy kulcsot fogunk használni, ami megadott karakterek (bájtok) listáját jelenti, a helyes működés miatt pedig fontos, hogy ugyanazt a kulcsot használjuk mind a titkosításhoz, mind a visszafejtéshez.

```
import System.IO
import Data.Char
import Data.Bits

foTitkosit :: IO ()
foTitkosit = do
   let key = "FP_Key_BadXOR"
   titkositF "file.pdf" "crypt.pdf" key
   putStrLn "vege a titkositasnak!"
   titkositF "crypt.pdf" "nFile.pdf" key
   putStrLn "vege a visszafejtesnek!"

titkositF :: FilePath -> FilePath -> [Char] -> IO ()
titkositF nameIn namOut key = do
   inf <- openBinaryFile nameIn ReadMode
   outf <- openBinaryFile namOut WriteMode
   bLs <- hGetContents inf
   let eLs = titkositB bLs key
   hPutStr outf eLs
   hClose inf
   hClose outf

titkositB :: [Char] -> [Char] -> [Char]
titkositB ls key = map auxF $ zip ls (cycle key)
   where
   auxF (t1, t2) = chr $ xor (ord t1) (ord t2)
```

A `foTitkosit` függvényben a `file.pdf` állomány titkosítását a `titkositF` végzi, az eredmény a `crypt.pdf` lesz. A visszafejtést is a `titkositF` kiértékelésével valósítjuk meg, az eredmény ezúttal az `nFile.pdf`-be kerül.

A tulajdonképpeni titkosítást a `titkositB` függvény végzi, amely paraméterként megkapja `file.pdf` bájtjainak listáját és a `key` értékét. A

titkosítás annyiból fog állni, hogy a `file.pdf` bájtjai és a `key` lista bájtjai között alkalmazzuk az `xor` műveletet. A `key` elemeit körkörösen fogjuk venni, ami azt jelenti, hogy ha elfogytak a `key` listából az elemek, akkor újból a `key` első elemével folytatjuk, egészen addig, amíg a `file.pdf` bájtjain is végig nem mentünk. Vegyük észre, hogy a `zip` függvény alkalmazásával az állomány bájtjaiból és a kulcs értékeiből elempárokat építünk, ahol a `cycle` függvény alkalmazásával a kulcs körkörös használatát tesszük lehetővé.

Ha a `key` elemeit a lehetséges bájtértékek közül teljesen véletlenszerűen választanánk ki, és ha a `key` lista hossza megegyezne a `file.pdf` bájtméretével, illetve ha minden egyes titkosításhoz más `key` értéket használnák, akkor ez a titkosítási eljárás egy nagyon biztonságos módszer lenne. A gyakorlatban azonban mindezeket nagyon nehéz megvalósítani.

Emlékeztetőül megjegyezzük, hogy az `xor` művelet során mindig két bitértéket dolgozunk fel, ahol a lehetséges bemeneti `x, y` bitértékeket és az eredményt a következő táblázat mutatja:

x	y	xor
0	0	0
0	1	1
1	0	1
1	1	0

Az `xor` művelet alkalmazásához szükséges a `Data.Bits` könyvtár importálása, ahol a függvénynek két, az `Integral` típusosztályhoz tartozó értéket kell megadni, és a kimenet is az `Integral` típusosztályhoz fog tartozni. A `titkositB` függvényben az állomány és kulcs értékeire szükséges meghívni az `ord` függvényt, mielőtt alkalmaznánk az `xor` műveletet, majd a művelet végén alkalmazni kell az eredményre a `chr` függvényt.

Az `xor` művelet tulajdonságából adódik, hogy a titkosítás és visszafejtés algoritmusa megegyezik, csupán a bemeneti paraméterek fognak változni. Ezért lehetséges az, hogy mind a titkosításhoz, mind a visszafejtéshez a `titkositF` függvényt hívjuk meg.

A `titkositB` függvényben a `map` helyett használhatjuk a `zipWith` függvényt. A `zipWith` kimenete egy lista, amely a listaelemeket úgy hozza létre, hogy alkalmazza az `auxF` kétparaméteres függvényt a második és harmadik paraméterében megadott listaelemeken.

```
titkositB_ :: [Char] -> [Char] -> [Char]
titkositB_ ls key = zipWith auxF ls (cycle key)
    where
    auxF t1 t2 = chr $ xor (ord t1) (ord t2)
```

A következő két lekérdezés a `zipWith` használatát mutatja:

```
> zipWith (*) [1,2,3,4,5] [6,8,10,12]
[6,16,30,48]

> ls = ["Nyugati", "Keleti", "Deli"]
> zipWith (++) ls $ repeat "-Karpatok"
["Nyugati-Karpatok","Keleti-Karpatok","Deli-Karpatok"]
```

6.5. Kivételkezelés

A futási hibák kezelése a Haskellben is fontos szerepet játszik. Több lehetőség között is választhatunk, a fejezet keretén belül ezekre fogunk kitérni.

Az **`error`** függvényt már korábban is használtuk. Típusdeklarációja a következő, amiből jól látható, hogy bemeneti paraméterként egy karakterláncot, azaz a hibát leíró üzenetet kell megadni:

```
error :: [Char] -> a
```

Általában kisebb, informális programkódok esetében alkalmazzák, mint ahogy azt tettük az alábbi kódsorban is, amely nullával való osztáskor jelez hibát.

```
osztF :: Double -> Double -> Double
osztF _ 0 = error "Nullaval valo osztas!"
osztF x y = x / y

> osztF 2 3
0.6666666666666666

> osztF 3 0
*** Exception: Nullaval valo osztas!...
```

A **`catch`** függvény a `Control.Exception` könyvtárcsomagban van. Két bemeneti értéket vár, ahol az első paraméter típusa `IO a`, ami akcióknak egy olyan sorát jelenti, ahol az utolsó akció `a` típusú értéket ad. Ezen akciók valamelyikének a futási hibáját kell *elkapja* a `catch`. A második paramétere a kivételt kezelő függvény, kimeneti értékének típusa szintén `IO a`, szignatúrája a következő:

```
catch :: Exception e => IO a -> (e -> IO a) -> IO a
```

A következő példa a `catch` segítségével figyeli, hogy mit eredményez a `print` függvény meghívása. Ha a korábban megadott `osztF` függvény nullával való osztást végez, akkor a `print` sem tudja a megfelelő értéket kiírni, ezért a `kivetelKezolo` függvény kerül meghívásra.

```
import Control.Exception
foOszt :: Double -> Double -> IO()
foOszt x y = catch (print $ osztF x y) kivetelKezelo
   where
   kivetelKezelo :: SomeException -> IO ()
   kivetelKezelo err = do
       let ls = "Hiba jellege: " ++ show err
       putStrLn ls

>  foOszt 2 3
0.6666666666666666

>  foOszt 2 0
Hiba jellege: Nullaval valo osztas!...
```

A következő példában a `catch` függvény segítségével azt a futási hibát kezeljük le, amely akkor adódik, amikor meg akarunk nyitni egy nem létező állományt:

```
import Control.Exception
import System.IO

foAll :: IO()
foAll = catch ( do
       inf <- openFile "valami.jpg" ReadMode
       ls <- hGetContents inf
       putStrLn ls
       hClose inf
   ) kivetelKezelo
   where
   kivetelKezelo :: SomeException -> IO ()
   kivetelKezelo err = do
       let ls = "Hiba jellege: " ++ show err
       putStrLn ls

> foAll
Hiba jellege: valami.jpg: openFile: does not exist...
```

A következő példában a `catch` függvény segítségével azt a futási hibát kezeljük le, amikor a billentyűzetről történő beolvasás során adódik futási hiba, például karaktereket olvasunk valós számok helyett:

```
import Control.Exception

olvasDouble :: IO Double
olvasDouble = do
   str <- getLine
   return (read str :: Double)

foOlvas :: IO ()
foOlvas = catch ( do
       putStr "n = "
       n <- olvasDouble
       putStr "negyzetgyoke: "
       print $ sqrt n
   ) kivetelKezelo
   where
   kivetelKezelo :: SomeException -> IO ()
   kivetelKezelo err = do
       let ls = "Hiba jellege: " ++ show err
       putStrLn ls

> foOlvas
n = t
negyzetgyoke: Hiba jellege: Prelude.read: no parse
```

Hibák kezelésére a Haskell két típust is definiál, a **`Maybe`** és az **`Either`** típusokat.

A `Maybe` egy monád, tulajdonképpen a monádok között a legegyszerűbb. Segítségével *eredmény nélküli* számítást lehet kezelni. A standard könyvtárban van definiálva, típusdeklarációja a következő:

```
data Maybe a = Nothing
             | Just a
```

A `data` kulcsszóval a Haskell egy új típust, egy paraméterezett típust definiál. Ez azt jelenti, hogy ezzel a típusdeklarációval egyaránt lehetséges a `Maybe Int`, a `Maybe String`, a `Maybe [Double]` stb. típusok használata.

A fenti definíció alapján még az is megállapítható, hogy egy `Maybe` típusú adat kétféle értéket kaphat, a `Nothing` konstanst, illetve a `Just a` értéket, amelyeket értékkonstruktoroknak hívunk. Az értékkonstruktorok között kötelező a **`|`** szimbólum használata. A programozó hasonló módon

definiálhat paraméterezett típust, erről egy későbbi fejezetben bővebben is szó lesz.

A `Maybe` típussal futási hibákat kezelhetünk, hiba esetén `Nothing` lesz a meghatározott érték, ellenkező esetben pedig `Just x`, ahol `x a` típusú. Részlegesen értelmezett függvényeknél (partial functions), ha nincsenek minden esetben definiálva az argumentumértékek, akkor is egy `Maybe` típusú értékkel érdemes dolgozni.

A következő függvény a `head` függvény egy implementációja, amely ellentétben a beépített függvénnyel, nem fog futási hibát adni, ha a bemenet üres lista. Ekkor a függvény által meghatározott érték a `Nothing` lesz.

```
myHead1 :: [a] -> Maybe a
myHead1 [] = Nothing
myHead1 (k : ve) = Just k

> myHead1 "Bekas-szoros-Nagyhagymas Nemzeti Park"
Just 'B'

> myHead1 ""
Nothing

> head ""
*** Exception: Prelude.head: empty list
```

A következő függvényben a `Maybe` típust nullával való osztás kezelésére használjuk :

```
osztF1 :: Double -> Double -> Maybe Double
osztF1 _ 0 = Nothing
osztF1 x y = Just (x / y)

> osztF1 2 3
Just 0.6666666666666666

> osztF1 3 0
Nothing
```

A `foOszt1`-ben, a `case` kifejezés segítségével vizsgáljuk meg, hogy az `osztF1` függvénynek mi a kimeneti értéke, így megoldható, hogy csak a kiszámolt érték vagy a hibaüzenet kerüljön kiíratásra:

```
foOszt1 :: Double -> Double -> IO()
foOszt1 x y =
   case osztF1 x y of
```

```
        Nothing -> putStrLn "Nullaval valo osztas!"
        Just k -> print k

> foOszt1 2 3
0.6666666666666666

> foOszt1 3 0
Nullaval valo osztas!
```

6.21. feladat Írjunk egy Haskell-függvényt, amely meghatározza a következő összeget: $\frac{1}{x_1} + \frac{1}{x_2} + \cdots + \frac{1}{x_n}$, ahol az $x_1, x_2, \ldots x_n$ értékek a függvény lista típusú bemeneti paraméterének az elemei lesznek.

```
foOsszeg :: [Double] -> Maybe Double
foOsszeg ls =
   if null ls then return 0.0
   else do
        temp <- osztF1 1.0 $ head ls
        res <- foOsszeg $ tail ls
        return (temp + res)

> foOsszeg [1,2,6,5]
Just 1.8666666666666667

> foOsszeg [1,2,0,5,7]
Nothing
```

A `foOsszeg` függvény törzse nem egy funkcionális stílusban megírt kódsor, az `else` ág több akció végrehajtásából áll. Vegyük észre, hogy a `foOsszeg`, illetve `osztF1` függvények kimenetének típusa egy-egy `Maybe monád`, ezért az általuk meghatározott értékekhez csak úgy tudunk hozzáférni, ha a `<-` szimbólumot használjuk.

6.22. feladat Írjunk egy Haskell-függvényt, amely a `foldr` segítségével implementálja a könyvtárfüggvény `init`-et, azaz határozzuk meg azt a listát, amely tartalmazza a bemeneti lista elemeit, kivéve az utolsót.

```
myInit :: Foldable t => t a -> [a]
myInit ls =
   case temp of
        Nothing -> error "Ures lista!"
        Just k -> k
```

```
    where
    temp = auxI ls

auxI :: Foldable t => t a -> Maybe [a]
auxI = foldr op Nothing
    where
    op k Nothing = Just []
    op k (Just ve) = Just (k : ve)

> auxI ["bekas","nera","revi","tordai"]
Just ["bekas","nera","revi"]

> myInit ["bekas","nera","revi","tordai"]
["bekas","nera","revi"]
```

Figyeljük meg, hogy az `auxI` függvény a `foldr` kezdőértékét `Nothing`-ra állítja, ami a `foldr` használata miatt csak a legutolsó rekurzív híváskor lesz feldolgozva. Ekkorra már csak egy elem marad a bemeneti listában, amely esetben a bináris operátor definíciója szerint a függvény kimenete `Just []` lesz. Ez azt fogja eredményezni, hogy a bemeneti lista utolsó eleme nem kerül be az eredmény listába.

Az **`Either`** hasonlóan a `Maybe`-hez egy monád, típusdefiníciója a standard könyvtárban van és a következő:

```
data Either a b = Left a
                | Right b
            deriving (Eq, Ord, Read, Show)
```

A definícióból jól látható, hogy az `Ether` is paraméterezett típus, amelynek két értékkonstruktora van, az egyik `a` típusú lesz, amikor a `Left`-et használjuk, a másik `b` típusú lesz, ha a `Right`-t használjuk, azaz a `Left` és a `Right` által meghatározott értékek típusa különbözhet. Éppen ezért a hiba jellegére vonatkozóan a `Maybe`-hez képest több információt adhatunk meg. Általában a `Left`-et használják a hiba jelzésére, a `Right`-ot pedig a helyes eredmény megadásához. A definícióban, a `deriving` után azok a típusosztályok vannak feltüntetve, amelyek műveleteit használhatja egy `Either` típusú érték. A következő példa egy lista első elemét határozza meg:

```
myHead2 :: [a] -> Either String a
myHead2 [] = Left "Ures lista!"
myHead2 (k : ve) = Right k

> myHead2 [3.1415, 2.7182, 1.6180, 1.4142]
```

```
Right 3.1415

> myHead2 ""
Left "Ures lista!"
```

A következő példában a nullával való osztást kezeljük az `Either` típussal:

```
osztF2 :: Double -> Double -> Either String Double
osztF2 _ 0 = Left "Nullaval valo osztas!"
osztF2 x y = Right (x / y)

> osztF2 2 3
Right 0.6666666666666666

> osztF2 3 0
Left "Nullaval valo osztas!"
```

Hasonlóan, ahogy már korábban tettük, egy `foOszt2` főfüggvényben lekezelhetjük az `osztF2` kimenetét, és csak az eredményt vagy adott esetben a futási hibát jelezzük:

```
foOszt2 :: Double -> Double -> IO()
foOszt2 x y =
   case osztF2 x y of
       Left r -> putStrLn r
        r -> print r

> foOszt2 2 3
0.6666666666666666

> foOszt2 3 0
Nullaval valo osztas!
```

Az utolsó hibakezelési mód az **ioError**, illetve a **userError** függvények használatát mutatja be, amelyeknek típusdefiníciója a következő:

```
ioError :: IOError -> IO a
userError :: String -> IOError
```

A nullával való osztás hibakezelése a fenti két függvény használatával a következő lesz:

```
osztF3 :: Double -> Double -> IO Double
osztF3 x 0 = ioError (userError "Nullaval valo osztas!")
osztF3 x y = return (x / y)
```

```
foOszt3 :: Double -> Double -> IO ()
foOszt3 x y =
   catch (do
           k <- osztF3 x y
           print k
       ) kivetelKezelo
   where
   kivetelKezelo :: SomeException -> IO ()
   kivetelKezelo err = do
       let ls = "Hiba jellege: " ++ show err
       putStrLn ls

> foOszt3 2 0
Hiba jellege: user error (Nullaval valo osztas!)
```

6.6. Kitűzött feladatok

6.1. feladat Írjunk egy-egy Haskell-függvényt, amely kezeli a futási hibákat, beolvas a billentyűzetről `n` természetes számot, és kiírja a képernyőre

- `n`-ig a Fibonacci-számok listáját (`n > 50`),
- `n`-ig a prímszámok listáját, soronként `10` számot írva,
- az `n` `2`-es számrendszerbeli alakját úgy, hogy minden negyedik bit után tesz egy szóközt,
- az `n` `16`-os számrendszerbeli alakját úgy, hogy minden második hexa szimbólum után tesz egy szóközt, használjuk az `a, b, c, d, e, f` szimbólumokat,
- azokat a listaelemeket, amelyek megadják `n`-ig az összes természetes szám `2`-es számrendszerbeli alakját, például `n = 4`-re:

```
0, 1, 10, 11, 100
```

- a következő listaelemeket, a következőképpen tördelve:

```
(0,0)
(0,1) (1,0)
(0,2) (1,1) (2,0)
 ...
(0,n) (1,n-1) ... (n,0)
```

6.2. feladat Írjunk egy-egy Haskell-függvényt, amely kezeli a futási hibákat, beolvas egész számokat a billentyűzetről, majd meghatározza

- a legnagyobb számot,
- a legnagyobb szám előfordulási pozícióit,
- a beolvasott számok átlagát,
- minden szám előfordulási számát, például:

```
12,35,12,39,35,29,12,35,47,39,12,47,39,47-re:
        12: 4
        35: 3
        39: 3
        29: 1
        47: 3
```

- a számok előfordulási száma szerinti csoportokat, az előfordulási számok szerinti növekvő sorrend alapján, például:

```
12,35,12,39,35,29,12,35,47,39,12,47,39,47-re:
        1: 29
        3: 35, 39, 47
        4: 12
```

6.3. feladat Írjunk egy-egy Haskell-függvényt, amely kezeli a futási hibákat, beolvas két számot a billentyűzetről, és meghatározza:

- a két szám közötti számok összegét,
- a két szám közötti prímszámok összegét,
- a két szám közötti páratlan, összetett számok összegét,
- a két szám közötti azon számokat, amelyeknek a legtöbb a valódi osztója.

6.4. feladat Írjunk egy-egy Haskell-függvényt, amely kezeli a futási hibákat, és kiírja egy szövegállományba az `n` és `m` közötti

- négyzetszámokat,
- prímszámokat,
- páratlan összetett számokat,
- egész számok négyzetgyökét.

6.5. feladat Írjunk egy Haskell-függvényt, amely szövegállományban levő számok mindegyikére meghatározza a számok `b` számrendszerbeli alakját, a `b` számrendszerbeli alakban a nullások számát, illetve a `b` számrendszerbeli számjegyek számát, majd formázva átírja ezeket az adatokat egy másik szövegállományba. Kezeljük a futási hibákat.

6.6. feladat Írjunk egy Haskell-függvényt, amely szövegállományban levő számok mindegyikére meghatározza a számok prímosztóit. Kezeljük a futási hibákat.

6.7. feladat Írjunk egy Haskell-függvényt, amely megvizsgálja, hogy egy adott bájtszekvencia benne van-e egy bináris állományban. Kezeljük a futási hibákat.

6.8. feladat Írjunk egy Haskell-függvényt, amely meghatározza, hogy két bináris állományban milyen pozíciókon található különböző bájt. Kezeljük a futási hibákat.

6.9. feladat Írjunk egy Haskell-függvényt, amely a billentyűzetről olvas be állományneveket, és mindegyik állomány esetében, amely az aktuális mappában található, meghatározza annak bájtméretet. Kezeljük a futási hibákat.

6.10. feladat Írjunk egy Haskell-függvényt, amely a billentyűzetről olvas be állományneveket, és mindegyik olyan állományról, amely az aktuális mappában található, készít egy másolatot. Kezeljük a futási hibákat.

6.11. feladat Írjunk egy Haskell-függvényt, amely a billentyűzetről olvas be állományneveket és egész számokat. A program mindegyik állomány és egész szám esetében határozza meg az állomány első `n` bájtjának hexa értékét, ahol `n` rendre a beolvasott egész számok értékét jelöli. Az állományneveket és a meghatározott hexa értékeket formázva írjuk ki egy szövegállományba, azaz minden sorba egy állománynév, szóköz, majd a hexa értékek szóközökkel elválasztva kerüljenek.

6.12. feladat Határozzuk meg, hogy a $\texttt{file}_1, \ldots, \texttt{file}_n$ állományok közül melyek azok az állományok, amelyeknek bitmérete egy megadott értéknél nagyobb. A $\texttt{file}_1, \ldots, \texttt{file}_n$ állományneveket egy `String` elemtípusú listában adjuk meg. Kezeljük a futási hibákat.

6.13. feladat Határozzuk meg, hogy a $\texttt{file}_1, \ldots, \texttt{file}_n$ állományok közül melyek azok az állományok, amelyeknek első nyolc bájtjának hexa értéke megegyezik a következő hexa értékekkel: `25 50 44 46 2d 31 2e 35`, azaz melyek a pdf állományok. A $\texttt{file}_1, \ldots, \texttt{file}_n$ állományneveket egy `String` elemtípusú listában adjuk meg. Kezeljük a futási hibákat.

6.14. feladat Határozzuk meg, hogy a $\texttt{file}_1$, ..., $\texttt{file}_n$ állományok közül melyik állomány mérete a legnagyobb. Ha több ilyen állomány van, akkor mindegyiket határozzuk meg. A $\texttt{file}_1$, ..., $\texttt{file}_n$ állományneveket egy `String` elemtípusú listában adjuk meg. Kezeljük a futási hibákat.

6.15. feladat Írjunk egy Haskell-függvényt, amely egy szövegállományban levő hexa értékek alapján meghatározza a hexa értékekből felépíthető bináris állományt, azaz írjuk meg az `5.16.` megoldott feladat inverzét.

6.16. feladat Írjunk egy Haskell-függvényt, amely titkosítja a paraméterként megadott `String`-et, alkalmazva az `xor` műveletet, a `String` karakterei és a paraméterként megadott kulcs értékei között, úgy, ahogy azt az elméleti részben korábban leírtuk. A titkosított bájtokat írjuk ki hexadecimális formába, jelöljünk minden bájtot két hexa szimbólummal, tegyünk minden két hexa szimbólum közé egy szóközt, és írjunk egy sorba `16` bájtot. Az ellenőrzéshez írjuk meg azt a függvényt, amelyik visszafejti a hexa stringet.

A következő sorok példát adnak a függvényhívásra és a kiértékelés eredményére:

```
> ls = "sapientia marosvasarhelyi tudomanyegyetem"
> key = "STR_Key"
> foTitkositStr ls key
a titkositott ertek:
   20 35 22 36 2e 0b 0d 3a 35 72 32 2a 17 16 20 22
   33 2c 2a 17 11 36 38 2b 36 6b 11 c 37 3b 3f 3e
   25 1c 1c 34 2d 37 2b 2e 8
```

7. fejezet

Típusok és adatszerkezetek

7.1. Rekord típusok

A korábban megismert adattípusok mellett a Haskellben lehetőség van rekord típusok definiálására, amelyeknek egészen bonyolult szerkezete is lehet. Az új adattípus definiálását a `data` kulcsszó használatával valósíthatjuk meg, ahol a választott név nagybetűvel kell kezdődjön.

```
data DiakT = DiakE String Int [Double]
   deriving(Show)
```

A fenti definícióban a `DiakT` lesz az új típusú adatszerkezet neve, amelyet típuskonstruktornak hívunk. A `DiakE` azonosító az értékkonstruktor lesz, ezt használjuk, amikor egy `DiakT` típusú adatnak értéket akarunk adni. A `String, Int, [Double]` a mezők típusát határozzák meg. A `deriving` kulcsszóval azt érjük el, hogy az újonnan definiált típusra is alkalmazhatók lesznek a feltüntetett típusosztályban definiált műveletek, jelen esetben a `Show` típusosztály `show` függvényét fogjuk tudni használni `DiakT` típusú értékekre. Természetesen több típusosztályt is meg lehet adni a `deriving` specifikálásakor.

A gyakorlatban a típus- és értékkonstruktorok gyakran azonos nevet kapnak, ez alapján újradefiniáljuk a fenti adatszerkezetet, és a továbbiakban így fogunk dolgozni.

```
data Diak = Diak String Int [Double]
   deriving(Show)
```

Egy ilyen típusú adatnak, ha értéket szeretnénk adni, akkor választanunk kell egy nevet, legyen ez a `diák`, és figyeljünk arra, hogy ez mindig kisbetűvel kezdődjön:

```
diak = Diak "David" 5 [7.90,9.85,9.95,8.55]
```

A `Diak` értékkonstruktor argumentumaként megadott értékek típusa meg kell egyezzen a típus definiálásakor megadott mezők típusával, ahol természetesen a sorrendet is be kell tartani, ellenkező esetben fordítási hibát kapunk:

```
diak = Diak 5 "David" [7.90,9.85,9.95,8.55]
...
• Couldn't match expected type 'Int' with actual type ...
```

A mezőkre való hivatkozást mintaillesztéssel lehet megoldani. A következő kódsor kiírja a `diak`-ban megadott nevet és a legnagyobb jegyet, ahol a `nev, kod, jegy` azonosítók helyett természetesen választhattunk volna más neveket is.

```
myShowDiak :: Diak -> String
myShowDiak k = nev ++ " " ++ show (maximum jegy)
   where
   Diak nev _ jegy = k

> putStrLn $ myShowDiak diak
David 9.95
```

A mintaillesztést a következőképpen is meg lehet oldani:

```
myShowDiak_ :: Diak -> String
myShowDiak_ (Diak nev _ jegy) = nev ++ " "
                                ++ show (maximum jegy)
```

Egy `Diak` elemtípusú lista inicializálása esetén a következőképpen kell eljárnunk, ahol figyeljünk arra, hogy vesszőt csak a listaelemek közé tegyünk, mert a mezőértékek között tilos a vesszők használata:

```
lsD = [ Diak "Ferenc" 1 [7.5,9.25],
        Diak "Katalin" 2 [7.75,6.25,10,7.55],
        Diak "Maria" 3 [6.5,8.25,9.33],
        Diak "Zsuzsa" 4 [7.33,8.25,9.75],
        Diak "David" 5 [7.90,9.85,9.95,8.55]]
```

A `myShowDiakLs` függvény az `lsD`-ben szereplő adatokat dolgozza fel, segítségével kiíratjuk soronként a diákok neveit és a legnagyobb jegyüket:

```
myShowDiakLs :: [Diak] -> String
myShowDiakLs [] = ""
myShowDiakLs (k : ve) = temp ++ myShowDiakLs ve
   where
   Diak k1 k2 k3 = k
   temp = k1 ++ " " ++ show (maximum k3) ++ "\n"

> putStr $ myShowDiakLs lsD
Ferenc 9.25
Katalin 10.0
Maria 9.33
Zsuzsa 9.75
David 9.95
```

A `mapM_` függvényt is alkalmazhatjuk a diáknevek és legnagyobb jegy kiíratására, ekkor a kód is tömörebb lesz, ahogy azt a következő `myPrintDiakLs` függvényben láthatjuk. A `mapM_` első paramétere egy függvénykompozíció lesz, ahol az `auxF` a megfelelő mezők `String` típussá való átalakítását, illetve összefűzését végzi.

```
myPrintDiakLs :: [Diak] -> IO()
myPrintDiakLs = mapM_ (putStr . auxF)
   where
   auxF :: Diak -> String
   auxF (Diak k1 k2 k3) =
       k1 ++ " " ++ show (maximum k3) ++ "\n"

> myPrintDiakLs lsD
Ferenc 9.25
Katalin 10.0
...
```

A kifejezőbb adatszerkezet-nevek definiálása végett, a `type` kulcsszóval típusszinonimákat fogunk létrehozni, és ezeket fogjuk használni az új adatszerkezetek definiálásánál. Például a korábban definiált `Diak`-ot a következőképpen érdemes módosítani:

```
type Nev = String
type Kod = Int
type Jegyek = [Double]

data DiakM = DiakM Nev Kod Jegyek
               deriving(Show)
```

Fontos megjegyeznünk, hogy a Haskell nem engedi meg, hogy az ugyanolyan szerkezetű, de más nevű adatszerkezetek használatát összekeverjük. Ha módosítjuk a `myShowDiak`, korábban megírt függvényünk szignatúráját, a `diak` érték inicializálását viszont változatlanul hagyjuk, akkor futási hibát kapunk:

```
myShowDiak :: DiakM -> String
myShowDiak k = nev ++ " " ++ show (maximum jegy)
   where
   DiakM nev _ jegy = k

 diak = Diak "David" 5 [7.90,9.85,9.95,8.55]

> myShowDiak diak
... error:
Couldn't match expected type 'DiakM' with actual type ...
```

Az adatszerkezetek definiálása történhet mezőnevek megadásával is, ahol a mezőnevek mindig kisbetűvel kell kezdődjenek. A következő `Hallgato` adatszerkezet, melyben mezőneveket is megadunk, hasonló lesz a korábban definiált `Diak` adatszerkezethez:

```
type Nev = String
type Jegy = Double
type Ev = Int

data Hallgato = Hallgato{
   hNev :: Nev,
   hJegy :: Jegy,
   hEv :: Ev
} deriving (Show)
```

Ha egy mező értékét akarjuk módosítani, akkor a következőképpen kell eljárnunk:

```
hallgatoA = Hallgato "Mari" 4.5 1

modosit :: Hallgato -> Double -> Int -> Hallgato
modosit hallgato j e = hallgato {hJegy = j, hEv = e}
```

A következő lekérdezés után a `hallgatoA1` a `hallgatoA` módosított értékét fogja jelölni.

```
> hallgatoA1 =  modosit hallgatoA 8.7 2
> hallgatoA1
Hallgato {hNev = "Mari", hJegy = 8.7, hEv = 2}
```

A következőkben egyszerű algoritmusokat adunk meg egy `Hallgato` elemtípusú listán, ennek érdekében konstans értékekkel inicializáljuk a `hallgatoL` listát:

```
hallgatoL :: [Hallgato]
hallgatoL =
   [Hallgato "Sari" 8.75 1, Hallgato "Mari" 4.25 1,
    Hallgato "Feri" 3.5 2, Hallgato "Zsuzsi" 10.0 2,
    Hallgato "Laci" 8.5 2, Hallgato "Lori" 7.5 2]
```

7.1. feladat Írjunk egy Haskell-függvényt, amely egy `Hallgato` elemtípusú lista esetében megszámolja, hogy hány diáknak van átmenőjegye.

A `szamol_` függvény explicit rekurziót használ, míg a `szamol` a `filter` segítségével kiválogatja a listából azokat az elemeket, ahol a jegy mező értéke nagyobb `4.5`-nél, majd az így kapott listának meghatározza az elemszámát.

```
szamol_ :: [Hallgato] -> Int
szamol_ [] = 0
szamol_ (k : ve)
   | hJegy k > 4.5 = 1 + szamol_ ve
   | otherwise = szamol_ ve

szamol :: [Hallgato] -> Int
szamol ls = length $ filter (\k -> hJegy k > 4.5) ls

> szamol hallgatoL
4
```

A `szamol` függvény megadható a `foldr`, illetve a `foldl'` függvények alkalmazásával:

```
szamolFoldR :: [Hallgato] -> Int
szamolFoldR = foldr op 0
   where
   op k res = if hJegy k > 4.5 then res + 1 else res

import Data.List (foldl')
szamolFoldL :: [Hallgato] -> Int
szamolFoldL = foldl' op 0
   where
   op res k = if hJegy k > 4.5 then res + 1 else res
```

7.2. feladat Írjunk egy Haskell-függvényt, amely egy `Hallgato` elemtípusú lista esetében kiválogatja az elsőéves személyeket.

A `valogat_` függvény explicit rekurziót használ, míg a `valogat` a könyvtárfüggvény `filter` segítségével oldja meg a feladatot:

```
valogat_ :: Int -> [Hallgato] -> [Hallgato]
valogat_ e [] = []
valogat_ e (k : ve)
   | hEv k == e = k : valogat_ e ve
   | otherwise = valogat_ e ve

valogat :: Int -> [Hallgato] -> [Hallgato]
valogat e = filter (\k -> hEv k == e)

> valogat 1 hallgatoL
[Hallgato {hNev = "Sari", hJegy = 8.75, hEv = 1},...
```

Mindkét `valogat` függvényben az első paraméter, az `e` fogja jelölni, hogy hányadéves diákokat szeretnénk kiválogatni, a második paraméterben pedig egy `Hallgato` elemtípusú listát kell megadni.

A következőkben a `valogat` függvény implementációját is megadjuk, a `foldr` függvény segítségével, illetve az eredmény lista kiíratását is módosítjuk, formázva végezzük. A kiíratáshoz a `putStrLn` függvényt használjuk, ahol a formázást a `myShow` függvény végzi.

```
valogatFold :: Ev -> [Hallgato] -> IO ()
valogatFold e ls = mapM_ (putStrLn . myShow) nLs
   where
   nLs = foldr op [] ls
   op :: Hallgato -> [Hallgato] -> [Hallgato]
   op k rLs = if hEv k == e then k : rLs else rLs

myShow :: Hallgato -> String
myShow k = hNev k ++ " " ++ show (hJegy k) ++ " "
           ++ show (hEv k)

> valogatFold 1 hallgatoL
Sari 8.75 1
Mari 4.25 1
```

Az érdekesség kedvéért a `valogat` függvényt halmazkifejezések segítségével is megadjuk:

```
valogatLC :: Ev -> [Hallgato] -> IO()
valogatLC e ls = mapM_ (putStrLn . myShow) nLs
   where
   nLs = [k | k <- ls, hEv k == e]
```

7.3. feladat Írjunk egy Haskell-függvényt, amely egy `Hallgato` elemtípusú lista esetében meghatározza a jegyek átlagát.

Több megoldás közül is választhatunk. Korábban már számoltunk átlagot, a `sum` és `length` függvényekkel. Most fontosnak tartjuk megjegyezni, hogy a `sum` és `length` függvények használata miatt kétszer is bejártuk a listát, ahelyett, hogy egyszer jártuk volna be. A következőkben ezek használata nélkül oldjuk meg a feladatot, és csak egyszer megyünk végig a listaelemeken, hatékonyabb, de ugyanakkor kevésbé egyszerű megoldáshoz jutva.

Az algoritmus azon a gondolatmeneten alapszik, hogy az `auxAtlag` segédfüggvény kimenete egy pár, ahol az első érték az elemek összegét, míg a második az elemek számát határozza meg.

```
atlagH :: [Hallgato] -> Jegy
atlagH ls = ossz / db
   where
   (ossz, db) = auxAtlag ls
   auxAtlag :: [Hallgato] -> (Jegy, Double)
   auxAtlag ls
       | null ls = (0.0, 0.0)
       | otherwise = (x1 + hJegy k, x2 + 1)
           where
           (x1, x2) = auxAtlag ve
           k = head ls
           ve = tail ls

> atlagH hallgatoL
7.083333333333333
```

A feladat a `foldr` függvény használatával is megoldható, a kódsor a következő:

```
atlagHFoldR :: [Hallgato] -> Jegy
atlagHFoldR ls = ossz/ db
   where
   (ossz, db) = auxAtlag ls
   auxAtlag :: [Hallgato] -> (Double, Double)
   auxAtlag ls = foldr op (0.0, 0.0) ls
```

```
        where
        op k (x1, x2) = (x1 + hJegy k, x2 + 1)
```

A következő kódsor is egy `Hallgato` elemtípusú listában számol átlagot, a `hJegy` mező alapján. Most is egyszer járjuk be a listaelemeket, az algoritmus azonban más. Az `auxAtlag` függvénynek az előzőhöz képest kettővel több bemeneti paramétere van, az egyikben a jegyek összegét, a másikban a jegyek számát számoljuk ki, mindkettő kezdetben `0`. Ennél a megoldásnál a számításokat akkor végezzük, amikor *megyünk be* a rekurzióba.

```
atlagH_ :: [Hallgato] -> Jegy
atlagH_ ls = ossz / db
   where
   (ossz, db) = auxAtlag ls 0.0 0.0
   auxAtlag :: [Hallgato] -> Jegy -> Double
               -> (Jegy, Double)
   auxAtlag [] x1 x2 = (x1, x2)
   auxAtlag (k : ve) x1 x2 =
           auxAtlag ve (hJegy k + x1) (1 + x2)
```

A fenti gondolatmenet alapján, a `foldl'` függvény használatával a kódsor a következő:

```
atlagHFoldL :: [Hallgato] -> Jegy
atlagHFoldL ls = ossz/ db
   where
   (ossz, db) = auxAtlag ls
   auxAtlag :: [Hallgato] -> (Double, Double)
   auxAtlag ls = foldl' op (0.0, 0.0) ls
       where
       op (x1, x2) k = (x1 + hJegy k, x2 + 1)
```

7.4. feladat Egy cég egy adott alkalmazottról a következő adatokat tárolta el: név, év-jövedelem értékpárok. Írjunk egy Haskell-függvényt, amely a következő `Alkalmazott` típust használva, meghatározza egy megadott évre minden alkalmazott jövedelmét, egy konstans értékekkel inicializált `lsAlk` listában.

```
type Jovedelem = (Int, Int)
data Alkalmazott = Alkalmazott {
   alkNev :: String,
   alkJovedelem :: [Jovedelem]
   } deriving (Show)
```

```
lsAlk = [
 Alkalmazott "KissCsaba"
   [(2012,86000),(2013,89000),(2014,87000)],
 Alkalmazott "SzaboJanos"
   [(2010,180000),(2011,210000),(2013,200000),(2014,240000)],
 Alkalmazott "NagySandor"
   [(2011,220000),(2012,230000),(2013,190000),(2014,240000)],
 Alkalmazott "KovacsMaria"
   [(2013,99000),(2014,98000)],
 Alkalmazott "JozsaIstvan" [(2012,230000), (2014, 97000)]
 ]
```

Két függvényt fogunk írni, ahol az `auxEv` függvény megvizsgálja, hogy egy `Jovedelem` elemtípusú listában, egy adott évre vonatkozóan van-e jövedelem, a másik a `foJovedelem` főfüggvény, amely meghatározza a feladat által kért adatokat.

Az `auxEv` függvény, ha lehetséges, akkor meghatározza az évre vonatkozó jövedelmet, ellenkező esetben `Nothing` lesz a kimeneti értéke, ezért az `auxEv` függvény kimenetének típusa `Maybe Int` lesz.

```
auxEv :: Int -> [Jovedelem] -> Maybe Int
auxEv ev = foldr (op ev) Nothing
   where
   op ev (k1, k2) res =
       if ev == k1 then Just k2 else res

> auxEv 2013 [(2012,86000),(2013,89000),(2014,87000)]
Just 89000

> auxEv 2010 [(2012,86000),(2013,89000),(2014,87000)]
Nothing
```

Megadjuk az `auxEv` függvénynek az `auxEv_` módosított változatát is, ahol explicit rekurziót használunk. Ez a változat segíthet jobban megérteni az `auxEv` függvény működését. Az `auxEv_` függvény kimenetének típusát is módosítottuk úgy, hogy visszatérési értéke `-1` legyen, ha az évre vonatkozóan nincs jövedelem nyilvántartva, ellenkező esetben pedig a nyilvántartott jövedelmet határozza meg a függvény.

```
auxEv_ :: Int -> [Jovedelem] -> Int
auxEv_ _ [] = -1
auxEv_ ev ((k1, k2): ve)
   | ev == k1 = k2
```

```
    | otherwise = auxEv_ ev ve

> auxEv_ 2013 [(2012,86000),(2013,89000),(2014,87000)]
89000

> auxEv_ 2010 [(2012,86000),(2013,89000),(2014,87000)]
-1
```

A `foJovedelem` főfüggvénynek paraméterként megadjuk a vizsgálandó év értékét és az alkalmazottak listáját. Az eredmények kiíratását a `putStrLn` függvénnyel a `mapM_ auxF` paraméterében végezzük. Ugyancsak a `mapM_` segítségével oldjuk meg, hogy minden egyes alkalmazott esetében meghívásra kerüljön az `auxEv` függvény, amelynek az eredményét egy `case`-ben elemezzük:

```
foJovedelem :: Int -> [Alkalmazott] -> IO()
foJovedelem ev = mapM_ (auxF ev)
   where
   auxF :: Int -> Alkalmazott -> IO()
   auxF ev k =
       case res of
           Just x -> putStrLn $
               alkNev k ++ ", " ++ show x
           Nothing -> putStrLn $
               alkNev k ++ ", nincs jovedelem"
       where
       res = auxEv ev (alkJovedelem k)

> foJovedelem 2010 lsAlk
KissCsaba, nincs jovedelem
SzaboJanos, 180000
NagySandor, nincs jovedelem
KovacsMaria, nincs jovedelem
JozsaIstvan, nincs jovedelem
```

7.5. feladat Írjunk egy Haskell-függvényt, amely a fent megadott `Alkalmazott` elemtípusú lista esetében meghatározza egy megadott évre a maximális jövedelmet és azon alkalmazottakat, akiknek maximális volt a jövedelme.

A feladat főfüggvénye a `foMaxJovedelem`, amelyben meghívásra kerül a tulajdonképpeni maximum keresést végző `maximumAlk` függvény, illetve amelyben az eredmények formázott kiíratása is megtörténik.

```
foMaxJovedelem :: Int -> [Alkalmazott] -> IO()
foMaxJovedelem ev ls = do
   let (mLs, max) = maximumAlk ev ls
   case max of
       -1 -> putStrLn "nincs jovedelem"
       _-> do
         putStrLn $ "maximalis jovedelem: " ++ show max
         mapM_ putStrLn mLs
```

A `maximumAlk` kimenete egy tuple típusú érték, ahol az `mLs`-ben azoknak az alkalmazottaknak a neveit határozza meg a függvény, akiknek az adott évben maximális volt a jövedelme. A `max` a maximális jövedelmet jelöli, amely azonban `-1` lesz, ha az adott évben senkinek sincs jövedelem megadva. A `maximumAlk` a `foldr` függvényt használva fogja a listaelemeket feldolgozni, ahol a kezdeti értéket a `res` jelöli, a bináris operátort pedig az `op`. Az `op` első paramétere a lista aktuális eleme lesz, míg a második paramétere egy kételemű tuple, ahol a tuple első eleme a kiválasztásra kerülő alkalmazottak neveinek listája lesz (ez kezdetben üres lista), a második pedig a maximális jövedelem értéke (ez kezdetben -1). Az aktuális listaelem, azaz az aktuális alkalmazott adott évbeli jövedelemértékének a kiválasztását a korábban megírt `auxEv` függvénnyel oldottuk meg, ahol egy `case` kifejezésben kezeljük a `Nothing` és `Just` kimeneteket.

```
maximumAlk :: Int -> [Alkalmazott] -> ([String], Int)
maximumAlk ev ls = (mLs, max)
   where
   (mLs, max) = foldr op res ls
   res = ([], -1)
   op kAlk t
       | k == m = (nev: nevLs, m)
       | k < m = (nevLs, m)
       | k > m = ([nev], k)
           where
           (nevLs, m) = t
           nev = alkNev kAlk
           temp = auxEv ev (alkJovedelem kAlk)
           k = case temp of
                   Nothing -> -1
                   Just x -> x

> foMaxJovedelem 2012 lsAlk
maximalis jovedelem: 230000
```

```
NagySandor
JozsaIstvan
```

7.6. feladat Írjunk egy Haskell-függvényt, amely egy `Alkalmazott` elemtípusú listában minden alkalmazott esetében meghatározza azt az évet, amikor a jövedelme a legnagyobb volt.

A feladat főfüggvénye a `foMaxJovedelem`, amelyben egy `mapM_`-ben kiírjuk az alkalmazottak neveit, illetve az `auxMaxJ` függvény által meghatározott év, jövedelem értékeket tartalmazó tuple első elemét, azaz a meghatározott év értékét. Az `auxMaxJ` függvény paraméterként egy `Jovedelem`, azaz egy kételemű tuple elemtípusú listát kap, és maximumot számol a második elem, azaz a jövedelem értéke szerint.

```
foMaxJovedelemEv :: [Alkalmazott] -> IO()
foMaxJovedelemEv = mapM_ auxF
   where
   auxF k = print (alkNev k,
                   fst $ auxMaxJ (alkJovedelem k))

auxMaxJ :: [Jovedelem] -> Jovedelem
auxMaxJ = foldr op res
   where
   res = (0, -1)
   op k res
       | snd k > snd res = k
       | otherwise = res

> foMaxJovedelemEv lsAlk
("KissCsaba",2013)
("SzaboJanos",2014)
("NagySandor",2014)
("KovacsMaria",2013)
("JozsaIstvan",2012)
```

A fejezet további részében a rendezési algoritmusokat tárgyaljuk újra, példákon keresztül mutatjuk be, hogy a különböző típusú adatszerkezetek esetében hogyan tudjuk őket használni.

7.7. feladat Egy adott telefonról a következő adatok vannak eltárolva: név, eladási adatok. Minden telefon esetében az eladási adatok egy érték-hármasból álló listát jelentenek, ahol az érték-hármas tartalmaz egy évszámot, az adott évhez tartozó telefonárat és az eladott telefonok számát.

Írjunk egy Haskell-függvényt, amely rendez egy `Telef` elemtípusú listát, a telefonnevek alapján ábécésorrendbe.

```
data Telef = Telef{
    tNev :: String,
    tEladas:: [(Int, Int, Int)]
    } deriving (Show)
```

Megadunk egy konstans értékekkel feltöltött `Telef` elemtípusú listát, mert az algoritmusokat erre a listára fogjuk meghívni:

```
lsT = [
    Telef "HuaweiP20" [(2017,1800,10),(2018,1600,20)],
    Telef "SamsungGalaxyS9" [(2018,3500,25)],
    Telef "HuaweiP10"
        [(2016,1200,20),(2017,1000,15),(2018,950,18)],
    Telef "Lenovo" [(2013,600,5),(2018,450,12)],
    Telef "iPhone"
        [(2016,2000,9),(2017,2300,15),(2018,2400,6)],
    Telef "SamsungGalaxyJ5" [(2013,700,10),(2018,900,15)]]
```

A feladatot gyorsrendezéssel oldjuk meg, ahol figyeljük meg, hogyan érjük el az adatszerkezet mezőit.

```
quickT :: [Telef] -> [Telef]
quickT [] = []
quickT (k : ve) = quickT kLs ++ [k] ++ quickT nLs
   where
   kLs = [x | x <- ve, tNev x < tNev k]
   nLs = [x | x <- ve, tNev x >= tNev k]

> quickT lsT
[Telef {tNev = "HuaweiP10", tEladas = [(2016,1200,20)...]}
...
```

Ha elegáns kiíratást szeretnénk, akkor még szükségünk lesz a következő függvényre is:

```
showTelef :: [Telef] -> String
showTelef [] = ""
showTelef (k : ve) = temp ++ "\n" ++ showTelef ve
   where
   temp = tNev k ++ "\n" ++ showElad (tEladas k)

   showElad :: [(Int, Int, Int)] -> String
```

```
    showElad [] = " "
    showElad (k : ve) = showTuple k ++ "\n" ++ showElad ve

showTuple :: (Int, Int, Int) -> [Char]
showTuple (k1, k2, k3) = show k1 ++ " " ++
                         show k2 ++ " " ++
                         show k3

> putStr $ showTelef $ quickT lsT
HuaweiP10
2016 1200 20
2017 1000 15
2018 950 18
...
```

A `showTelef` függvény megoldható explicit rekurzió nélkül is, a `concatMap` függvényt használva:

```
showTelef_ :: [Telef] -> String
showTelef_ = concatMap auxT
   where
   auxT :: Telef -> [Char]
   auxT k = tNev k ++ "\n" ++
            showElad (tEladas k) ++ "\n"

   showElad :: [(Int, Int, Int)] -> String
   showElad = concatMap auxE
       where
       auxE k = showTuple k ++ "\n"
```

A `concatMap` a `Prelude`-ben van definiálva, az első paraméterként megadott függvényt alkalmazza a listaelemekre, majd a kapott elemeket egymás után fűzi. A következő lekérdezések és az eredmények segítenek megérteni a működését:

```
> ls = ["Nera-szurdok", "Nemzeti", "Park"]
> concatMap (++ " ") ls
"Nera-szurdok Nemzeti Park "

> ls = ["nera-szurdok", "nemzeti", "park"]
> auxC xLs = ' ' : (toUpper . head) xLs : tail xLs
> concatMap auxC ls
" Nera-szurdok Nemzeti Park"
```

7.8. feladat Írjunk egy Haskell-függvényt, amely rendez egy `Telef` elemtípusú listát, a telefonok összeladási darabszáma alapján.

A rendezést most is a gyorsrendezés algoritmusával végezzük. A `quickTa` függvényben, az elemek összehasonlításakor, a `sumE` függvényt alkalmazzuk, amelynek az a szerepe, hogy egy adott telefon esetében meghatározza, hogy hány darab telefont adtak el. A `sumE` függvényben a megfelelő elemek összegét a könyvtárfüggvény `sum`-mal határoztuk meg.

```
quickTa :: [Telef] -> [Telef]
quickTa [] = []
quickTa (k : ve) = quickTa kLs ++ [k] ++ quickTa nLs
    where
    kLs = [x | x <- ve, sumE x < sumE k]
    nLs = [x | x <- ve, sumE x >= sumE k]

sumE :: Telef -> Int
sumE x = sum [thd y | y <- tEladas x]
    where
    thd (y1, y2, y3) = y3
```

Az adatok formázott kiíratását a következő `showTelefSum` függvénnyel végezzük, amelyben a lista elemeinek a feldolgozását a `foldl'` segítségével oldjuk meg, illetve a telefonnevek mellett az összeladási értéket is fetüntetjük:

```
import Data.List (foldl')
showTelefSum :: [Telef] -> String
showTelefSum = foldl' op ""
    where
    op res k = res ++ tNev k ++ "\nOssz ertek: " ++
               showElad (tEladas k) ++ "\n\n"

    showElad :: [(Int, Int, Int)] -> String
    showElad ls = show $ sum $ map thd ls
        where
        thd (y1, y2, y3) = y3

> putStr $ showTelefSum $ quickTa lsT
Lenovo
Ossz ertek: 17

SamsungGalaxyS9
```

```
Ossz ertek: 25
...
```

A következő kódsorokban bemutatjuk a gyorsrendezési algoritmusnak azt a változatát, amikor az összehasonlítási feltételt függvényparaméterként adjuk meg. A módosított `quickSort` függvénynek eggyel több paramétere lesz, ezáltal a `quickSort` alkalmas lesz tetszőleges típusú adatszerkezetek adatainak a rendezésére. Hasonlóan módosítható a többi rendezési algoritmus is.

```
quickSort :: (a -> a -> Bool) -> [a] -> [a]
quickSort fg [] = []
quickSort fg (k : ve) =
   quickSort fg kLs ++ [k] ++ quickSort fg nLs
   where
   kLs = [x | x <- ve, fg x k]
   nLs = [x | x <- ve, not $ fg x k]
```

Ezzel az implementációval a `quickSort` függvény alkalmas lesz különböző adatszerkezetek különböző mezői szerinti rendezésére is. Például egy `Telef` típus esetében, ha név szerinti rendezést akarunk, akkor egy `compareNev` függvényt kell írni, amelyet paraméterként kell alkalmazni a `quickSort`-ben.

```
compareNev :: Telef -> Telef -> Bool
compareNev x y = tNev x <= tNev y

> putStr $ showTelef $ quickSort compareNev lsT
HuaweiP10
2016 1200 20
2017 1000 15
2018 950 18

HuaweiP20
...
```

Ha darabszám szerinti rendezést akarunk, akkor a `compareSum` függvény alkalmazásával tudjuk ezt megvalósítani.

```
compareSum :: Telef -> Telef -> Bool
compareSum x y = sumE x <= sumE y

> putStr$ showTelefSum $ quickSort compareSum lsT
Lenovo
Ossz ertek: 17
```

```
SamsungGalaxyJ5
Ossz ertek: 25
...
```

Ha valós számokból álló lista elemeit szeretnénk rendezni, akkor a következőképpen kell eljárni:

```
compareR :: Ord a => a -> a -> Bool
compareR x y = x < y

> quickSort compareR [8.5, 7.33, 9.25, 8,75, 5.5, 7.33]
[5.5,7.33,7.33,8.0,8.5,9.25,75.0]
```

Az előző két feladat megoldható természetesen a beépített `sortBy` függvénnyel is, próbáljuk ki a következő lekérdezéseket:

```
> import Data.List (sortBy)
> import Data.Ord (comparing)
> putStr $ showTelef $ sortBy (comparing tNev) lsT
> putStr $ showTelefSum $ sortBy (comparing sumE) lsT
> sortBy compare [8.5, 7.33, 9.25, 8,75, 5.5, 7.33]
```

7.9. feladat Egy adott személyről a következő adatok vannak eltárolva: vezetéknév, keresztnév, születési dátum. Írjunk egy Haskell-programot, amely egy `Szemely` elemtípusú lista minden elemére, azaz minden személy esetében meghatározza, hogy a személy a hét milyen napján született.

A feladat megoldásához kétféle adatszerkezetet definiálunk, egyet a dátum, egy másikat pedig egy személy adatainak a kezeléséhez:

```
data Datum = Datum {
   dNap :: Int,
   dHonap:: Int,
   dEv :: Int
   } deriving (Show)

data Szemely = Szemely {
   szVnev :: [Char],
   szKnev :: [Char],
   szSzulD :: Datum
   } deriving (Show)
```

A Haskell-program több függvényből fog állni, ahol a `hetSzulnap` függvényben definiáljuk a hét napjait, majd a `napsz` függvénnyel meghatározzuk az időszámításunk kezdete óta eltelt napok számának a `7`-tel való osztási

maradékát. A hét megfelelő napjának a kiválasztásához ezt az értéket fogjuk használni.

```
hetSzulnap:: Datum -> String
hetSzulnap elem = ls !! k
   where
   ls = ["Vasarnap", "Hetfo", "Kedd", "Szerda",
         "Csutortok", "Pentek", "Szombat"]
   k = napsz (dNap elem) (dHonap elem) (dEv elem)
```

A `napsz` függvényben a napok számának a meghatározásánál figyelembe kell venni, hogy időszámításunk kezdete óta hány szökőévre került sor. Egy év akkor számít szökőévnek, ha osztható az évszám néggyel, viszont nem számítanak szökőévnek a `100`-zal osztható évszámok, csak a `400`-zal oszthatók. E szerint a számítás szerint az algoritmus csak az `1582`-es évtől kezdődően fog helyesen számolni, mert `1582` előtt a Julian-naptár volt érvényben, amely a szökőéveket másképp határozta meg.

```
napsz :: Int -> Int -> Int -> Int
napsz n h e = mod temp 7
   where
    temp = (e-1) * 365
       + div (e-1) 4 - div (e-1) 100 + div (e-1) 400
       + sum (take (h-1) (honapok e))
       + n
```

A `honapok` függvény egy listát határoz meg, amely az aktuális éven belül megadja, hónapokra lebontva, a hónapok napszámát. A `szokoev` függvény meghatározza a februárban esedékes napok számát, amelyet aszerint számol ki, hogy a bemeneti paraméter szökőévnek számít vagy sem.

```
honapok :: Int -> [Int]
honapok y
   = [31, feb, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
   where
   feb
       | szokoev y = 29
       | otherwise = 28

szokoev :: Int -> Bool
szokoev y
   | mod y 100 == 0 = mod y 400 == 0
   | otherwise = mod y 4 == 0
```

A `foSzNap` függvény lesz a főfüggvény, amelynek bemeneti paraméterként az `egySz`, `Szemely` típusú konstans értéket adjuk.

```
foSzNap :: Szemely -> IO()
foSzNap elem = putStrLn $ t1 ++ " szuletesi napja: " ++ t2
   where
   t1 = szVnev elem ++ " " ++ szKnev elem
   t2 = hetSzulnap (szSzulD elem)

egySz = Szemely "Nagy" "Ferenc" (Datum 10 5 1945)

> foSzNap egySz
Nagy Ferenc szuletesi napja: Csutortok
```

7.2. Algebrai adattípusok

Az algebrai adattípusok esetében több értékkonstruktor is lehetséges. A legismertebb algebrai adattípus a `Bool`, aminek típusdeklarációja a standard könyvtárban van. Két értékkonstruktorral rendelkezik, a `True`-val és a `False`-szal. A `Maybe` és `Either` típusok tárgyalásánál említettük, hogy a típusdeklarációban az értékkonstruktorokat `|`-al kell elválasztani.

```
data Bool = False | True
```

Algebrai adattípust mi is definiálhatunk, például létrehozhatunk egy saját `Bool` típust:

```
data MyBool = Igaz | Hamis
   deriving (Show)
```

A következő `parosT` függvényben, egy természetes szám párosságának a vizsgálatához a `MyBool` típust használjuk:

```
parosT :: Integral a => a -> MyBool
parosT n = if even n then Igaz else Hamis

> parosT 10
Igaz
```

A következő példában először egy `Szemely` típust hozunk létre, amelynek két értékkonstruktora lesz, az első, a `Diak` három, míg a második, a `Tanár` két mezővel rendelkezik. Az értékkonstruktorokat írhatjuk egymás mellé, ahogyan fent tettük, de tördelve, külön sorba is meg lehet adni őket, ahogy a következő definícióban láthatjuk:

```
data Szemely = Diak String String Double
             | Tanar String String
   deriving (Show)
```

Egy `Szemely` elemtípusú listának pedig a következőképpen adhatunk kezdőértéket:

```
lsSz :: [Szemely]
lsSz = [Diak "Laci" "ELTE" 4.5, Tanar "Feri" "ELTE",
   Tanar "Mari" "Sapientia", Diak "Lori" "Sapientia" 7.5,
   Diak "Sari" "Sapientia" 8.75, Tanar "Zsuzsi" "ELTE"]
```

A `szamolDiak` függvény meghatározza a `4.5`-nél nagyobb jeggyel rendelkező diákok számát, ahol mintaillesztéssel oldjuk meg, hogy különválasszuk a `Tanar`, illetve a `Diak` értékekkel való műveletvégzést:

```
szamolDiak :: [Szemely] -> Int
szamolDiak ls = length $ filter auxF ls
   where
       auxF (Tanar _ _) = False
       auxF (Diak _ _ jegy) = jegy > 4.5

> szamolDiak lsSz
2
```

A feladat megoldható a `foldl'` függvény használatával is, a kódsor a következő:

```
import Data.List (foldl')
szamolDiak_ :: [Szemely] -> Int
szamolDiak_ = foldl' op 0
   where
   op res k = case k of
       Tanar _ _ -> res
       Diak _ _ jegy -> if jegy > 4.5 then 1 + res
                        else res
```

A következőkben előbb négy típusszinonimát definiálunk, majd egy `BankSzamla` típust, három értékkonstruktorral, amelyekben használjuk a típusszinonimaként megadott típusokat.

```
type KartyaSz = String
type Tulajdonos = String
type Cim = [String]
type FelhasznaloID = Int
```

```
data BankSzamla = BankKartya KartyaSz Tulajdonos Cim
                | Keszpenz
                | Szamla FelhasznaloID
                deriving (Show, Eq)
```

Konstans értékeket a következőképpen is létrehozhatunk:

```
sz1 = BankKartya "12321" "Kiss Antal" ["Mvh", "Romania"]
sz2 = Keszpenz
sz3 = Szamla 12
sz4 = BankKartya "54321" "Nagy Antal" ["Kv", "Romania"]
sz5 = BankKartya "98765" "Beres Antal" ["Mvh", "Romania"]
sz6 = Szamla 13

lsBsz :: [BankSzamla]
lsBsz = [sz1, sz2, sz3, sz4, sz5, sz6]
```

7.10. feladat Írjunk egy Haskell-függvényt, amely kiválogatja egy `BankSzamla` elemtípusú listából azokat a személyeket, akiknek értékkonstruktora `BankKartya`.

A feladatot háromféleképpen is megoldjuk. A `valogatSz` explicit rekurziót használva dolgozza fel a listát, a `valogatSzF` a `foldr` függvényt alkalmazza, a `valogatSzH` pedig halmazműveleteket használ. A `foldr` helyett alkalmazhattuk volna valamelyik `foldl` változatot is.

```
valogatSz :: [BankSzamla] -> [Tulajdonos]
valogatSz [] = []
valogatSz (k : ve) = case k of
   BankKartya _ tu _ -> tu : valogatSz ve
   _ -> valogatSz ve

valogatSzF :: [BankSzamla] -> [Tulajdonos]
valogatSzF ls = foldr op [] ls
   where
   op k res = case k of
       BankKartya _ tu _ -> tu : res
       _ -> res

valogatSzH :: [BankSzamla] -> [Tulajdonos]
valogatSzH ls = [tu | BankKartya _ tu _ <- ls]

> valogatSz lsBsz
["Kiss Antal","Nagy Antal","Beres Antal"]
```

7.11. feladat Írjunk egy Haskell-függvényt, amely külön listákba teszi a `BankKartya`, a `Keszpenz` és a `Szamla` értékkonstruktorokra vonatkozó adatokat. A `Keszpenz` esetében csupán számoljuk meg, hogy hány készpénzkifizetés van.

Figyeljük meg, hogy a kiválogatás során, a `case`-ben, a `BankKartya` értékkonstruktor esetében a `_ _ _` helyett a `{}` szimbólumokat használjuk, mert a `->` jobb oldalán egyetlenegy mezőértékkel sincs műveletvégzés.

```
valogatBK :: [BankSzamla] ->
             ([BankSzamla], [Int], [BankSzamla])
valogatBK ls = auxV ls [] [0] []
   where
   auxV [] bLs kLs sLs = (bLs, kLs, sLs)
   auxV (k : ve) bLs kLs sLs = case k of
       BankKartya {} -> auxV ve (k : bLs) kLs sLs
       Keszpenz -> auxV ve bLs [1 + head kLs] sLs
       Szamla _ -> auxV ve bLs kLs (k : sLs)

> valogatBK lsBsz
([BankKartya "98765" "Beres Antal" ["Mvh",...
```

A következő kódsorban az elegánsabb kiíratáshoz `mapM_`-et használunk, amelyet külön-külön alkalmazunk a tuple elemekre, amelyeket a `valogatBK` függvény visszatérési értékeként kapunk meg.

```
foValogat :: [BankSzamla] -> IO ()
foValogat ls = do
   let (t1, t2, t3) = valogatBK ls
   putStrLn "Bankkartya adatok:"
   mapM_ print t1
   putStrLn "Keszpenzkifizetesek szama: "
   mapM_ print t2
   putStrLn "Szamla adatok:"
   mapM_ print t3

> foValogat lsBsz
Bankkartya adatok:
BankKartya "98765" "Beres Antal" ["Mvh","Romania"]
...
```

7.12. feladat Írjunk egy Haskell-függvényt, amely meghatározza különböző alakzatok (kör, téglalap) területeit, ahol ismerjük az alakzatok térbeli

koordinátapontjait, azaz adva vannak a következő típusszinonimák, illetve típusdeklarációk:

```
type KozepPont = (Double, Double)
type Sugar = Double
type Koord = (Double, Double)

data Alakzat = Kor KozepPont Sugar
             | Teglalap Koord Koord
       deriving (Show)
```

Az `Alakzat` algebrai adattípus két értékkonstruktorral rendelkezik, az egyik a kör adatainak, a másik a téglalap adatainak a kezeléséhez szükséges.

Több konstans értéket is megadunk, amelyeken a számításokat fogjuk végezni:

```
kor1 = Kor (1.5, 2) 10
tegl1 = Teglalap (1.5, 1.5) (3, 4.5)
kor2 = Kor (3.5, 3.5) 23
tegl2 = Teglalap (2.5, 3.5) (10.6, 4.5)
tegl3 = Teglalap (1.5, 1.7) (0.6, 2.5)

lsA = [kor1, tegl1, kor2, tegl2, tegl3]
```

A következő függvény egy alakzat területét határozza meg, a `teruletLs` pedig listában levő térbeli alakzatok területeit számolja ki.

```
terulet :: Alakzat -> Double
terulet k = case k of
       Kor _ r -> r * r * pi
       Teglalap (x1, y1) (x2, y2)
           -> abs (x2 - x1) * abs (y2 - y1)

> terulet kor2
1661.9025137490005

> terulet tegl1
4.5

teruletLs :: [Alakzat] -> [Double]
teruletLs = map terulet

> teruletLs lsA
[314.1592653589793,4.5,1661.9025137490005,8.1]
```

Az eredmény elegánsabb megjelenítéséhez a következőkben másképp fogunk eljárni. Az alakzatok típusa szerint külön listákat fogunk létrehozni. A `nevLs` függvény segítségével egy olyan listát hozunk létre, amelybe az alakzatok típusa szerint az alakzatok neveit tesszük. A `foAlakzat` függvényben a kiíratást végezzük, amelyben a `zip` segítségével az alakzatok neveit és az alakzatok területeit tartalmazó listákat összekapcsoljuk. A kiíratás módját pedig a `mapM_`-nek megadott függvényparaméterben fogjuk meghatározni:

```
nevLs :: [Alakzat] -> [String]
nevLs [] = []
nevLs (k : ve) = case k of
   Kor {} -> "Kor" : nevLs ve
   Teglalap {} -> "Teglalap" : nevLs ve

foAlakzat :: [Alakzat] -> IO()
foAlakzat ls = mapM_ auxF rLs
   where
   auxF (k1, k2) = putStrLn $ k1 ++ ": " ++ show k2
   nLs = nevLs ls
   tLs = teruletLs ls
   rLs = zip nLs tLs

> foAlakzat lsA
Kor: 314.1592653589793
Teglalap: 4.5
...
```

7.3. Paraméterezett típusok

Haskellben, ahogyan a függvények típusdeklarációi, úgy valamely adat vagy adatszerkezet típusának definiálásakor is használhatunk típusváltozót, azaz valamely típuskonstruktor is lehet *paraméterezett*. Korábban, a hibakezeléseknél jeleztük, hogy a `Maybe` típus paraméterezett, azaz alkalmazható tetszőleges típusú adat vagy adatszerkezet esetében.

Paraméterezett típust mi is definiálhatunk, például megadhatunk egy saját `Maybe` típust:

```
data MyMaybe a = Semmi | Ertek a
   deriving (Show)
```

A korábban megírt `init` függvényt is átírhatjuk, használva a fent definiált `MyMaybe` típust:

```
initMyMaybe :: [a] -> MyMaybe [a]
initMyMaybe = foldr op Semmi
    where
    op k Semmi = Ertek []
    op k (Ertek ve) = Ertek (k : ve)

> initMyMaybe ""
Semmi

> initMyMaybe "Retyezat Nemzeti Park"
Ertek "Retyezat Nemzeti Par"
```

7.4. Rekurzív típusok

A Haskell megengedi a rekurzív adatszerkezetek definiálását, például a lista adatszerkezetet is meg lehet adni rekurzív formában:

```
data MyList a = Ures | Fuz a (MyList a)
    deriving (Show)
```

A fenti definíció azt jelenti, hogy egy `MyList` típusú adat kétfajta értékkonstruktorral rendelkezik. Az első azt az esetet kezeli, amikor a lista üres (`Ures`), a második pedig azt az esetet, amikor egy `MyList` típusú adatot úgy hozunk létre, hogy kombinálunk egy tetszőleges típusú `a` elemet és egy tetszőleges típusú `MyList` elemet. Ez utóbbi esetben rekurziót használtunk a `MyList` típusának a leírásánál.

Értékadó műveleteket és függvényeket is írhatunk a `MyList` típusra:

```
myLength :: Num t => MyList t1 -> t
myLength Ures = 0
myLength (Fuz k ve) = 1 + myLength ve

> ls1 = Fuz 4 Ures
> ls2 = Fuz 3 (Fuz 4 Ures)
> ls3 = Fuz 1 $ Fuz 2 ls2
> ls3
Fuz 1 (Fuz 2 (Fuz 3 (Fuz 4 Ures)))
```

```
> myLength ls3
4
```

Egy másik adatszerkezet, amelyet meg lehet adni rekurzív típusdeklarációval, az a **bináris fa**. Bináris fákat számos helyen használunk, elsősorban azért, mert ha egy ilyen szerkezet segítségével kezeljük az adatokat, akkor a hozzáadás, keresés, törlés stb. hatékonyan megvalósítható. A lista adatszerkezettel ellentétben, amely egy lineáris struktúra, a bináris fa egy hierarchikus szerkezet, éppen ezért az adatokat kezelő algoritmusok időigényét lehet javítani. Felépítésük során számos aspektust szoktak figyelembe venni, ezek közül a jegyzetben a legegyszerűbbekről lesz szó. A bináris fák esetében fontos még megjegyezni, hogy egy adatból, azaz egy csomópontból maximum két további adatot (csomópontot) tudunk elérni.

Egy bináris fa szerkezetét azért is nagyon egyszerű rekurzív típusdeklarációval megadni, mert a bináris fa fogalma is legtöbbször rekurzívan van értelmezve.

```
data BinFa a = UresFa | Csomop a (BinFa a) (BinFa a)
   deriving (Show, Read, Eq)
```

A típusdeklarációból leolvasható, hogy a `BinFa` egy olyan szerkezet, amely két értékkonstruktorral rendelkezik, az egyik esetben a csomópontot üres (`UresFa`) értékként határozzuk meg, a másik értékkonstruktor pedig egy olyan csomópontot definiál, amelynek két `BinFa` típusú mezője van, ahol a csomópontot *szülőnek*, a két mezőt, amelyek tulajdonképpen két további csomópontot jelölnek, pedig *gyerekeknek* is szokták hívni. Vegyük észre azt is, hogy a megadott `BinFa` egy paraméterezett típus lesz, ami azt jelenti, hogy a csomópontokban tetszőleges típusú adat tárolható.

A következő `egyelemuFa` függvény egy egyelemű fa létrehozására lesz alkalmas, amelyet a `beszurFa` függvényben fogunk használni akkor, amikor egy adott ágon már nem tudunk lennebb menni, azaz `UresFa` csomóponthoz jutottunk.

```
egyelemuFa :: Ord a => a -> BinFa a
egyelemuFa x = Csomop x UresFa UresFa

beszurFa :: Ord a => a -> BinFa a -> BinFa a
beszurFa x UresFa = egyelemuFa x
beszurFa x (Csomop a bal jobb)
   | x <= a = Csomop a (beszurFa x bal) jobb
   | x > a = Csomop a bal (beszurFa x jobb)
```

A fenti két függvény típusdeklarációjából látható, hogy megköveteljük, hogy az adatok, amelyeket a csomópontokba teszünk, azok az `Ord` típusosztályhoz tartozzanak. Ez azért szükséges, mert a beszúrás során a bal oldali ághoz csatoljuk azt a csomópontot, ami a szülő csomópontban levő értékhez képest kisebb értéket tárol, és a jobb oldalihoz a nagyobbat. Ezzel a beszúrási algoritmussal tulajdonképpen egy **bináris keresőfát** építünk, amelyből az adatok rendezett sorrendjét a bináris fa csomópontjainak egy bejárásával hatékonyan meg tudjuk határozni.

A következő `letrehoz` függvény segítségével egy `4` csomópontból álló bináris keresőfát építünk, ahol a csomópontokba `String` típusú adatokat teszünk, ahol a rendezettségi sorrendet a lexikografikus sorrend adja.

```
letrehoz = do
   let f1 = beszurFa "mari" UresFa
   let f2 = beszurFa "zsuzsa" f1
   let f3 = beszurFa "mari" f2
   let f4 = beszurFa "feri" f3
   print f4

> letrehoz
Csomop "mari" (Csomop "mari" (Csomop "feri" UresFa UresFa)
   UresFa) (Csomop "zsuzsa" UresFa UresFa)
```

A csomópontok száma azért lesz `4`, mert a `"mari"` adat kétszer is szerepelni fog a keresőfában.

Ha azt szeretnénk, hogy a csomópontok egyedi értékeket tároljanak, akkor a `beszurFa` a következőképpen módosul:

```
beszurFa_ :: Ord a => a -> BinFa a -> BinFa a
beszurFa_ x UresFa = egyelemuFa x
beszurFa_ x (Csomop a bal jobb)
   | x == a = Csomop x bal jobb
   | x < a = Csomop a (beszurFa_ x bal) jobb
   | x > a = Csomop a bal (beszurFa_ x jobb)
```

Az adatok fenti beszúrási módja mellett lehetséges, hogy listában megadott elemek alapján hozzuk létre a bináris keresőfát. A `letrehozFa` függvény a `beszurFa` függvény alkalmazásával ezt végzi. A függvény bemeneti paramétere egy olyan lista kell legyen, amelynek elemei az `Ord` típusosztályhoz tartoznak, kimenete pedig egy `BinFa a` típusú adat lesz.

```
letrehozFa :: (Ord a) => [a] -> BinFa a
letrehozFa [] = UresFa
letrehozFa (k : ve) = beszurFa k $ letrehozFa ve
```

```
> letrehozFa ["mari", "zsuzsa", "feri", "mari"]
Csomop "mari" (Csomop "feri" UresFa (Csomop "mari"
   UresFa UresFa)) (Csomop "zsuzsa" UresFa UresFa)
```

Ha lépésenkét szeretnénk látni, ahogy a listaelemek alapján épül a fa, akkor módosíthatjuk a fenti függvényt:

```
letrehozFaIr :: (Ord a, Show a) => [a] -> IO (BinFa a)
letrehozFaIr [] = return UresFa
letrehozFaIr (k : ve) = do
       rFa <- letrehozFaIr ve
       putStr $ show rFa ++ "\n"
       return $ beszurFa k rFa

> letrehozFaIr ["mari", "zsuzsa", "feri", "mari"]
UresFa
Csomop "mari" UresFa UresFa
...
```

Listaelemek alapján egy bináris fa felépítésének legegyszerűbb kódsora a `foldr` függvény használatával adható meg:

```
letrehozFaFold :: Ord a => [a] -> BinFa a
letrehozFaFold = foldr beszurFa UresFa

> letrehozFaFold ["mari", "zsuzsa", "feri", "mari"]
Csomop "mari" (Csomop "feri" UresFa (Csomop "mari"
   UresFa UresFa)) (Csomop "zsuzsa" UresFa UresFa)
```

Ha sikerült felépíteni a bináris fát, akkor három különböző bejárási sorrendet adhatunk meg, a preorder, az inorder és a posztorder bejárásokat:

- A preoder esetében először a szülő, utána a bal oldali, majd a jobb oldali csomópontot kell érinteni.
- Az inoder esetében először a bal oldali, utána a szülő, majd a jobb oldali csomópontot kell érinteni.
- A posztorder esetében először a bal oldali, utána a jobb oldali, majd a szülő csomópontot kell érinteni.

Egy bináris keresőfa esetében a csomópontok inorder bejárási sorrendje rendezett sorrendben fogja megadni az elemeket:

```
inorderFa :: (Ord a) => BinFa a -> [a]
inorderFa UresFa = []
inorderFa (Csomop a bal jobb) =
       inorderFa bal ++ [a] ++ inorderFa jobb
```

```
> :set +m
> f = letrehozFaFold ["mari", "zsuzsa", "feri", "mari",
|     "kati", "laci"]
> inorderFa f
["feri","kati","laci","mari","mari","zsuzsa"]
```

A preorder, illetve posztorder bejárásokat a `preorderFa`, illetve `postorderFa` függvények végzik. Habár ismert a függvények hatékonyabb változata, mégis a következő implementációkat adjuk meg, mert ezek felépítése a korábban megadott értelmezéseket követik.

```
preorderFa :: (Ord a) => BinFa a -> [a]
preorderFa UresFa = []
preorderFa (Csomop a bal jobb) = [a] ++ preorderFa bal
       ++ preorderFa jobb

postorderFa :: (Ord a) => BinFa a -> [a]
postorderFa UresFa = []
postorderFa (Csomop a bal jobb) = postorderFa bal
       ++ postorderFa jobb ++ [a]
```

Egy bináris keresőfa legjobboldalibb eleme a legnagyobb eleme lesz a fának, meghatározását a következő `maximumFa` függvény adja:

```
maximumFa :: BinFa a -> Maybe a
maximumFa UresFa = Nothing
maximumFa (Csomop a bal UresFa) = Just a
maximumFa (Csomop a bal jobb) = maximumFa jobb

> maximumFa f
Just "zsuzsa"
```

A bináris fa mélysége elsősorban a keresési, illetve bejárási algoritmusok hatékonyságát befolyásolja, meghatározását a következő `melysegFa` függvényben fogjuk végezni, ahol két elem maximumának a meghatározásához a `max` könyvtárfüggvényt használjuk:

```
melysegFa :: BinFa a -> Int
melysegFa UresFa = 0
melysegFa (Csomop a bal jobb) = 1 +
       max (melysegFa bal) (melysegFa jobb)

> melysegFa f
3
```

7.5. Kitűzött feladatok

7.1. feladat Egy szövegállományban egy adott városról a következő adatok vannak eltárolva: városnév, népességszám, területméret, azaz adott a következő adatszerkezet:

```
data Varos = Varos {
   vNev :: [Char],
   vNepSzam :: Int,
   vTerMeret :: Double
   } deriving (Show)
```

Írjunk egy Haskell-programot, amely az állományban levő adatok alapján létrehoz egy `Varos` elemtípusú listát, majd

- meghatározza, hogy hány olyan város van, amelyiknek a népsűrűsége egy megadott `[a, b]` intervallumba esik, ahol az `a` és `b` értékeket a billentyűzetről olvassuk be,
- meghatározza a városok népsűrűség szerinti rendezett sorrendjét, az eredményt *elegáns* formában kiírva a képernyőre `(népsűrűség = népesség-szám / terület-méret)`.

7.2. feladat Egy `Fesztivalok` elemtípusú listában a következő adatok vannak eltárolva: fesztiválnév, fesztiválkód, jegyár és az együttesnevek, azaz adott a következő adatszerkezet:

```
data Fesztivalok = Fesztivalok{
   fFesztival :: String,
   fKod :: Int,
   fAr :: Int,
   fEgyuttes :: [String]
   } deriving (Show)
```

Írjunk egy Haskell-programot, amely egy `Fesztivalok` elemtípusú lista esetében:

- meghatározza, hogy hány olyan fesztivál szerepel a listában, amely egy adott értéknél olcsóbban árusítja a jegyeket,
- meghatározza, hogy egy adott fesztiválon milyen együttesek lépnek fel,
- meghatározza azt a fesztivált, ahol a legtöbb együttes lép fel,
- meghatározza minden egyes fesztivál esetében a részt vevő együttesek számát,
- kiírja formázva, a jegyárak alapján rendezve, a fesztiválok adatait,

– létrehoz egy bináris keresőfát a fesztiválnév alapján, majd inorder bejárást alkalmazva meghatározza a fesztiválok ábécésorrendjét.

7.3. feladat Egy szövegállományban egy adott sportolimpiáról a következő adatok vannak eltárolva: ország és az eredmények sportáganként. Az eredmények egy (sportág, érmék száma) értékpárokból álló listát jelent, azaz adott a következő adatszerkezet:

```
data Olimpia = Olimpia {
   oOrszag :: String,
   oSportagak :: [(String, Int)]
   } deriving (Show)
```

Írjunk egy Haskell-programot, amely az állományban levő adatok alapján létrehoz egy `Olimpia` elemtípusú listát, és

– meghatározza, hogy egy adott ország összesen hány érmét szerzett,
– meghatározza, hogy melyik ország szerezte a legtöbb érmét a sportolimpián,
– meghatározza, hogy milyen sportágak esetében osztottak díjakat,
– meghatározza, hogy egy adott sportágon belül hány díjat osztottak,
– egy adott ország esetében kiírja, formázva, a sportáganként szerzett érmék száma szerinti rendezett sorrendet,
– létrehoz egy bináris keresőfát, az országnevek alapján, majd inorder bejárást alkalmazva kiírja formázva az ábécésorrendet.

7.4. feladat Egy szövegállományban egy adott személyről a következő adatok vannak eltárolva: vezetéknév, keresztnév, születési dátum, azaz adottak a következő adatszerkezetek:

```
data Datum = Datum {
   dNap :: Int,
   dHonap:: Int,
   dEv :: Int
   } deriving (Show)

data Szemely = Szemely {
   szVnev :: [Char],
   szKnev :: [Char],
   szDatum :: Datum
   } deriving (Show)
```

Írjunk egy Haskell-programot, amely az állományban levő adatok alapján létrehoz egy `Szemely` elemtípusú listát, majd meghatározza mindegyik személyről, hogy a hét milyen napján született.

7.5. feladat Írjunk Haskell-függvényt, amely létrehoz egy valós számokat tároló bináris keresőfát, és meghatározza inorder bejárással a számok rendezett sorrendjét, illetve a csomópontokban található számok összegét.

8. fejezet

Algoritmusok és megoldott feladatok

8.1. Kombinatorikai feladatok

A funkcionális programozási paradigma könnyedén alkalmazható kombinatorikai feladatok megoldásához. Az algoritmusok kompaktak, olvashatók és viszonylag hatékonyak lesznek. Ebben a fejezetben a legfontosabb kombinatorikai algoritmusok Haskell-kódját mutatjuk be.

8.1. feladat Írjunk egy Haskell-függvényt, amely meghatározza az összes olyan `m` hosszúságú listát, amely egy bemeneti lista elemeiből képezhető.

```
lGen_ :: (Eq a) => [a] -> Int -> [[a]]
lGen_ ls 0 = [[]]
lGen_ ls m = [k : ve | k <- ls, ve <- lGen_ ls (m-1)]
```

Az `lGen_` függvény halmazműveleteket alkalmaz, és azon az elgondoláson alapszik, hogy létrehozzuk azokat a listákat, amelyek első elemei rendre a bemeneti lista elemei lesznek, a többi elemet pedig rekurzívan generáljuk. A rekurzív hívásban a függvény első paramétere az eredeti lista lesz, a második paramétert pedig minden egyes alkalommal csökkentjük eggyel. Ha az értéke `0` lesz, akkor ezt a triviális esetben fogjuk kezelni.

```
> lGen_ [0, 1] 2
[[0,0],[0,1],[1,0],[1,1]]

> lGen_ [0, 1] 4
```

```
[[0,0,0,0],[0,0,0,1],[0,0,1,0],[0,0,1,1],[0,1,0,0],
 [0,1,0,1],[0,1,1,0],[0,1,1,1],[1,0,0,0],[1,0,0,1],
 [1,0,1,0],[1,0,1,1],[1,1,0,0],[1,1,0,1],[1,1,1,0],
 [1,1,1,1]]

> lGen_ "ab" 3
["aaa","aab","aba","abb","baa","bab","bba","bbb"]

> lGen_ "ab" 2
["aa","ab","ba","bb"]
```

Vegyük észre, hogy ha a bemeneti lista a `0` és `1` értékekből áll, akkor a kimenet az `m` elemű bitkonfigurációk listája lesz, ha azonban a bemeneti listába *ábécésorrendben* adjuk meg a betűket, akkor az eredmény listába lexikografikus sorrendben kigenerált szavak kerülnek.

A következő kódsorban módosítjuk a listaelemek generálási módjának a sorrendjét, azaz először generáljuk a `ve` listaelemeket, és csak ezután adjuk meg a `k` értékének a generálási módját.

```
lGen :: (Eq a) => [a] -> Int -> [[a]]
lGen ls 0 = [[]]
lGen ls m = [k : ve | ve <- lGen ls (m-1), k <- ls]
```

Ez a módosítás a listaelemek más sorrendjét fogja eredményezni és hatékonyság szempontjából is nagyon fontos lesz, ahol figyeljük meg a `foGen` és a `foGen_` időigényei közötti lényeges különbséget.

```
> lGen "ab" 3
["aaa","baa","aba","bba","aab","bab","abb","bbb"]

foGen :: (Show a, Eq a) => [a] -> Int -> IO ()
foGen ls m = do
   let rLs = lGen ls m
   print $ last rLs

> :set +s
> foGen [0,1] 20
[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
(0.67 secs, 352,405,392 bytes)

foGen_ :: (Show a, Eq a) => [a] -> Int -> IO ()
foGen_ ls m = do
   let rLs = lGen_ ls m
   print $ last rLs
```

```
> foGen_ [0,1] 20
[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
(3.93 secs, 3,120,641,320 bytes)
```

A `foGen`, illetve `foGen_` függvényekben csak a kigenerált lista utolsó elemét írattuk ki, így a függvények időigényének értékét nem befolyásolta a listaelemek kiíratásának időigénye.

8.2. feladat Írjunk egy Haskell-függvényt, amely meghatározza `n` elem `m`-ed rendű kombinációit.

```
komb :: (Ord a) => [a] -> Int -> [[a]]
komb ls 0 = [[]]
komb ls m = [k : ve |
       ve <- komb ls (m-1),
       k <- ls, feltKomb k ve]

feltKomb :: (Ord a) => a -> [a] -> Bool
feltKomb x [] = True
feltKomb x (k : ve)
   | k <= x = False
   | otherwise = feltKomb x ve

> komb [4, 1, 7, 9] 3
[[1,4,7],[1,4,9],[4,7,9],[1,7,9]]

> komb "wxyz" 2
["wx","wy","xy","wz","xz","yz"]
```

A `komb` függvény algoritmusa a korábbi `lGen` függvényen alapszik. A `komb` függvény paraméterként egy listát kap és az `m` értéket, ahol a lista elemszáma adja a feladat megfogalmazásában szereplő `n` értéket. A `komb` függvény tehát az `n` elemszámú lista elemeiből határozza meg az `m`-ed rendű kombinációkat. Most is hasonló elgondolás alapján generáljuk a listákat, a különbség az lesz, hogy mivel a kigenerált listák nem mindegyike felel meg a feladat kritériumának, ezért egy tesztelő függvényt alkalmazunk. Ez a `feltKomb` nevű függvény lesz, ahol a függvény kimenete akkor lesz `True`, ha a bemeneti lista elemei szigorúan növekvő sorrendben lesznek, pontosabban nem adhatunk a listához olyan elemet, amely az előző elemek valamelyikénél kisebb vagy vele egyenlő. Így oldjuk meg, hogy ne generáljunk olyan

listákat, amelyekben az elemek ugyanazok, de a sorrend különböző. Például a fenti bemenet esetében az eredmény lista nem tartalmazhatja az `[1, 4, 7]`, `[1, 7, 4]`, `[7, 4, 1]` stb. mindegyikét. A kódsor szerint csak az `[1, 4, 7]`-t fogja tartalmazni. A `feltKomb`-ban, ha módosítjuk a feltételt `k >= x` = `False`-ra, akkor a `[7, 4, 1]` fog szerepelni az eredménylistában.

A következő kódsor a `feltKomb` függvény egy kompaktabb változata:

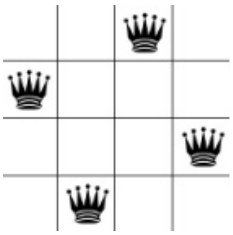
```
feltKomb_ :: (Ord a) => a -> [a] -> Bool
feltKomb_ x = all (aux x)
   where
   aux x k = k > x
```

8.3. feladat Írjunk egy Haskell-függvényt, amely elhelyez egy sakktáblán `8` királynőt úgy, hogy azok ne üssék egymást. Általánosan is oldjuk meg a feladatot, azaz helyezzünk el egy `nxn`-es sakktáblán `n` királynőt úgy, hogy azok ne üssék egymást.

A gyakorlatban egy `nxn`-es sakktáblán, ha el akarunk helyezni `n` királynőt úgy, hogy azok ne üssék egymást, ez azt fogja jelenteni, hogy nem tehetjük őket ugyanabba a sorba, oszlopba, illetve átlóra.

A megoldás listák kigenerálását fogja jelenti, ahol a listaelemek egész számok lesznek, ahol az első listaelem az első sorban levő királynő oszlop-pozícióját fogja jelenteni, a második listaelem a második sorban levő királynő oszlop-pozícióját jelenti, és így tovább.

Egy `4x4`-es sakktáblán a `[3, 1, 4, 2]` helyes megoldás azt fogja jelenteni, hogy a királynőt az első sorban a harmadik, a második sorban az első, a harmadik sorban a negyedik és a negyedik sorban a második oszlopba tettük:



Azzal, hogy megoldásnak a listaelemek ilyen formáját választjuk, máris megoldottuk a sorok szerinti ütközés problémáját. A kigenerált listaelemek esetében tehát most már csak az oszlopok és átlók menti ütközés problémáját kell megoldani.

Az elgondolás az előző feladat algoritmusának egy változata lesz, ahol más lesz a feltételeket vizsgáló függvény, ilyen módon a feladat főfüggvénye a `kiralyno` lesz, a feltételek teljesülését pedig a `feltKir` biztosítja.

```
kiralyno :: Int -> [[Int]]
kiralyno n = auxK n n
   where
   auxK :: Int -> Int -> [[Int]]
   auxK n 0 = [[]]
   auxK n m = [k : ve |
           ve <- auxK n (m-1),
           k <- [1..n], feltKir k ve ]

feltKir :: Int -> [Int] -> Bool
feltKir x ls = auxFeltK 1 x ls
   where
   auxFeltK :: Int -> Int -> [Int] -> Bool
   auxFeltK i x [] = True
   auxFeltK i x (k : ve)
       | k == x = False
       | abs (x - k) == i = False
       | otherwise = auxFeltK (i + 1) x ve

> kiralyno 5
[[4,2,5,3,1],[3,5,2,4,1],[5,3,1,4,2],[4,1,3,5,2],
 [5,2,4,1,3],[1,4,2,5,3],[2,5,3,1,4],[1,3,5,2,4],
 [3,1,4,2,5],[2,4,1,3,5]]
```

A `n`-királynő feladat esetében a listaelemek nem kell növekvő sorrendben legyenek, elég, ha különböznek, amivel megoldjuk a királynők oszlop szerinti ütközésének a problémáját. Vegyük észre azt is, hogy az eredménylisták elemeit akkor határozzuk meg, amikor *jövünk vissza* a rekurzióból. Például az `5x5`-ös sakktábla esetében a `[4,2,5,3,1]` lista elsőként meghatározott eleme az `1` lesz, majd eléje kerül a `3`, majd az elé az `5`-ös, és így tovább.

Az átlók menti ütközés kizárásához egy plusz feltételt kellett bevezetni. A kódsorban az `abs (x - k) == i = False` sor fogja azt jelenteni, hogy a `x` elem a már kigenerált listabeli elemek közül a `k`-val, amely `i` pozícióval helyezkedik el az `x`-től, az átlók mentén, ütközni fog. Figyeljük meg, hogy `i` kezdeti értéke `1`. Például a `6x6`-os sakktábla esetében a `[2,6,3]` lista elejére nem lehet `1`-est tenni, mert az átlós ütközést eredményezne a `2`-vel (`abs (2 - 1) == 1`), de `4`-est sem lehet tenni, mert az a `6`-tal okozna átlós ütközést (`abs(6 - 4) == 2`).

A `feltKir` függvénynek is megadjuk a tömörebb változatát:

```
feltKir_ :: Int -> [Int] -> Bool
feltKir_ x ls = auxFeltK 1 x ls
   where
   auxFeltK :: Int -> Int -> [Int] -> Bool
   auxFeltK i x ls = all (aux x) $ zip [1..] ls
       where
       aux x (i, k) = (k /= x) && (abs(x - k) /= i)
```

A feladat megoldásait elegánsabban is meg tudjuk jeleníteni a képernyőn, a `foKiralyno` függvény meghívásával:

```
kiirSor :: Int -> Int -> IO()
kiirSor n k = do
   mapM_ (auxF k) [1..n]
   putStrLn ""
       where
       auxF k x =
           if k == x then putStr "Q " else putStr ". "

kiirTabla :: Int -> [Int] -> IO()
kiirTabla n ls = do
   mapM_ (kiirSor n) ls
   putStrLn ""

foKiralyno :: Int -> IO()
foKiralyno n = mapM_ (kiirTabla n) $ kiralyno n
```

```
> foKiralyno 5
. . . Q .
. Q . . .
. . . . Q
. . Q . .
Q . . . .

. . Q . .
. . . . Q
. Q . . .
. . . Q .
Q . . . .

...
```

Észrevehető egy algoritmikai hasonlóság az `lGen`, a `komb`, illetve a `kiralyno` függvények között. Megállapítható, hogy a különbség közöttük csak az, hogy különböző feltételeket definiáló függvényt alkalmaztunk a megfelelő listák kigenerálására. Éppen ezért megadható a fenti feladatoknak egy általánosabb változata.

A következő `rekurzio` függvény, a már ismertetett elgondolás alapján, kigenerálja a megfelelő listákat, és mindig az `[1..n]` bemenetből válogat. A generálási szabálynak megfelelő feltételfüggvényt paraméterként adjuk meg, így a `rekurzio` függvény helyettesíteni tudja a korábban megadott három implementációt, illetve alkalmas lesz permutációk, variációk generálására is. Természetesen a megfelelő feltételeket definiáló függvényt minden esetben meg kell adni.

```
rekurzio :: Int -> Int -> (Int -> [Int] -> Bool)
            -> [[Int]]
rekurzio n 0 fg = [[]]
rekurzio n m fg = [k : ve | ve <- rekurzio n (m-1) fg,
       k <- [1..n], fg k ve]

lGenR :: Int -> Int -> [[Int]]
lGenR n m = rekurzio n m (\ k ve -> True)

kombR :: Int -> Int -> [[Int]]
kombR n m = rekurzio n m feltKomb

kiralynoR :: Int -> [[Int]]
kiralynoR n = rekurzio n n feltKir

permutacioR :: Int -> [[Int]]
permutacioR n = rekurzio n n (\ k ve -> notElem k ve)

variacioR  :: Int -> Int -> [[Int]]
variacioR n m = rekurzio n m notElem
```

A fenti kódsorokban `5` olyan függvényt adtunk meg, amelyek mindegyike a `rekurzio` függvényt alkalmazza, a megfelelő feltétel-függvénnyel. Ezek közül például ha `3` elem permutációját szeretnénk meghatározni, akkor a következő lekérdezéssel tudjuk ezt kigenerálni:

```
> permutacioR 3
[[3,2,1],[2,3,1],[3,1,2],[1,3,2],[2,1,3],[1,2,3]]
```

Vegyük észre, hogy a `permutacioR` és `variacioR` függvényekben a feltételfüggvényként megadott kifejezések tulajdonképpen ekvivalensek.

8.4. feladat Írjunk egy Haskell-függvényt, amely meghatározza, hogy hányféleképpen állítható elő egy adott `sumV` összeg az `ls` listában megadott számokból, ha mindegyik számot csak egyszer használhatjuk fel.

Az algoritmus a korábban megadott kombinációkat kigeneráló függvényt fogja felhasználni, mert a feladat tulajdonképpen átfogalmazható a következőképpen: határozzuk meg az adott `ls` listából képezhető `1, 2, ..., m` elemű kombinációkat úgy, hogy az elemek összege egyenlő legyen egy adott `sumV` értékkel. A függvény egy adott listában levő elemek összegének a meghatározását a `sum` beépített függvénnyel fogja meghatározni.

```
felSum1 :: Int -> [Int] -> [[Int]]
felSum1 sumV ls = auxSum 1 (length ls) sumV ls
   where
   auxSum :: Int -> Int -> Int -> [Int] -> [[Int]]
   auxSum i len s ls
       | i == len = []
       | otherwise =
           [kLs | kLs <- komb ls i, sum kLs == s]
           ++ auxSum (i+1) len s ls

> felSum1 13 [4, 2, 7, 9]
[[4,9],[2,4,7]]
```

Az `auxSum` segédfüggvény az `1, 2, ..., length ls` rendű kombinációk közül válogat, és a `++` operátorral egy kimeneti listába fűzi a feltételnek eleget tevő listákat.

Az előző feladat megoldható hatékonyabb algoritmussal is, mégpedig úgy, hogy meghatározzuk az adott `ls` listából előállítható **részhalmazokat**, kivéve az üres halmazt, és ezekből válogatunk, a feltételnek megfelelően, azaz kiválasztjuk azokat a részhalmazokat, amelyek elemeinek összege `sumV`.

```
felSum2 :: Int -> [Int] -> [[Int]]
felSum2 sumV ls = [ kLs |
       kLs <- reszH ls, sum kLs == sumV]

reszH :: [Int] -> [[Int]]
reszH [] = []
reszH (k : ve) = auxH k nVe ++ nVe
```

```
    where
    nVe = reszH ve

auxH :: Int -> [[Int]] -> [[Int]]
auxH x [] = [[x]]
auxH x (kL : veL) = (x : kL) : auxH x veL

> reszH [1,2,3]
[[1,2,3],[1,2],[1,3],[1],[2,3],[2],[3]]

> auxH 3 [[2, 1], [2], [1], []]
[[3,2,1],[3,2],[3,1],[3]]

> felSum2 30 [1, 2, 3, 5, 7, 8, 9, 10, 15]
[[1,2,3,5,9,10],[1,2,3,7,8,9],[1,2,3,9,15],[1,2,5,7,15],
[1,2,8,9,10],[1,3,7,9,10],[1,5,7,8,9],[1,5,9,15],
[2,3,7,8,10],[2,3,10,15],[2,5,8,15],[3,5,7,15],
[3,8,9,10],[5,7,8,10],[5,10,15],[7,8,15]]
```

Megadott listaelemek alapján a részhalmazok előállítását a `reszH` függvény végzi, ennek működését pedig az `1,2,3,4` elemekből képezhető részhalmazok előállításával szemléltetjük:

- feltételezzük, hogy előállítottuk egy listában az összes részhalmazt, amelyeket az `1,2` elemekből képezhetünk, legyen ez:

  `[[2,1],[2],[1],[]]`

- ha most az `1,2,3` elemekből képezhető részhalmazokat akarjuk előállítani, akkor a következőket kell elvégezzük:
  - szúrjuk be a `3`-as elemet minden már előállított részhalmazba (ezt végzi az `auxH` függvény):

    `[[3,2,1],[3,2],[3,1],[3]]`

  - fűzzük egymás után a két listát:

    `[[3,2,1],[3,2],[3,1],[3]``,[2,1],[2],[1],[]]`

- a `4`-es beszúrása a leírtakhoz hasonló módon történik
- a kiindulási pont az üres lista lesz, ezt fogjuk rendre bővíteni az `1, 2, 3, 4, ...` elemekkel.

A fenti gondolatmenet alapján az `1,2,3,4` elemekből képezhető részhalmazok a következőképpen lesznek létrehozva:

`[]` $\xrightarrow{1}$ `[1]`
`[1],[]` $\xrightarrow{2}$ `[2,1],[2]`

```
[2,1],[2],[1],[] -3-> [3,2,1],[3,2],[3,1],[3]
[3,2,1],[3,2],[3,1],[3],[2,1],[2],[1],[] -4->
    [4,3,2,1],[4,3,2],[4,3,1],[4,3],[4,2,1],[4,2],[4,1],[4]
```

Az eredmény:

```
[4,3,2,1],[4,3,2],[4,3,1],[4,3],[4,2,1],[4,2],[4,1],[4],
[3,2,1],[3,2],[3,1],[3],[2,1],[2],[1],[]
```

8.5. feladat Az előző feladatot oldjuk meg több bemeneti értékre, ahol a bemeneti értékek a `szamok.txt` állományban vannak, sorokba tördelve. Az eredményt az `eredmeny.txt` állományba írjuk.

A `szamok.txt` állomány minden egyes sorában az első szám a `sumV` összeget jelöli, azaz ezt az összeget kell előállítani, a többi szám pedig azokat az értékeket jelöli, amelyekből elő kell állítani a `sumV` összeget, azaz ezek lesznek az `ls` lista elemei.

Ha a `szamok.txt` állomány tartalma a következő:

```
30 1 2 3 5 7 9 10 15
230 2 90 10 120 15 40 20 50 70
13 4 2 7 9
```

akkor az `eredmeny.txt` állomány tartalma a következő lesz, ahol az eredmények elé kiírtuk a bemeneteket is:

```
30 [1,2,3,5,7,9,10,15]
[[1,2,3,5,9,10],[1,2,3,9,15],[1,2,5,7,15],[1,3,7,9,10],
[1,5,9,15],[2,3,10,15],[3,5,7,15],[5,10,15]]

230 [2,90,10,120,15,40,20,50,70]
[[90,10,40,20,70],[90,120,20],[90,20,50,70],[120,40,20,50],
[120,40,70]]

13 [4,2,7,9]
[[4,2,7],[4,9]]
```

A következő kódsor második sorában egy új import jelenik meg. A `Data.List.Split` könyvtárat importáltuk, hogy használni tudjuk a `wordsBy` függvényt. A könyvtár használatához szükség van a `split` installálására, amely a `cabal` segítségével könnyedén megoldható. Windows alatt el kell indítani adminisztrátor módban egy `PowerShell`-t, majd a következő parancsot kell kiadni:

```
> cabal update
> cabal v1-install split
```

A `wordsBy` helyett használhattuk volna a korábban már bemutatott `lines` függvényt is, a kódsorban a megfelelő műveletsort megjegyzésben tüntettük fel. Figyeljük meg, hogy az adatok beolvasása és kiíratása során hogyan használtuk a `map`, `mapM_` függvényeket.

```
import System.IO
import Data.List.Split ( wordsBy )

foOsszeg :: IO ()
foOsszeg = do
   outf <- openFile "eredmeny.txt" WriteMode
   inf <- openFile "szamok.txt" ReadMode
   temp <- olvasL inf
   let lsT = map auxF1 temp
   mapM_ (auxF2 outf) lsT
   hClose outf
   hClose inf
       where
       auxF1 :: [Int] -> (Int, [Int], [[Int]])
       auxF1 kLs = (sumV, ls, felSum2 sumV ls)
           where
           sumV = head kLs
           ls = tail kLs
       auxF2 :: Handle -> (Int, [Int], [[Int]]) -> IO ()
       auxF2 outf kLs = do
           hPutStrLn outf $ show k1 ++ " " ++
                     show k2 ++ "\n" ++ show k3
           hPutStrLn outf ""
               where
               (k1, k2, k3) = kLs

olvasL :: Handle -> IO [[Int]]
olvasL inf = do
   temp <- hGetContents inf
   let ls = map auxF $ wordsBy (== '\n') temp
   -- let ls = map auxF $ lines temp
   return ls
       where
       auxF :: String -> [Int]
       auxF ls = map (read :: String -> Int) $ words ls
```

A `szamok.txt` állományban levő adatok beolvasását az `olvasL` függvény valósítja meg, amelynek kimenete egy `[Int]` típusú elemekből álló lista

lesz, azaz minden egyes listaelem az állomány egy adott sorában levő számokat fogja jelölni. Egy adott listaelemre a szükséges számítási folyamatot az `auxF1` végzi el, és azért, hogy ez minden egyes listaelemre megtörténjen, a `map` paramétereként hívtuk meg. Az `auxF1`-be kerül tehát meghívásra az előző feladatnál megadott `felSum2`.

Az `eredmeny.txt` állományba való formázott kiíratásért az `auxF2` felelős, és hogy ez minden listaelemre megtörténjen, a `mapM_`-nek adtuk át a függvényt paraméterként.

Az `olvasL` függvény a `hGetContents` függvénnyel beolvassa az állomány teljes tartalmát, amelyet egy `String` típusú `temp` listába tesz. Ezt a `wordsBy` függvénnyel, az új sor jelek mentén feldarabolja, így az eredmény egy `String` típusú elemekből álló lista lesz. Ezek mindegyikére alkalmazza a `words` függvényt, majd a `words` után kapott listaelemeket a `read` függvénnyel `Int` típusú értékekké alakítja.

Ahogy korábban jeleztük, a `wordsBy` a `Data.List.Split` könyvtárban van, tulajdonképpen a `words` függvénynek az általánosítása, használatát a következő példák szemléltetik:

```
> ls = "Nyerges\tBucsin\tKalonda\tTolvajos"
> wordsBy (== '\t') ls
["Nyerges","Bucsin","Kalonda","Tolvajos"]

> ls = "Nyerges:Bucsin:Kalonda:Tolvajos"
> wordsBy (== ':') ls
["Nyerges","Bucsin","Kalonda","Tolvajos"]
```

8.6. feladat Írjunk egy Haskell-függvényt, amely meghatározza a Pascal-háromszög `n`-edik sorát, illetve `n`-edik sorának `k`-adik elemét.

A Pascal-háromszög a binomiális együtthatók háromszög formában való rendezését jelenti, ahol a binomiális együttható azt a pozitív egész számot jelenti, amely az $(1 + x)^n$ polinom x^k tagjának az együtthatója. Ez az érték tulajdonképpen egyenlő lesz `n` elem `k`-ad rendű kombinációinak a számával. A Pascal-háromszög kiíratása során az `n` értéke a sor, míg a `k` értéke az oszlop értékét jelöli, és fennáll: $n \geq k \geq 0$. Értékét direkt módon a következő képlettel is meg tudjuk határozni:

$$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}.$$

A binomiális együtthatók értékét azonban meg lehet határozni úgy is, hogy az `(n-1)`-edik sorban levő értékekből generáljuk az `n`-edik sorban levő értékeket, ahol a nulladik sorban az `1` érték található, amely $\binom{0}{0}$ értéknek felel meg.

A `pascal` függvény ezt a gondolatmenetet követi, ahol a függvény egy egyelemű listaként `1`-es értékkel definiálja a nulladik sort. A következőt, azaz az első sort a nulladik, a második sort az első stb. alapján fogja meghatározni. Az új sor első elemként mindig az `1`-es értéket rögzíti, majd az előző sor elemei alapján a többi elemet az `auxPascal` függvénnyel határozza meg úgy, hogy az előző sorban (listában) levő egymás melletti két elemet összeadja, majd a kapott értékekből felépíti az új listát, azaz az új sor elemeit. Az előző sor (lista) utolsó elemét pedig változatlanul teszi át az új listába.

Az algoritmus az első négy sornak megfelelő listákat a következőképpen határozza meg:

```
[1] → 1:[1] → [1,1]
[1,1] → 1: [1+1, 1] → [1,2,1]
[1,2,1] → 1: [1+2, 2+1, 1) → [1,3,3,1]
[1,3,3,1] → 1: [1+3, 3+3, 3+1, 1] → [1,4,6,4,1]
```

```
auxPascal :: Integral a => [a] -> [a]
auxPascal [k] = [k]
auxPascal (k1 : k2 : ve) = (k1 + k2) : auxPascal (k2 : ve)

pascal :: Integral a => Int -> [a]
pascal 0 = [1]
pascal n = 1 : auxPascal (pascal (n-1))

> pascal 5
[1,5,10,10,5,1]

> auxPascal [1, 5, 10, 10, 5, 1]
[6,15,20,15,6,1]
```

A `foPascal` függvény a bemeneti `n` és `k` értékeket a billentyűzetről olvassa be, az eredményt, amelyet a `pascal` függvény hoz létre, kiírja a képernyőre. Az `n`-edik sor `k`-edik elemét a `!!` operátorral választjuk ki.

```
foPascal :: IO ()
foPascal = do
   putStr "n = "
```

```
    temp <- getLine
    let n = read temp :: Int
    putStr "k = "
    temp <- getLine
    let k = read temp :: Int
    let pL = pascal n
    let str = "A Pascal-háromszög " ++ show n
                ++ "-edik sora: "
    putStr str
    print $ pL
    let str = "A sor " ++ show k ++ "-adik eleme: "
    putStr str
    print $ pL !! k
```

```
> foPascal
n = 10
k = 7
A Pascal-háromszög 10-edik sora: [1,10,45,120,210,...]
A sor 7-edik eleme: 120
```

8.7. feladat Írjunk egy Haskell-függvényt, amely kiírja a Pascal-háromszög első `n` sorát egy szövegállományba, háromszög formában.

A Pascal-háromszög első `n` sorának a generálását a `pascalN` fogja végezni, amely a korábbi `pascal` függvény módosított változata. A módosítás azt jelenteni, hogy a generált listákat egymás után fűzzük a `++` operátorral, az `auxPascal` függvény bemenetének pedig az eredménylistához utolsónak hozzáfűzött listát adjuk meg.

```
pascalN :: Integral a => Int -> [[a]]
pascalN 0 = [[1]]
pascalN n = temp ++ [1 : auxPascal (last temp)]
   where
   temp = pascalN (n - 1)
```

```
> pascalN 4
[[1],[1,1],[1,2,1],[1,3,3,1],[1,4,6,4,1]]
```

Ahhoz, hogy háromszög formában tudjuk megjeleníteni a számokat, a számok közé egy-egy szóközt, illetve minden egyes sor elé bizonyos számú szóközt kell tenni. A sorok elejére kerülő szóközök számát a `db` változó jelöli, és a `szokozok` függvényben kerül ez pontos meghatározásra. Ez a `db` érték

attól függ, hogy hány számjegy, illetve szóköz található a Pascal-háromszög legutolsó, illetve aktuális sorában.

```
szokozok :: String -> Int -> String
szokozok ls i = replicate db ' '
   where
   db = div (i - length ls) 2
```

A Pascal-háromszög legutolsó sorában található szóközök és számjegyek számát az `i` konstans jelöli, és a `printPascal` függvényben kerül meghatározásra. A `listaToStr` függvény első paramétere ez az `i` érték lesz, második paraméterként pedig megkapja a Pascal-háromszög soraiban levő számokat. A `listaToStr` függvény az `auxSor`-t alkalmazva, betesz az aktuális sor elejére `db` darab szóközt, majd az aktuális sorban levő további számokat az `auxSz`-et meghívva, szóközökkel elválasztva összefűzi. Mindezt egy `map` segítségével elvégzi az összes listára, ami a bemenetben szerepel.

```
listaToStr :: Show a => Int -> [[a]] -> [String]
listaToStr i = map (auxF i)

auxF :: Show a => Int -> [a] -> String
auxF i k = auxSor i k ++  auxSz k

auxSor :: Show a => Int -> [a] -> String
auxSor i k = szokozok (unwords $ map show k) i

auxSz :: Show a => [a] -> String
auxSz [] = ""
auxSz (k: ve)= show k ++ " " ++ auxSz ve
```

Az állományba való kiíratást a `printPascal` függvény végzi, ez lesz a feladat főfüggvénye, amelynek paraméterként meg kell adni az állomány nevét, illetve hogy hány sort kell meghatározni a Pascal-háromszögből.

```
printPascal :: FilePath -> Int -> IO ()
printPascal nev n = do
   let pLs = pascalN n
   let i = length $ unwords $ map show $ last pLs
   let tLs = listaToStr i pLs
   outf <- openFile nev WriteMode
   mapM_ (hPutStrLn outf) tLs
   hClose outf

> printPascal "pascal5.txt" 5
```

A `pascal5.txt` állomány tartalma a `> printPascal "pascal5.txt" 5` meghívás után a következő lesz:

```
      1
     1 1
    1 2 1
   1 3 3 1
  1 4 6 4 1
1 5 10 10 5 1
```

8.8. feladat Írjunk egy Haskell-függvényt, amely római számokat alakít át arab számokká, majd alkalmazzuk a függvényt egy szöveges állományra úgy, hogy a szövegállományban előforduló római számokat alakítsuk át arab számmá.

A római számok felírásához összesen `7` betűt használnak: `I, V, X, L, C, D, M`, és a következő megfeleltetés érvényes:

`I = 1, V = 5, X = 10, L = 50, C = 100, D = 500, M = 1000.`

A legnagyobb szám, amit le lehet írni ezekkel a betűkkel, az a `3999`. A számok megadása során a `7` betűt megadott szabályrendszer szerint kell kombinálni, és ahhoz, hogy nagyobb számértékeket is meg tudjunk adni, mint `3999`, a fenti `7` betűt függőleges vagy vízszintes vonalakkal együtt kell használni.

A programkódot a szükséges `import`-ok megadásával kezdjük, majd egy konstans értékekből álló `romLs` listát hozunk létre, amelyben definiáljuk a római számok esetében használható betűkészletet, a megfelelő arab számértékekkel. Az átalakítás során feltételezzük, hogy a legnagyobb szám, amit át akarunk alakítani, az a `3999`.

```
import Data.List (isPrefixOf)
import System.IO

romLs :: [(Int, [Char])]
romLs = [(1000,"M"),(900,"CM"),(500,"D"),(400,"CD"),
         (100,"C"),(90,"XC"),(50,"L"),(40,"XL"),
         (10,"X"),(9,"IX"),(5,"V"),(4,"IV"),(1,"I")]
```

Egy római szám arab számmá való átalakításához a `toArabic` függvényt kell meghívni, amelynek kimenete egy `Maybe Int` típusú érték lesz, amely abban az esetben lesz `Nothing`, ha a bemenet nem alakítható át arab számmá:

```
toArabic :: [Char] -> Maybe Int
toArabic = toArabicAux 0 romLs

> toArabic "MMMCCCVIXXXIVV"
Nothing

> toArabic "MMMCCCXXXIV"
Just 3334
```

A tulajdonképpeni átalakítást a `toArabicAux` függvény végzi:

```
toArabicAux :: Int -> [(Int, [Char])] -> [Char]
               -> Maybe Int
toArabicAux db romLs str = case tNr of
   Just (nr, [], _, _) -> Just nr
   Just (nr, _, [], _) -> Nothing
   Just (nr, tRls, romLsN, nDb) ->
       if nDb >= 3 then Nothing
       else case toArabicAux nDb romLsN tRls of
               Just temp -> Just (nr + temp)
               Nothing -> Nothing
   Nothing -> Nothing
   where
   tLs = auxGen romLs str
   tNr = auxA db tLs
```

A `toArabicAux` függvény a harmadik paramétereként megadott, `str`-hez tartozó, arab számértéket próbálja meghatározni a `romLs`-ben megadott értékek alapján. Ennek érdekében a `tLs`-be, az `auxGen` függvény segítségével háromelemű tuple elemtípusú listát hoz létre, a bemeneti `str` lehetséges prefix értékei alapján. Az `auxGen` függvényben a prefix értékek meghatározását az `isPrefixOf` függvénnyel végezzük, amely a `Data.List` könyvtármodulban van. Az `isPrefixOf` megvizsgálja, hogy az első paramétereként megadott karakterlánc prefixe-e a második paraméterként megadott karakterláncnak.

```
auxGen :: [(Int, [Char])] -> [Char]
          -> [(Int, [Char], [Char])]
auxGen romLs str = [(nr, take (length rLs) str,
                  drop (length rLs) str) |
                  (nr,rLs) <- romLs, isPrefixOf rLs str]

> auxGen romLs "XLIV"
[(40,"XL","IV"),(10,"X","LIV")]
```

```
> auxGen romLs "XXXIII"
[(10,"X","XXIII")]
```

Az `auxA` függvény, amennyiben nem `Nothing` a visszatérési értéke, egy négyelemű tuple típust határoz meg. A tuple első eleme azt az arab számértéket fogja jelölni, amely társítható az első talált prefixhez. A második elem a fennmaradó szimbólumsorozat lesz, amelyen folytatni fogja a számolást. A harmadik elem a `romLs`-nek az a végrésze lesz, amely még szükséges a fennmaradó szimbólumsorozat kiértékeléséhez. A negyedik elem az egymás után következő, azonos szimbólumok számát jelenti. Erre azért van szükség, mert egymás után nem állhat több mint három `M, C, X` vagy `I` szimbólum.

```
auxA :: Int -> [(Int, [Char], [Char])]
       -> Maybe (Int, [Char], [(Int, [Char])], Int)
auxA _ [] = Nothing
auxA db tLs = Just (nr, tRls, romLsN, nDb)
   where
   (nr, hRls, tRls) = head tLs
   (nDbN, romLsN) = auxR db (take 1 $ show nr) hRls romLs
   nDb =
       if head hRls /= head tRls then 0
       else nDbN

> auxA 0 $ auxGen romLs "XXXIII"
Just (10,"XXIII",[(10,"X"),(9,"IX"),(5,"V"),
                  (4,"IV"),(1,"I")],1)
```

Az `auxR` visszatérési értéke egy kételemű tuple elemtípusú lista, amelynek első eleme `db`, a második pedig a `romLs`-en végzett változtatásokat jelöli (a megfelelő végrészt határozza meg):

```
auxR :: Int -> [Char] -> [Char] -> [(Int, [Char])]
        -> (Int, [(Int, [Char])])
auxR db nStr cStr romLs = case nStr of
   "1" -> (db+1, ls)
   "9" -> (0, drop 4 ls)
   "5" -> (0, drop 2 ls)
   "4" -> (0, drop 2 ls)
   where
   auxD c (t1, t2) = t2 /= c
   ls = dropWhile (auxD cStr) romLs
```

```
> auxR 0 (take 1 $ show 10) "X" romLs
(1,[(10,"X"),(9,"IX"),(5,"V"),(4,"IV"),(1,"I")])
```

Visszatérve az eredeti feladathoz, az állományban levő római számok átalakítását a `foRomaiSz` függvény végzi, amelyben egy `szovegRomai.txt` nevű állományt dolgozunk fel, amely letölthető a következő linkről:

https://ms.sapientia.ro/~mgyongyi/Funk_Log/Jegyzet/szovegRomai.txt

A `foRomaiSz` függvény az állomány minden egyes sorát szavakra bontja a `words` függvénnyel, majd a `myPhrase`-ben ellenőrzi, hogy római számról van-e szó. Az állományban római számot képezhet egy római számjegyekből álló karakterlánc, amelyet a `testRoman` függvény vizsgál, amely ugyanakkor szóközök között helyezkedik el, például:

```
...it stands LCCCXXVIII meters...
```

vagy a római számjegyekből álló karakterláncot követheti valamilyen írásjel, például:

```
...was completed in MDCCCLXXXIX. It...
```

Az átalakítást a korábban bemutatott `toArabic` függvénnyel végezzük, és abban az esetben, ha ez lehetséges, meghatározzuk a megfelelő arab számot. Ellenkező esetben a `NONE` karakterlánc lesz a `myPhrase` kimeneti értéke. A szavakból az `unwords` függvénnyel hozzuk létre azt a karakterláncot, amit kiírunk a képernyőre.

```
foRomaiSz :: IO()
foRomaiSz = do
   ls <- readFile "szovegRomai.txt"
   mapM_ (putStrLn . unwords . aux) $ lines ls
       where
       aux str = map myPhrase $ words str

myPhrase :: [Char] -> [Char]
myPhrase k =
   | lK `elem` ".,!?;:"
       if testRoman iK then
           case tRes of
               Just temp -> show temp ++ [lK]
               Nothing -> "NONE"
       else k
   | otherwise =
       if testRoman k then
           maybe "NONE" show tResK
```

```
        else k
    where
    lK = last k
    iK = init k
    tRes = toArabic iK
    tResK = toArabic k

    testRoman :: [Char] -> Bool
    testRoman = all aux
        where
        aux k = elem k "IVXLCDM"
```

8.9. feladat Írjunk Haskell-függvényt, amely egy arab számot átalakít római számmá. Abban az esetben, ha a bemeneti arab szám nagyobb mint 3999, írassunk ki hibaüzenetet.

A toRoman függvény paramétere az átalakítandó nr szám, amelynek kiértékelt értéke egyenlő lesz az auxRoman függvény eredményével. Az auxRoman számjegyenként végzi az átalakítást, ezért hátulról számjegyekre bontja a nr-t, a p-ben pedig számolni fogja, hogy hányadik számjegynél tart.

```
toRoman :: Int -> [Char]
toRoman nr = auxRoman nr 0
   where
   auxRoman :: Int -> Int -> [Char]
   auxRoman nr p
       | nr <= 0 = ""
       | otherwise = auxRoman (div nr 10) (p+1) ++
              toRomanDigit p (mod nr 10)

> toRoman 4000
*** Exception: tul nagy romai szam...

> toRoman 3999
"MMMCMXCIX"
```

Egy adott számjegy átalakítását a toRomanDigit függvény végzi, és aszerint, hogy hátulról a hányadik, illetve melyik számjegy átalakításán van a sor, aszerint kerülnek meghívásra, megfelelő paraméterekkel, a toROne vagy toRth függvények. A toROne függvényt akkor hívja meg, ha az utolsó, az utolsó előtti vagy a százasok helyén álló számjegyet kell helyettesíteni. Az utolsó számjegy helyettesítéséhez az I, V, X szimbólumokat, az utolsó

előttihez az `X, L, C`-ket alkalmazzza, míg a százasok helyén álló számjegy esetében `C, D, M` szimbólumok lesznek a függvény paraméterei. A függvény negyedik `d` paramétere, abban az esetben, ha a `2, 3, 6, 7, 8` számjegyeket kell helyettesíteni, azt fogja jelezni, hogy egy adott római számjegyet hányszor írjunk egymás mellé.

A `toRth` függvénnyel annyi darab `M` szimbólumot fogunk előállítani, amennyit az ezresek helyértéke mutat.

```
toRomanDigit :: Int -> Int -> [Char]
toRomanDigit p d =
       case p of
           0 -> toROne 'I' 'V' 'X' d
           1 -> toROne 'X' 'L' 'C' d
           2 -> toROne 'C' 'D' 'M' d
           3 ->
               if d < 4 then toRTh 'M' d
               else error "tul nagy romai szam"
           _ -> error "tul nagy romai szam"

toROne :: Char -> Char -> Char -> Int -> [Char]
toROne sz1 sz2 sz3 d
   | d < 4 = replicate d sz1
   | d == 4 = sz1 :  [sz2]
   | d == 5 = [sz2]
   | d == 9 = sz1 : [sz3]
   | otherwise = sz2 : replicate (d - 5) sz1

toRTh :: Char -> Int -> [Char]
toRTh sz d = replicate d sz
```

A következő lekérdezések azt is mutatják, hogy miként történik számjegyenként az átalakítás:

```
> toRoman 2378
"MMCCCLXXVIII"

> toRomanDigit 0 8
"VIII"
> toRomanDigit 1 7
"LXX"
> toRomanDigit 2 3
"CCC"
> toRomanDigit 3 2
"MM"
```

8.2. Bináris keresés

A bináris keresés a keresési algoritmusok egyik legfontosabb és legegyszerűbb keresési módszere, ahol a keresés során feltételezzük, hogy egy rendezett adathalmazban kell egy adott értékű elemet megkeresnünk. A bemutatásra kerülő algoritmusban feltételezzük, hogy az adathalmazban növekvő sorrendben vannak az elemek. Az adathalmaz középső eleménél kisebb elemek által meghatározott intervallumot alsó intervallumnak, a nagyobb elemek által meghatározott intervallumot felső intervallumnak fogjuk nevezni. A keresési algoritmus szerint a következőképpen járunk el: a keresett elemet mindig az adathalmaz középső pozícióján elhelyezkedő elemmel hasonlítjuk össze, és ha ennél az elemnél nagyobb a keresett elem, akkor a keresést a felső intervallumban folytatjuk, ellenkező esetben az alsó intervallumban. Találat esetén abbahagyjuk a keresést, ellenkező esetben a kiválasztott felső vagy alsó intervallumban hasonló elgondolás alapján folytatjuk a keresést, amíg végig nem mentünk az intervallumon.

A továbbiakban bemutatásra kerül a bináris keresés, különböző típusú adatszerkezetek esetében.

8.10. feladat Írjunk egy Haskell-függvényt, amely egy tetszőleges típusú elemeket tartalmazó rendezett **lista** esetében, bináris keresést alkalmazva, találat esetén megadja a keresett elem pozícióját, ellenkező esetben pedig `-1` lesz az általa meghatározott érték.

```
foKeres1 :: Ord a => [a] -> a -> IO()
foKeres1 ls x = do
   let m = length ls
   let res = bKeres1 ls x 0 (m - 1)
   if res == -1 then putStrLn "Nincs benne!!"
   else putStrLn $ "a " ++ show res ++ ". pozicion van!!"

bKeres1 :: Ord a => [a] -> a -> Int -> Int -> Int
bKeres1 ls k l h
   | h < l = -1
   | k < p = bKeres1 ls k l (i-1)
   | k > p = bKeres1 ls k (i+1) h
   | otherwise = i
       where
       i = div (l + h) 2
       p = ls !! i
```

```
> ls = [1,5,7,9,10,11,21,34,55,67,90]
> foKeres1 ls 21
a 6. pozicion van!!

> foKeres1 "abcdefgh" 'z'
Nincs benne!!
```

A `foKeres1` függvénynek az első paramétere egy rendezett lista, második paramétere pedig a keresett elem lesz. A tulajdonképpeni keresést a `bKeres1` függvény végzi, amelynek négy paramétert kell megadnunk, az első a rendezett lista lesz, amiben végezzük a keresést, a második a keresett elem, a harmadik és negyedik pedig a keresési intervallum alsó, illetve felső határértéke. A középső elem kiválasztása azok után lehetséges, hogy meghatározzuk a keresési intervallum középső elemének pozícióját. Ezt az értéket a `bKeres1` függvényben az `i` jelöli, a középső elemet pedig a `p`. Az algoritmus hatékonyság szempontjából nem közelíti meg a megszokott bináris keresési algoritmusok hatékonyságát, mert a lista adatszerkezet használata miatt a középső elem kiválasztása nem konstans, hanem lineáris időben történik.

8.11. feladat Írjunk egy Haskell-függvényt, amely egy tetszőleges típusú elemeket tartalmazó rendezett **lista** esetében, bináris keresést alkalmazva, találat esetén a `Benne van!!` üzenetet, ellenkező esetben pedig a `"Nincs benne!!"` üzenetet írja ki a képernyőre.

```
foKeres2 :: Ord a => [a] -> a -> IO()
foKeres2 ls x = do
   let res = bKeres2 ls x
   case res of
       Nothing -> putStrLn "Nincs benne!!"
       Just k -> putStrLn "Benne van!!"

bKeres2 :: Ord a => [a] -> a -> Maybe a
bKeres2 [] _ = Nothing
bKeres2 ls k
   | k < p = bKeres2 (take i ls) k
   | k > p = bKeres2 (drop (i+1) ls) k
   | otherwise = Just k
       where
       i = div (length ls - 1) 2
       p = ls !! i

> ls = [1,5,7,9,10,11,21,34,55,67,90]
```

```
> foKeres2 ls 100
Nincs benne!!

> foKeres2 "abcdefgh" 'a'
Benne van!!
```

A `bKeres2` függvény által meghatározott érték egy `Maybe a` típusú adat lesz. Ha a keresett elem nincs benne az adathalmazban, `Nothing` lesz a függvény kimenete. Találat esetén egy `Just k` értéket határoz meg, ahol `k` a keresett elem. A `foKeres2` függvényben tehát aszerint választunk a `case`-ben megadott kiíratási szövegek között, hogy milyen kimeneti értéket határoz meg a `bKeres2`. A `bKeres2` függvény még abban is különbözik a `bKeres1`-től, hogy csak két paraméterrel dolgozik. A keresési intervallum kiválasztása itt nem az intervallum határértékeinek a változtatásával kerül meghatározásra, hanem a `take`, illetve `drop` függvények segítségével módosítjuk a teljes keresési intervallumot.

A fejezet további részében azt vizsgáljuk, hogy sikerül-e a bináris keresés hatékonyságán javítani, ha a lista adatszerkezet helyett **tömb** adatszerkezettel dolgozunk.

A Haskell is rendelkezik beépített tömb típussal, mégpedig nem is eggyel, hanem többfélével is. A változhatatlan tömb, az *immutable array* típus, amivel a fejezetben dolgozni fogunk, a `Data.Array` könyvtármodulban van definiálva. Az immutable tulajdonság azt jelenti, hogy a már létrehozott tömb elemeit nem változtathatjuk meg, módosításokat csak úgy végezhetünk, ha egy új tömböt hozunk létre, ami a változtatásokat is tartalmazza.

A Haskell `listArray` függvénye egy tömböt hoz létre a megadott listaelemekből, ahol a tömb elemeinek indexértékeit a tuple-ként megadott értékek alapján határozza meg.

```
> import Data.Array
> lA = listArray (3, 5) "Kiraly-hago"
> lA
array (3,5) [(3,'K'),(4,'i'),(5,'r')]

> lA = listArray (0, 20) "Kiraly-hago"
> lA
array (0,20) [(0,'K'),(1,'i'),(2,'r'),(3,'a'),
 (4,'l'),(5,'y'),(6,'-'),(7,'h'),(8,'a'),
 (9,'g'),(10,'o'),(11,> *** Exception:
 (Array.!): undefined array element
```

```
> lA = listArray (0, 3) "Kiraly-hago"
> lA
array (0,3) [(0,'K'),(1,'i'),(2,'r'),(3,'a')]
```

Vegyük észre, hogy ha nem megfelelően adjuk meg a listaelemeket és indexértékeket, akkor hibaüzenetet is kapunk. Az ilyen módon létrehozott tömb típusa paraméterezett `Array` típus lesz, ami a következő lekérdezésből is jól látszik, ahol az `Ix` típusosztályt egy tömb elemeinek indexelésekor kell használni.

```
> :t lA
lA :: (Ix i, Num i) => Array i Char
```

A tömb létrehozása után egy megadott sorszámú elem kiválasztása a **!** operátorral történik. A `bounds` függvénnyel lekérhetjük a tömb kezdeti és végső elemének indexértékét, az `indices` az index értékeket, az `elems` a tömb elemeit adja meg, míg az `assocs` az index értékek és az elemek közötti társítást mutatja meg. Ezeket próbálhatjuk ki a következő lekérdezésekkel:

```
> lA ! 1
'i'

> lA ! 0
'K'

> bounds lA
(0,3)

> indices lA
[0,1,2,3]

> elems lA
"Kira"

> assocs lA
[(0,'K'),(1,'i'),(2,'r'),(3,'a')]
```

Korábban említettük, hogy az így létrehozott tömbök immutable tömbök, amelyeknek elemei olyan értelemben változtathatók meg, hogy minden egyes változtatás maga után vonja egy új tömb létrehozását. A változtatáshoz a **//** operátort kell használni:

```
> lA // [(0,'k')]
array (0,3) [(0,'k'),(1,'i'),(2,'r'),(3,'a')]

> lA
array (0,3) [(0,'K'),(1,'i'),(2,'r'),(3,'a')]

> lA1 = lA // [(0,'k')]

> lA1
array (0,3) [(0,'k'),(1,'i'),(2,'r'),(3,'a')]
```

A fejezet további részében a tömb típus használatával mutatjuk be a bináris keresést.

8.12. feladat Írjunk egy Haskell-függvényt, amely tetszőleges típusú elemeket tartalmazó rendezett **tömb** esetében, bináris keresést alkalmazva, találat esetén megadja a keresett elem pozícióját, ellenkező esetben pedig a `"Nincs benne!!"` üzenetet írja ki a képernyőre.

```
import Data.Array
foKeres3 :: [Int] -> Int -> IO()
foKeres3 ls x = do
   let m = length ls
   let ar = listArray (0, m-1) ls
   let res = bKeresA ar x 0 (m-1)
   case res of
       Nothing -> putStrLn "Nincs benne!!"
       Just k -> putStrLn $ "a " ++ show k ++
                       ". pozicion van!!"

bKeresA :: Array Int Int -> Int -> Int -> Int ->
                                    Maybe Int
bKeresA ar k l h
   | h < l = Nothing
   | k < p = bKeresA ar k l (i - 1)
   | k > p = bKeresA ar k (i + 1) h
   | otherwise = Just i
       where
       i = div (l + h) 2
       p = ar ! i

> ls = [1,5,7,9,10,11,21,34,55,67,90]
> foKeres3 ls 10
a 4. pozicion van!!
```

A tulajdonképpeni keresést a `bKeresA` függvényben végezzük, amelynek négy bemeneti paramétere van, amelyek szerepe ugyanaz, mint a `bKeres1` függvény esetében a bemeneti paramétereké. Ha nem találjuk meg a keresett elemet, akkor a kimenet `Nothing` lesz, találat esetén pedig `Just k` lesz, ahol `k` a megtalált elem tömbbeli pozíciója.

Ha össze szeretnénk hasonlítani a három algoritmus hatékonyságát egy `10000000` elemű adathalmazon, akkor a következőképpen járjunk el:

```
> m = 10000000
> ls = [1..m]
> :set +s
> bKeres1 ls 13 0 (m-1)
12
(0.77 secs, 360,061,584 bytes)

> bKeres2 ls 13
Just 13
(1.36 secs, 1,280,062,448 bytes)

> ar = listArray (0, m-1) ls
(0.02 secs, 0 bytes)
> bKeresA ar 13 0 (m-1)
Just 12
(0.90 secs, 800,144,464 bytes)
```

8.13. feladat Írjunk egy Haskell-programot, amely a `Datum` és `Személy` adatszerkezeteket használva megállapítja egy adott személyről, hogy mikor van a névnapja.

```
data Datum = Datum {
   dNap :: Int,
   dHonap:: Int,
   dEv :: Int
   } deriving (Show)

data Szemely = Szemely {
   szVnev :: [Char],
   szKnev :: [Char],
   szSzulD :: Datum
   } deriving (Show)
```

A névnapok megállapításához használjuk a `nevnapok.txt` állományt, amely letölthető a következő linkről:

https://ms.sapientia.ro/~mgyongyi/Funk_Log/Jegyzet/nevnapok.txt

A `nevnapok.txt` állományban ábécésorrendben találhatók a különböző nevek úgy, hogy minden egyes sorban egy név szerepel, és a név után zárójelben megjelennek a névnapok. A nevek írásakor nincsenek ékezetes betűk használva. A különböző betűvel kezdődő névnapok között üres sorok vannak, illetve az azonos kezdőbetűvel kezdődő nevek előtt fel van tüntetve a kezdőbetű.

A feladat főfüggvénye a `foNevnap` függvény lesz, amely kiolvassa a `hGetContents` segítségével a `nevnapok.txt` teljes tartalmát. A `splitOn` függvénnyel ezt az értéket feldaraboljuk úgy, hogy az eredmény egy `String` elemtípusú lista legyen, amely alapján létrehozzuk az `ar` tömböt. A `foNevnap` az `ar` tömbben bináris kereséssel határozza meg egy adott személy névnapját, ahol a keresett személy adatait konstans értékként adjuk meg, de ezt akár be lehetne kérni a billentyűzetről vagy be lehetne olvasni egy állományból.

```
import Data.Array ( Array, (!), listArray )
import Data.List.Split ( splitOn )
import Control.Exception ( SomeException, catch )
import System.IO

foNevnap :: IO ()
foNevnap =
   catch ( do
       inf <- openFile "nevnapok.txt" ReadMode
       str <- hGetContents inf
       let ls = splitOn "\n" str
       let m = length ls
       let ar = listArray (0, m-1) ls
       let egySz = Szemely "Kiss" "Szabolcs"
                   (Datum 10 5 1945)
       let res = bKeresA1 compA ar (szKnev egySz) 0 (m-1)
       if res == -1 then print "Nincs benne!!"
       else print $ ar ! res
       hClose inf
       ) hibaKezelo
       where
       hibaKezelo :: SomeException -> IO ()
       hibaKezelo err = putStrLn $ "IO hiba!!: "
                        ++ show err

> foNevnap
"Szabolcs (julius 17., julius 28., szeptember 19.)"
```

A `splitOn` két paraméteres, és az első paramétere alapján egy tetszőleges elemtípusú listát részlistákra darabol fel.

```
splitOn :: Eq a => [a] -> [a] -> [[a]]
```

Használatához importálni kell a `Data.List.Split`-et. A következő lekérdezések egy kis kitérőt jelentenek, mert a `splitOn` függvény használatát mutatják be:

```
> import Data.List.Split

> splitOn "0" "120123012340 12"
["12","123","1234","12"]

> splitOn "00" "11001001111010001111101"
["11","1","111101","1111101"]
```

A `bKeresA1` bináris keresést alkalmaz azért, hogy a paraméterként megadott keresztnevet megkeresse az `ar` tömbben. Az elemek összehasonlítását az első paramétereként megadott `fg` függvény szerint végzi, amely a `bKeresA1` meghívásakor a `compA` függvényértéket fogja felvenni. Találat esetén a függvény kimenete a keresett elem tömbbeli pozíciója lesz, ellenkező esetben pedig `-1` lesz.

```
bKeresA1 :: ([Char] -> [Char] -> Ordering) ->
            Array Int [Char] -> [Char] -> Int -> Int -> Int
bKeresA1 fg a k l h
    | h < l = -1
    | cRes == LT = bKeresA1 fg a k l (i - 1)
    | cRes == GT = bKeresA1 fg a k (i + 1) h
    | cRes == EQ = i
        where
        cRes = fg k aElem
        i = div (l + h) 2
        aElem = a ! i
```

A `compA` függvény a `compare` beépített függvény segítségével oldja meg a nevek, azaz a `String` típusú értékek összehasonlítását, ahol a második paramétere az `ar` tömb egy eleme lesz, amelynek szerkezetét a következő példa mutatja: `Abbas (november 12.)`. A `compA`-ban a `takeWhile`-t alkalmaztuk, hogy a tulajdonképpeni nevet meghatározhassuk, azaz lekértük a karaktereket az első szóközig.

```
compA :: [Char] -> [Char] -> Ordering
compA s1 s2 = compare s1 nS2
        where
        nS2 = takeWhile (/= ' ') s2
```

8.14. feladat Írjunk egy Haskell-programot, amely az előző feladatnál definiált `Datum` és `Személy` adatszerkezeteket használva, kiolvassa különböző személyek adatait a `szemely.txt` állományból, beolvas egy időpontot, és meghatározza azokat a személyeket, akik a beolvasott időpontban születtek.

A `szemely.txt` állományban a személyekre vonatkozó adatok soronként vannak eltárolva, ahol minden sor tartalmazza, szóközökkel elválasztva, a személy vezetéknevét, keresztnevét, születési dátumát nap, hónap és év szerint. Feltételezhetjük, hogy az állomány első pár sora a következő:

```
Nagy Ferenc 25 1 1998
Bandi Zoltan 7 1 1999
Joo Monika 25 1 2000
```

A keresett születési dátumot a billentyűzetről olvassuk be, a keresés előtt rendezzük az adatokat, a születési időpont szerint növekvő sorrendbe, majd bináris keresést alkalmazva, meghatározzuk a beolvasott időpontban született személyek listáját.

A feladat főfüggvénye a `foSzulnapok`, amelyben a meghívásra kerülő `dateProc, compQ, strProc, bKeresA2, auxF, myPrint` függvényeket a továbbiakban fogjuk elmagyarázni, a `quickSort` függvényt pedig a `6.1` fejezetben adtuk meg.

```
import Data.Array ( Array, (!), listArray )
import Data.List.Split ( splitOn )
import Control.Exception ( SomeException, catch )
import System.IO

foSzulnapok :: IO ()
foSzulnapok =
   catch ( do
       putStr "keresett datum (nap honap ev): "
       temp <- getLine
       let szem = dateProc temp
       inf <- openFile "szemely.txt" ReadMode
       temp <- hGetContents inf
       let lsSz = map strProc $ splitOn "\n" temp
       let ls = quickSort compQ lsSz
       let m = length ls
       let ar = listArray (0, m-1) ls
       let res = bKeresA2 comp ar szem 0 (m-1) (-1)
       if res == -1 then
           print "Nincs ilyen szuletesi datum!!"
       else do
```

```
            putStrLn "Keresett szemely(ek): "
            let tLs = [ar ! i | i <- [res..]]
            let szD = szSzulD szem
            let rLs = takeWhile (auxF szD) tLs
            mapM_ myPrint rLs
        hClose inf
    ) fExp
    where
    fExp :: SomeException -> IO ()
    fExp err = putStrLn $ "IO hiba!!: " ++ show err
```

A `dateProc` függvény a születési időpontot jelölő `String` típusú paramétere alapján létrehoz egy `Szemely` típusú adatot, amelynek `szSzulD` mezőértékét a megadott paraméter alapján inicializálja, a `szVnev`, `szKnev` mezők azonban üres `String`-ek lesznek.

```
dateProc :: String -> Szemely
dateProc str = Szemely "" "" d
   where
   ls = words str
   nap = read (ls !! 0) :: Int
   honap = read (ls !! 1) :: Int
   ev = read (ls !! 2) :: Int
   d = Datum nap honap ev

> dateProc "13 06 1972"
Szemely {szVnev = "", szKnev = "",
   szSzulD = Datum {dNap = 13, dHonap = 6, dEv = 1972}}
```

A `strProc` függvény a bemeneti `String` típusú adat alapján létrehoz egy `Szemely` típusú adatot.

```
strProc :: String -> Szemely
strProc iLs = Szemely vNev kNev kD
   where
       ls = words iLs
       vNev = ls !! 0
       kNev = ls !! 1
       nap = read (ls !! 2) :: Int
       honap = read (ls !! 3) :: Int
       ev = read (ls !! 4) :: Int
       kD = Datum nap honap ev

> strProc "Nagy Ferenc 25 1 1998"
```

```
Szemely {  szVnev = "Nagy", szKnev = "Ferenc",
           szSzulD = Datum {dNap = 25,
           dHonap = 1, dEv = 1998}}
```

A `strProc` függvény a `foSzulnapok`-ban, egy `map` keretén belül kerül meghívásra, ami azt fogja eredményezni, hogy az állomány soraiban található személyi adatok alapján létrejön egy `Szemely` elemtípusú lista. A születési időpontok alapján való rendezést a korábban megadott `quickSort` függvénnyel végezzük, ahol az összehasonlító `fg` függvény a `compQ` lesz. A `compQ` függvényben a `comp` egy-egy `Datum` típusú adat összehasonlítását fogja végezni, ahol a `Datum` mező értékeinek az összehasonlításához a `compare`-t használjuk.

```
compQ :: Szemely -> Szemely -> Bool
compQ k1 k2
   | temp == LT = True
   | otherwise = False
   where
   temp = comp (szSzulD k1) (szSzulD k2)

comp :: Datum -> Datum -> Ordering
comp d1 d2
   | tEv /= EQ = tEv
   | tHonap /= EQ = tHonap
   | otherwise = tNap
           where
           tEv = compare (dEv d1) (dEv d2)
           tHonap = compare (dHonap d1) (dHonap d2)
           tNap = compare (dNap d1) (dNap d2)
```

A bináris keresést a `bkeresA2` függvényben implementáltuk, a megszokott algoritmushoz képest azonban módosítottuk azért, hogy az ugyanazon a napon született személyek mindegyikét meg tudjuk találni. Ennek érdekében az algoritmusnak lesz egy hatodik paramétere, az `r`, amely kezdetben `-1`, és amelynek értékét csak akkor módosítjuk, amikor találat van. Ugyanekkor módosítani fogjuk a keresési intervallum felső határértékét is azért, hogy a legelső személy pozícióját tudjuk meghatározni azon személyek közül, akiknek ugyanazon a napon van a születésnapjuk. A `bKeresA2` tehát meghatározza az első olyan személy pozícióját a rendezett listából, akinek a születési dátuma egyenlő a beolvasott születési dátummal. A `foSzulnapok` függvényben pedig ettől a pozícióértéktől kezdődően a `rLs` listába áttesszük

az összes olyan személyt, akinek a születési dátuma egyenlő a beolvasott születési dátummal.

```
bKeresA2 :: (Datum -> Datum -> Ordering)
            -> Array Int Szemely
            -> Szemely -> Int -> Int -> Int -> Int
bKeresA2 fg a x l h r
        | h < l = r
        | cRes == LT = bKeresA2 fg a x l (i - 1) r
        | cRes == GT = bKeresA2 fg a x (i + 1) h r
        | otherwise = bKeresA2 fg a x l (i - 1) i
            where
            cRes = fg (szSzulD x) (szSzulD aElem)
            i = div (l + h) 2
            aElem = a ! i
```

Egy `Szemely` típusú adat kiíratását a `myPrint` függvénnyel oldottuk meg:

```
myPrint :: Szemely -> IO ()
myPrint k = do
        putStrLn $ szVnev k ++ " " ++ szKnev k ++ " " ++
            show (dEv d) ++ " " ++
            show (dHonap d) ++ " " ++ show (dNap d)
            where
                d = szSzulD k
```

A `auxF` függvényt a talált személyek kiválogatásakor használjuk, a `takeWhile` függvény paramétereként:

```
auxF :: Datum -> Szemely -> Bool
auxF d elem = r == EQ
   where
   r = comp d (szSzulD elem)
```

8.3. A ByteString típus

8.15. feladat A `film.txt` szövegállományban egy adott filmről `9` típusú adat van eltárolva: megjelenési év, filmcím, a film hossza, a film típusa, népszerűségi index, díjazott-e a film, a főszerepet játszó színész, a színésznő és a rendező. Írjunk egy Haskell-programot, amely meghatározza az `fEv`-ben készült filmek listáját, ahol az `fEv` értékét a billentyűzetről olvassuk be.

A `film.txt`-ben a filmekre vonatkozó adatok sorokba vannak tördelve. Egy sorban, a különböző típusú adatok között tabulátor jel van és `Unknown` jelenik meg, ha valamely adatra vonatkozóan nincs meghatározott érték. Egy ilyen szerkezetű állomány letölthető a következő linkről:

https://ms.sapientia.ro/~mgyongyi/Funk_Log/Jegyzet/film.txt

A példát többféleképpen oldjuk meg, az első két módszernél a `hGetContents` függvénnyel fogjuk az állomány tartalmát beolvasni, majd a `lines` segítségével végezzük el a soronkénti feldolgozást. A harmadik megoldáshoz bevezetjük a `ByteString`-eket, amelyek az állományok egy hatékonyabb feldolgozási módját teszik lehetővé. Első lépésben mindhárom megoldás bekér a billentyűzetről egy évszámot, amelyet az `fEv` fog jelölni.

Az első megoldásnál a főfüggvény a `foFilm1` lesz, amely egy `map` segítségével létrehozza a `fLs` listát. Az `fLs` lista `Nothing`-okat vagy az `fEv`-ben készült filmek címét mint `Just` értékek fogja tartalmazni. A `map` első paramétere a `strProc fEv` lesz, amely a `splitOn` segítségével, a tabulátorok mentén feldarabolja az állomány aktuális sorát. Így hozzá tud férni a sorban levő filmre vonatkozó adatokhoz. A `strProc` kimenete egy `Maybe String` típusú adat lesz, amely tartalmazni fogja a filmcímet, mint `Just` érték, vagy a `Nothing`-ot, abban az esetben, ha a film megjelenési éve nem egyezik meg az `fEv`-vel. A `foFilm1` a `mapM_` alkalmazásával írja ki a kiválogatott filmcímeket.

```
import Control.Exception ( SomeException, catch )
import Data.List.Split ( splitOn )
import System.IO

foFilm1 :: IO ()
foFilm1 = catch ( do
   putStr "ev: "
   tStr <- getLine
   let fEv = read tStr :: Int
   inf <- openFile "film.txt" ReadMode
   temp <- hGetContents inf
   let fLs = map (strProc fEv) $ lines temp
   mapM_ myPutStr1 fLs
   hClose inf
   ) fExp
   where
   fExp :: SomeException -> IO ()
   fExp err = putStrLn $ "IO hiba!!: " ++ show err
```

```
strProc :: Int -> String -> Maybe String
strProc fEv str = res
   where
   ls = splitOn "\t" str
   kEv = read (ls !! 0) :: Int
   kCim = ls !! 1
   res = if kEv == fEv then Just kCim else Nothing

myPutStr1 :: Maybe String -> IO ()
myPutStr1 k = case k of
       Just x -> putStrLn x
       Nothing -> putStr ""

> foFilm1
ev: 1950
Stromboli
All about Eve
...
```

A második megoldásnál a főfüggvény, a `foFilm2` egy `filter` segítségével kiválogatja azokat a sorokat az állományból, ahol a film megjelenési éve megegyezik `fEv`-el, az eredmény a `fLs` lista lesz. Ebben az esetben a `myPutStr`-ben fogjuk az aktuális sort a `splitOn`, függvénnyel a tabulátorok mentén feldarabolni, majd ezután a sorból a filmcímet kiválasztani és kiíratni.

```
foFilm2 :: IO ()
foFilm2 = catch ( do
   putStr "ev: "
   tStr <- getLine
   let fEv = read tStr :: Int
   inf <- openFile "film.txt" ReadMode
   temp <- hGetContents inf
   let fLs = filter (auxFilter . strProc fEv) $
                        lines temp
   mapM_ myPutStr fLs
   hClose inf
   ) fExp
   where
   fExp :: SomeException -> IO ()
   fExp err = putStrLn $ "IO hiba!!: " ++ show err

auxFilter :: Maybe a -> Bool
```

```
auxFilter val = case val of
       Just k -> True
       Nothing -> False

myPutStr :: String -> IO()
myPutStr str = putStrLn kCim
   where
   ls = splitOn "\t" str
   kCim = ls !! 1
```

A Haskell az állományfeldolgozás egy hatékonyabb módját a **ByteString** típussal végzi. A Haskell ezen belül is megkülönböztet lassú (lazy) és szigorú (strict) `Bytestring`-eket. A `Data.ByteString` könyvtármodulban vannak a szigorú feldolgozási mód szerint működő függvények, ahol minden karakter `8` biten van tárolva. A `Data.ByteString.Lazy`, illetve a `Data.ByteString.Lazy.Char8` könyvtármodulban pedig a lusta kiértékelési stratégia szerint működő függvények vannak, ahol egy `ByteString` elemei úgynevezett `chunk`-okban, azaz nagyobb darabokban vannak tárolva, amelyek mérete maximum `64` kilobájt.

Ezekben a könyvtármodulokban számos olyan függvény van, amelyek ugyanazt végzik, mint a `Prelude`-ben, a `Data.List`-ben vagy a `Data.Char`-ban stb.-ben található függvények, sőt nevük is megegyezik, ilyenek például a `lines`, a `words`, az `intercalate`, a `readFile` stb. Fontos tehát, hogy különbséget tegyünk az ugyanolyan nevű, de különböző könyvtármodulokban elhelyezkedő függvények között. Korábban szó volt, hogy egy könyvtármodul importálásakor, a `qualified` kulcsszó használatával lehetőség van egy névválasztásra, aminek segítségével explicit módon lehet jelezni, hogy mikor melyik konyvtármodul függvényeit akarjuk használni. Így fogunk most is eljárni, az `L8` névválasztással jelezni fogjuk, hogy mikor használjuk a `Data.ByteString.Lazy.Char` könyvtármodul függvényeit.

`ByteString` típusú adatok használatakor legtöbbször szükség van `String` és `ByteString` típusok közötti átalakításra, amelyeket a `pack`, illetve `unpack` függvények végeznek. A következő lekérdezések egyszerű példákon keresztül mutatják be ezeket az átalakításokat, illetve a `words` és az `intercalate` függvények használatát.

```
> import qualified Data.ByteString.Lazy.Char8 as L8
> ls = "Borgoi-hago Torcsvari-hago Gyimesi-hago"
> lsB = L8.pack ls

> lsBL = L8.words lsB
```

```
> lsBL
["Borgoi-hago","Torcsvari-hago","Gyimesi-hago"]

> lsB1 = L8.intercalate (L8.pack "#") lsBL
> lsB1
"Borgoi-hago#Torcsvari-hago#Gyimesi-hago"

> ls1 = L8.unpack lsB1
> ls1
"Borgoi-hago#Torcsvari-hago#Gyimesi-hago"
```

Érdemes megvizsgálni a különböző adatok típusát is, ezért próbáljuk ki a következőket: `:t ls`, `:t lsB`, `:t lsBL`, `:t lsB1`, `:t ls1`.

Általában a Haskellben a `""` közötti karakterszekvenciáknak `String` típusa van. Ha azonban engedélyezve van az **`OverloadedStrings`** nyelvi kiterjesztés, akkor a Haskell megengedi, hogy a `ByteStringek` megadásakor is használhassuk a `""` jelet, ilyenkor fölösleges a `pack`, `unpack` használata.

Hasonlóan más programozási nyelvekhez, a GHC Haskell is több pragmával, a fordító felé irányuló utasítással rendelkezik, amelyek segítségével befolyásolható a kód hatékonysága. A pragmákat `{-# pragma_nev...#-}` közé kell írni, és az állomány első sorában kell feltüntetni. A `LANGUAGE` pragma használatával nyelvi kiterjesztéseket lehet engedélyezni, alkalmazásával most az `OverloadedStrings` használatát tesszük lehetővé.

A `foBS`, illetve `foBS_` függvényekben, ahogy fennebb is tettük, egy `Bytestring`-ben a szóközöket `#`-re helyettesítjük. Vegyük észre, hogy a két függvényhívás ugyanazt eredményezi, a feldolgozásra kerülő karakterlánc értékadását azonban másképp oldják meg:

```
{-# LANGUAGE OverloadedStrings #-}
import qualified Data.ByteString.Lazy.Char8 as L8
import Data.List.Split ( splitOn )

foBS :: IO ()
foBS = do
   let lsBL = L8.words lsB
   print lsBL
   let lsB1 = L8.intercalate "#" lsBL
   print lsB1
       where
       lsB :: L8.ByteString
       lsB = "Borgoi-hago Torcsvari-hago Gyimesi-hago"
```

```
foBS_ :: IO ()
foBS_ = do
   let lsBL = L8.words $ L8.pack ls
   print lsBL
   let lsB1 = L8.intercalate "#" lsBL
   print lsB1
       where
       ls :: String
       ls = "Borgoi-hago Torcsvari-hago Gyimesi-hago"

> foBS
["Borgoi-hago","Torcsvari-hago","Gyimesi-hago"]
"Borgoi-hago#Torcsvari-hago#Gyimesi-hago"
```

A továbbiakban a feladat harmadik megoldását mutatjuk be, amelyben `ByteString` típust fogunk használni, és a `Data.ByteString.Lazy.Char8` könyvtármodulban levő függvények segítségével végezzük az állomány feldolgozását. A feladat főfüggvénye a `foBS1` lesz, ahol a `filter`-hez használt `auxFilter` függvény kódsorát a `foFilm2`-nél adtuk meg.

```
{-# LANGUAGE OverloadedStrings #-}
import qualified Data.ByteString.Lazy.Char8 as L8

foBS1 :: IO ()
foBS1 = do
   putStr "ev: "
   tStr <- getLine
   let fEv = read tStr :: Int
   temp <- L8.readFile "film.txt"
   let tLs = map (strProcBS fEv) $ L8.lines temp
   let fLs = filter auxFilter tLs
   let rLs = L8.intercalate "\n" $ map auxF fLs
   L8.writeFile "filmekL.txt" rLs
       where
       auxF = \ (Just k) -> k
```

Az állomány tartalmát a `readFile` függvénnyel olvassuk be, majd a soronkénti feldolgozását a `lines` függvénnyel végezzük. Az `fLs` most is a kiválogatott filmcímeket tartalmazza. A következő sorban az `intercalate` függvény segítségével `\n` jeleket fűzünk a filmcímek, azaz a listaelemek közé, és az így kapott `ByteString` típusú értéket írjuk ki a `writeFile` függvény segítségével a `filmekL.txt` állományba. Figyeljük meg, hogy

a Data.ByteString.Lazy.Char8 könyvtármodulból milyen függvényeket használtunk.

```
strProcBS :: Int -> L8.ByteString -> Maybe L8.ByteString
strProcBS fEv iLs = res
   where
   ls = L8.split '\t' iLs
   temp = L8.readInt (ls !! 0)
   Just (kEv, tLs) = temp
   kCim = ls !! 1
   res = if kEv == fEv then Just kCim else Nothing
```

A strProcBS függvényben történik az aktuális sor feldarabolása és a megfelelő filmcím kiválasztása. A split az első paramétere mentén végzi a feldarabolást, ami jelen esetben egy tabulátor.

A feldarabolt ByteString elemtípusú listából a nulladik elem fogja tartalmazni a film megjelenési évét, a következő a film címét, majd a film hossza következik, és így tovább, aszerint, ahogy azt korábban leírtuk. A nulladik elemet a továbbiakban Int típusú értékké alakítjuk, alkalmazva a readInt függvényt. A readInt a bemeneti ByteString-ben levő prefix számjegyekből létrehoz egy egész számot, majd az első olyan karaktertől kezdve, ahol már nem számjegyek vannak, létrehoz egy ByteString típusú adatot. Az egész szám a kimeneti tuple első eleme, míg a ByteString típusú adat a tuple második eleme lesz. A következő lekérdezés egy karakterláncon alkalmazza a readInt függvényt:

```
> import qualified Data.ByteString.Lazy.Char8 as L8
> ls = "1802 Kolozsvar december 15"
> L8.readInt $ L8.pack ls
Just (1802," Kolozsvar december 15")
```

A következő kódsor az eredeti példának egy újabb implementációja lesz, a legutolsóhoz képest pedig két helyen végzünk módosítást. A keresett évszámot nem alakítjuk át Int-té, sem a beolvasásnál, sem az összehasonlításnál, és a film megjelenési évét String típusú adatként kezeljük. A második módosítás az állományba való kiírásra vonatkozik, a filmekL.txt-be soronként írunk a mapM_ függvény segítségével:

```
import System.IO
import qualified Data.ByteString.Lazy.Char8 as L8

foBS1_ :: IO ()
foBS1_ = do
```

```
    putStr "ev: "
    fEv <- getLine
    temp <- L8.readFile "film.txt"
    let tLs = map (strProcBS_ fEv) $ L8.lines temp
    let fLs = filter auxFilter tLs
    outf <- openFile "filmekL.txt" WriteMode
    mapM_ (auxF outf) fLs
    hClose outf
        where
        auxF = \ outf (Just k)
                    -> hPutStrLn outf (L8.unpack k)

strProcBS_ :: String -> L8.ByteString ->
              Maybe L8.ByteString
strProcBS_ fEv iLs = res
    where
    ls = L8.split '\t' iLs
    kEv = L8.unpack $ ls !! 0
    kCim = ls !! 1
    res = if kEv == fEv then Just kCim else Nothing
```

A következő kódsor a fenti példának egy harmadik, egyben utolsó feldolgozási módszere lesz, ahol szintén `ByteString` típust alkalmazunk a szövegállományban levő adatok feldolgozására. A filmcímek kiválasztásához azonban előbb kiválasztjuk az állomány azon sorait, ahol a megjelenési év megegyezik a keresett évvel, majd a filmcímek képernyőre való kiíratását a `myPutStr` korábban már megadott függvénnyel végezzük. Az aktuális sor feldarabolását a korábban megadott `strProcBS` függvénnyel végezzük.

```
foBS2 :: IO ()
foBS2 = do
    putStr "ev: "
    tStr <- getLine
    let fEv = read tStr :: Int
    temp <- L8.readFile "film.txt"
    let fLs = filter (auxFilter . strProcBS fEv) $
              L8.lines temp
    mapM_ (myPutStr . L8.unpack) fLs
```

8.16. feladat Írjunk egy Haskell-programot, amely ábécésorrendbe rendezi az előző feladatban is használt `film.txt` állományban megjelenő rendezőket, majd kiírja az így kapott adatokat a `rendezoL.txt` állományba.

Emlékeztetőül a `film.txt` szövegállományban egy adott filmről `9` típusú adatot tárolunk, ahol egy filmre vonatkozó adatok egy sorban vannak, a különböző típusú adatok között tabulátor jel van, és `Unknown` jelenik meg, ha az adatra vonatkozóan nincs megadva érték.

A feladat főfüggvénye a `foBS3` lesz, amelyben az előző példáknál használt módszer szerint dolgozzuk fel az állomány sorait:

```
import Data.List (sort)
foBS3 :: IO ()
foBS3 = do
   temp <- L8.readFile "film.txt"
   let tLs = halmazL $ (map strProcBS3 . L8.lines) temp
   let rLs = sort tLs
   L8.writeFile "rendezoL.txt" $ L8.intercalate "\n" rLs

strProcBS3 :: L8.ByteString -> Maybe L8.ByteString
strProcBS3 iLs = res
   where
   ls = L8.split '\t' iLs
   kRendezo = ls !! 8
   res = if kRendezo /= L8.pack "Unknown"
           then Just kRendezo
           else Nothing

halmazL :: Eq a => [Maybe a] ->[a]
halmazL [] = []
halmazL (val: ve) = case val of
   Just k -> k : halmazL [x | x <- ve, Just k /= x]
   Nothing -> halmazL [x | x <- ve]
```

Az állomány egy adott sorát az `strProcBS3` függvényben dolgozzuk fel, mint `ByteString`. Először tördeljük a sort a tabulátorok mentén, majd kiválasztjuk a rendezőre vonatkozó értéket, ha ez az érték `Unknown`, akkor `Nothing` lesz az eredmény.

A `halmazL` függvényben szűrjük ki a többször előforduló rendezőket, mivel egy adott rendező több filmet is rendezhet, ezért több sorban is szerepelhet. Ugyanitt oldjuk meg azt a helyzetet is, amikor nincs rendező feltüntetve valamely filmnél. A `halmazL` függvénynek a kimenete nem egy `Maybe a` elemtípusú lista lesz, hanem egy tetszőleges lista, amely csak a rendezők neveit fogja tartalmazni. Az így kapott `ByteString`-ekből álló lista elemeit rendezzük, majd az `intercalate` függvény segítségével egy

`ByteString`-et hozunk létre úgy, hogy az elemek közé `\n`-t szúrunk, és ezt írjuk ki a `writeFile`-lal az állományba.

8.4. JSON formátumú adatok

JSON formátumú adatokat, állományokat először JavaScript-tel dolgoztak fel, de manapság a legtöbb programozási nyelv támogatja, így a Haskell is. Ilyen típusú állományok esetében az adatok emberek által is olvasható formátumban vannak rögzítve, ahol a szerkezet kötött, és több típusú adat tárolása biztosított, például a számok, a logikai értékek, a karakterláncok, az egydimenziós tömbök és az objektumok is fedolgozhatók.

A Haskell a `Data.Aeson`, illetve `Data.Aeson.Encode.Pretty` könyvtármodulok importálásával nyújt lehetőséget JSON formátumú adatok kezelésére. Ezek installálása, Windows alatt a `PowerShell`-en keresztül a következőképpen történik:

```
> cabal update
> cabal v1-install aeson
> cabal v1-install aeson-pretty
```

A `Data.Aeson` könyvtármodul definiálja a `Value` adatszerkezetet, amely, ahogy a következő definícióból is látszik, hat konstruktorral rendelkezik.

```
data Value =
 Object (HashMap Text Value) |
 Array (Vector Value) |
 String Text |
 Number Scientific |
 Bool Bool |
 Null
```

A következő kódsorban a `szemelyToObj` közvetlen módon hoz létre konstans értékek alapján egy `Value` típusú értéket. Az `objToSzemely` függvény pedig egy `Value` típusú adatot alakít vissza tuple típusú adattá. A `szemelyToObj`-nek két argumentuma van, az első `Text` típusú, ami az Unicode karakterek, a második `Scientific` típusú, ami a valós értékek hatékony kezelését teszi lehetővé. A meghívások során a `String` és `Text` típusok közötti átalakítást a `pack` függvénnyel végezzük. Két új könyvtármodul importja is megjelenik, a `Data.Text`-re a `Text` típushasználat miatt, a `Data.Scientific`-re pedig a `Scientific` típus miatt van szükség.

```
{-# LANGUAGE OverloadedStrings #-}
import qualified Data.HashMap.Strict as HM
import Data.Aeson
import Data.Text
import Data.Scientific ( Scientific )

szemelyToObj :: Text -> Scientific -> Value
szemelyToObj n s = Object $ HM.fromList [
   ("nev", String n),
   ("szEv", Number s)]

objToSzemely :: MonadFail m => Value
                -> m (Text, Scientific)
objToSzemely (Object obj) = do
   n <- case HM.lookup "nev" obj of
       Just (String x) -> return x
       Just _   -> fail "nem String"
       Nothing -> fail "nem nev mezo"

   s <- case HM.lookup "szEv" obj of
       Just (Number x) -> return x
       Just _ -> fail "nem Number"
       Nothing -> fail "nem szEv mezo"
   return (n, s)

> import Data.Text ( pack )
> import Data.Aeson ( encode )

> szemelyToObj (pack "Kiss Zsuzsa") 2000
Object (fromList [ ("szEv",Number 2000.0),
                   ("nev",String "Kiss Zsuzsa")])

> encode $ szemelyToObj (pack "Kiss Zsuzsa") 2000
"{\"szEv\":2000,\"nev\":\"Kiss Zsuzsa\"}"

> objToSzemely (szemelyToObj (pack "Kiss Zsuzsa") 2000)
("Kiss Zsuzsa",2000.0)
```

A `szemelyToObj` függvényben az `Object` konstruktor segítségével hoztuk létre a `Value` típusú adatot, amit jelen esetben két kulcsérték pár alapján építettünk fel. A Haskell `HashMap.Strict` könyvtármoduljában levő `fromList` függvénnyel, kulcs-érték párok típusú adatokból hoztunk létre

`HashMap` típusú adatokat. Lekérdezéskor a `Data.Aeson` könyvtármodulban található `encode`-dal jelenítettük meg a JSON formátumú adatot, mint `ByteString`. A `objToSzemely` a fordított műveletet végzi.

A `HashMap.Strict` könyvtármodulban további hasznos függvények implementációit találjuk, ahogy ezt a következő kódsorokban is láthatjuk, ahol figyeljük meg azt is, ahogyan beállítottuk a `OverloadedStrings` kiterjesztést.

```
> import qualified Data.HashMap.Strict as HM
> :set -XOverloadedStrings
> :set +m
> hm = HM.fromList [("nev", String "Kiss Zsuzsa"),
|                   ("szEv", Number 2000),
|                   ("cnp", Number 1234567)]

> HM.size hm   -- a lista mérete
3

> HM.keys hm   -- a hashmap-ben használt kulcsok
["szEv","cnp","nev"]

> HM.elems hm  -- a hashmap értékei
[Number 2000.0,Number 1234567.0,String "Kiss Zsuzsa"]

> HM.lookup "cnp" hm -- adott kulcs alapján való keresés
Just (Number 1234567.0)
```

A `Data.Aeson` könyvtármodulban definiált `ToJSON` és `FromJSON` típusosztályok segítségével `HashMap` típusok explicit használata nélkül is létrehozhatunk különböző típusú adatokból JSON formátumú adatokat, illetve JSON formátumú adatokat alakíthatunk át tetszőleges típusú adattá. A JSON adatok kódolásához, illetve dekódolásához két függvényt találunk itt, ahol az `encode`-ot fentebb is használtuk.

```
encode :: ToJSON a => a -> ByteString
decode :: FromJSON a => ByteString -> Maybe a
```

Az `encode` egy JSON formátumú adatot próbál `ByteString`-é átalakítani, míg a `decode` függvény, amennyiben az lehetséges, egy `ByteString` típusú adatot dekódol JSON formátumú adattá.

A két függvény használatát a következő lekérdezés is szemlélteti:

```
> import Data.Aeson ( decode, Object )
> set: +m
```

```
> decode (encode $ szemelyToObj (pack "Kiss Zsuzsa")
|       2000) :: Maybe Object
Just (fromList [   ("szEv",Number 2000.0),
                   ("nev",String "Kiss Zsuzsa")])
```

Aszerint, hogy milyen értékeket tárol egy JSON formátumú adat, szükséges megadni a megfelelő típusdeklarációkat, illetve a típusdeklarációk szerint példányosítani kell a `FromJSON`, illetve `ToJSON` típusosztályokat, amelyek keretén belül meg kell adni a `toJSON` és `parseJSON` függvényeket. Ezek után az `encode` és `decode` függvények segítségével el tudjuk végezni a megfelelő kódolást, illetve dekódolást.

A következő kódsor erre ad példát úgy, hogy egy `Szemely` típusú értéket alakít át JSON formátumba, majd elvégzi a visszaalakítást.

```
{-# LANGUAGE OverloadedStrings #-}
import Data.Aeson
import Data.Maybe

data Szemely = Szemely {
       nev :: String,
       szEv :: Int
   }
   deriving (Show)

instance ToJSON Szemely where
   toJSON (Szemely n s) = object
       [ "nev" .= n
       , "szEv" .= s
       ]

instance FromJSON Szemely where
   parseJSON = withObject "Szemely" $ \v -> do
       n <- v .: "nev"
       s <- v .: "szEv"
       return $ Szemely n s

foJSON :: IO ()
foJSON = do
   let lsJ = encode $ Szemely {
            nev = "Kiss Zsuzsa",
            szEv = 2000}
   putStrLn $ "JSON formaban: " ++ show lsJ
```

```
    let lsD = decode lsJ :: Maybe Szemely
    putStrLn $ "dekodolva: " ++ show (fromJust lsD)
```

```
> foJSON
JSON formaban: "{\"szEv\":\"2000\",\"nev\":Kiss Zsuzsa\}"
dekodolva: Szemely {nev = "Kiss Zsuzsa", szEv = 2000}
```

A `toJSON` függvényben az `object` függvény segítségével hoztuk létre a megfelelő JSON értéket, ahol a kulcs-érték párok definiálásához a `(.=)` operátort használtuk.

A `parseJSON` függvényben a `withObject` megvizsgálja, hogy a bemenet illeszkedik-e a megadott típusra, ha igen, akkor a `(.:)` operátorral létrehozza a megfelelő mezőértékeket, egyébként hibaüzenetet ad.

A `ToJSON` és `FromJSON` típusosztályok mellett, ha a JSON formátumú adatfeldolgozásnál a `Generic` típusosztályt is használjuk, akkor biztosítva lesz az adatok automatikus konverziója, szükségtelenné válik a `toJSON` és `parseJSON` függvények implementálása. Használatához a `GHC.Generics` importálása mellett szükséges engedélyezni a fordító `DriveGeneric` kiterjesztését. Ezeknek értelmében a kódsor a következőképpen módosul:

```
{-# LANGUAGE OverloadedStrings, DeriveGeneric #-}
import Data.Aeson
import Data.Maybe
import GHC.Generics

data Szemely = Szemely {
       nev :: String,
       szEv :: Int
   }
   deriving (Show, Generic)

instance ToJSON Szemely
instance FromJSON Szemely

foJSON1 = do
   let lsJ = encode $ Szemely {
       nev = "Kiss Zsuzsa",
       szEv = 2000}
   putStrLn $ "JSON formaban: " ++ show lsJ
   let lsD = decode lsJ :: Maybe Szemely
   putStrLn $ "dekodolva: " ++ show (fromJust lsD)
```

```
> foJSON1
JSON formaban: "{\"szEv\":\"2000\",\"nev\":Kiss Zsuzsa\}"
dekodolva: Szemely {nev = "Kiss Zsuzsa", szEv = 2000}
```

Amennyiben engedélyezzük a `DeriveAnyClass` kiterjesztést, akkor szükségtelenné válik az explicit példányosítás, csak a `Szemely` típusdeklarációjánál kell változtatnunk:

```
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE DeriveGeneric #-}
{-# LANGUAGE DeriveAnyClass #-}

import Data.Aeson
import GHC.Generics
import Data.Maybe

data Szemely = Szemely {
   nev :: String,
   szEv :: Int
   }
   deriving (Show, Generic, ToJSON, FromJSON)

foJSON2 = do
   let lsJ = encode $ Szemely {
       nev = "Kiss Zsuzsa",
       szEv = 2000}
   putStrLn $ "JSON formaban: " ++ show lsJ
   let lsD = decode lsJ :: Maybe Szemely
   putStrLn $ "dekodolva: " ++ show (fromJust lsD)

> foJSON2
JSON formaban: ...
```

8.17. feladat A `szemelyek.json` állomány JSON szerkezetű személyek adatait tárolja: név, születési év. Írjunk egy Haskell-programot, amely dekodólja az állományban levő adatokat, azaz átalakítja a JSON formátumú adatokat `Szemelyek` típusú adatokká, rendezi őket születési év alapján, majd a rendezett listát visszaalakítja JSON formátumba, és kiírja az eredményt a `szemelyekR.json` állományba.

Legyen a `szemelyek.json` állomány tartalma a következő:

```
{"szemelyek":[
   {"nev"          :"Nagy Zsuzsa"
```

```
    ,"szEv"          :2001
    },

    {"nev"           :"Kiss Tibor"
    ,"szEv"          :1999
    },

    {"nev"           :"Szasz Zsombor"
    ,"szEv"          :2002
    },

    {"nev"           :"Szep Attila"
    ,"szEv"          :1998
    }
]}
```

A feladat megoldása egyetlen `foSzemely` függvényből fog állni, amelyben a `readFile`-lal beolvassuk az adatokat, a `decode`-dal átalakítjuk a JSON formátumú adatokat `Maybe Szemelyek` típussá, rendezzük őket a `sortBy`-al a `szEv` mező alapján, az `encodePretty`-vel visszaalakítjuk, majd a `writeFile`-al kiírjuk a rendezett JSON formátumú adatokat az állományba.

```
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE DeriveGeneric #-}
{-# LANGUAGE DeriveAnyClass #-}
import qualified Data.ByteString.Lazy.Char8 as L8
import Data.Aeson
import Data.Aeson.Encode.Pretty
import GHC.Generics
import Data.List
import Data.Ord

data Szemely = Szemely {
       nev :: String,
       szEv :: Int
   }
   deriving (Show, Read, Generic, ToJSON, FromJSON)

data Szemelyek = Szemelyek {
       szemelyek :: [Szemely]
   }deriving (Show, Read, Generic, ToJSON, FromJSON)
```

```
foSzemely = do
   temp <- L8.readFile "szemelyek.json"
   let Just jTemp = decode temp :: Maybe Szemelyek
   let ls = szemelyek jTemp
   let sLs = sortBy (comparing szEv) ls
   let jLs = encodePretty (Szemelyek sLs)
   L8.writeFile "szemelyekR.json" jLs
```

Az `encode` helyett az `encodePretty` függvényt használtuk, azért, hogy a JSON formátumban az új sor jelek és tabulátorok segítségével tudjuk megoldani az adatok elegáns formázását.

Ahhoz, hogy az adatok konvertálása hibamentesen történjen, fontos, hogy megtartsuk a `Szemely` és `Szemelyek` típusdeklarációk esetében a `nev`, `szEv` és `szemelyek` megnevezéseket, azaz olyan mezőneveket válasszunk, amelyek a JSON formátumban levő állományban használatosak.

8.18. feladat A `tudosok.json` állomány JSON szerkezetű, tudósok adatait tárolja: vezetéknév, nemzetiség, születési év és elhalálozási év. Írjunk egy Haskell-programot, amely meghatározza az állományban szereplő tudósok születési év szerint rendezett sorrendjét, illetve egy másik opció választása esetén az életkor szerinti rendezett sorrendet írja ki. Abban az esetben, ha nem jelenik meg az elhalálozási év, az életkor helyett írja ki a `kortars` szót. Egy ilyen szerkezetű állomány letölthető a következő linkről:

https://ms.sapientia.ro/~mgyongyi/Funk_Log/Jegyzet/tudosok.json

Hasonlóan a korábbi kódsorokhoz, engedélyezzük a `DeriveAnyClass` kiterjesztést, azért, hogy ne kelljen explicit példányosítást végezni, így a következők lesznek az engedélyezett kiterjesztések:

```
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE DeriveGeneric #-}
{-# LANGUAGE DeriveAnyClass #-}
```

A szükséges importok a következők:

```
import qualified Data.ByteString.Lazy.Char8 as L8
import Data.Aeson
import GHC.Generics
import Data.List
import Data.Ord
```

Az állományban levő adatok kezeléséhez pedig két adatszerkezetet definiálunk:

```
data Tudos = Tudos {
    nev :: String,
    nemzetiseg :: String,
    szEv :: Int,
    hEv :: Maybe Int
    } deriving (Show, Generic, FromJSON)

data Tudosok = Tudosok {
    tudosok :: [Tudos]
    } deriving (Show, Generic, FromJSON)
```

A példát megoldó főfüggvény a `foTudos1`, ahol a rendezésekhez a `sortBy` beépített függvényt használjuk. Az `auxEv` a kiíratáshoz szükséges karakterláncot határozza meg. A `korF` függvényben, ha létezik elhalálozási év, akkor kiszámítjuk az életkor értékét, ellenkező esetben pedig `-1` lesz a függvény kimenetének értéke. A `myPrint` az adatok formázott kiíratását végzi.

```
foTudos1 = do
    inf <- L8.readFile "tudosok.json"
    let temp = decode inf :: Maybe Tudosok
    case temp of
        Nothing -> print "JSON adatfeldolgozasi hiba"
        Just tLs -> do
            putStrLn "(szEv/eKor)?"
            str <- getLine
            let ls = tudosok tLs
            case str of
                "szEv" -> do
                    putStrLn "rendezve:"
                    let sLs = sortBy (comparing szEv) ls
                    mapM_ (putStrLn . auxEv) sLs
                "eKor" -> do
                    putStrLn "rendezve: "
                    let sLs = sortBy (comparing korF) ls
                    mapM_ myPrint sLs
                _ -> putStrLn "hibas bemenet!"

auxEv :: Tudos -> String
auxEv ls = nev ls ++ ", szuletesi ev: "
                  ++ (show . szEv) ls

korF :: Tudos -> Int
```

```
korF x = case hEv x of
   Just xHEv -> xHEv - szEv x
   Nothing -> -1

myPrint :: Tudos -> IO()
myPrint ls =
   if kF == -1 then do
       let rLs = nev ls ++ " " ++ nemzetiseg ls
                        ++ " kortars"
       putStrLn rLs
   else do
       let temp = show kF
       let rLs = nev ls ++ " " ++ nemzetiseg ls ++ " "
                        ++ temp
       putStrLn rLs
   where
   kF = korF ls

> foTudos1
(szEv/eKor)?
eKor
rendezve:
Perelman orosz kortars
Lovasz magyar kortars
Galois francia 21
...
```

8.19. feladat Írjunk egy Haskell-programot, amely kiíratja a képernyőre adott nemzetiségű tudósok listáját, ahol az adatokat az előző feladatban megadott `tudosok.json` állományban találjuk. Használjuk a megadott `Tudos` és `Tudosok` adatszerkezeteket.

```
foTudos2 = do
    inf <- L8.readFile "tudosok.json"
    let temp = decode inf :: Maybe Tudosok
    case temp of
        Nothing -> print "JSON adatfeldolgozasi hiba"
        Just tLs -> do
            let ls = tudosok tLs
            let hLs = halmazL ls
            mapM_ (\k -> putStr $ k ++ " ") hLs
            putStr "\nnemzetiseg: "
            temp <- getLine
```

```
            putStrLn $ "\n" ++ temp ++ " tudosok: "
            mapM_ putStrLn $ valogat temp ls

halmazL :: [Tudos] ->[String]
halmazL [] = []
halmazL (k : ve) = nemzetiseg k : halmazL [x | x <- ve,
                   nemzetiseg k /= nemzetiseg x]

valogat :: String -> [Tudos] -> [String]
valogat nemz ls = [nev x ++ " " ++ show (szEv x) |
                   x <- ls, nemzetiseg x == nemz]
```

```
> foTudos2
svajc magyar orosz angol perzsa francia indiai nemet
norveg lengyel
nemzetiseg: perzsa

perzsa tudosok:
Hvarizmi 780
```

A `foTudos2` függvény meghívása feltételezi, hogy a kódsor engedélyezi a korábbi példánál is használt kiterjesztéseket, illetve tartalmazza azokat az importokat, amelyeket az előző példánál is használtunk.

A `foTudos2` az `ls` listába megadja a `tudosok.json` állományban található tudósok listáját, ahol a `halmazL` függvény az `ls` lista alapján a `hLs` listába meghatározza az állományban szereplő különböző nemzetiségeket. A `foTudos2` a képernyőre ki is írja a `hLs` lista tartalmát, hogy látható legyen, hogy milyen nemzetiségek között válogathatunk. A megfelelő nemzetiségű tudósok kiválogatását a `valogat` függvény végzi, amely a válogatás mellett létrehoz egy `String` típusú értékekből álló listát, ahol a `String` típusú értéket a megfelelő nemzetiségű tudós `nev` és `szEv` mezőjéből építjük fel.

8.20. feladat Írjunk egy Haskell-programot, amely létrehozza megadott nemzetiségű tudósok listáját, illetve a létrehozott lista alapján JSON formátumban kiírja az adatokat egy állományba, ahol az adatokat az előző feladatban megadott `tudosok.json` állományban találjuk.

A `foTudos3` meghívásához ugyanazokat a kitejesztéseket kell engedélyezni, mint az előző példánál az importok listájában, azonban szerepelnie kell az `Encode.Pretty`-nek is ahhoz, hogy az állományba automatikusan, JSON formátumba tudjunk kiíratni. A kódsorban csak ez utóbbi van feltüntetve. Az előző példához képest a `Tudos` és `Tudosok` típusdeklarációkat

is módosítani kellett, mert a származtatásnál fel kellett tüntetni a `ToJSON` típusosztályt.

```
...
import Data.Aeson.Encode.Pretty

data Tudos = Tudos {
    nev :: String,
    nemzetiseg :: String,
    szEv :: Int,
    hEv :: Maybe Int
    } deriving (Show, Generic, FromJSON, ToJSON)

data Tudosok = Tudosok {
    tudosok :: [Tudos]
    } deriving (Show, Generic, FromJSON, ToJSON)


foTudos3 = do
    inf <- L8.readFile "tudosok.json"
    let temp = decode inf :: Maybe Tudosok
    case temp of
        Nothing -> print "JSON adatfeldolgozasi hiba"
        Just tLs -> do
            let ls = tudosok tLs
            let nLs = halmazL ls
            mapM_ (\k -> putStr $ k ++ " ") nLs
            putStr "\nnemzetiseg: "
            temp <- getLine
            let nLs = valogatF temp ls
            let jLs = encodePretty (Tudosok nLs)
            L8.writeFile "tudosokN.json" jLs

valogatF :: String -> [Tudos] -> [Tudos]
valogatF nemz = filter (\x -> nemzetiseg x == nemz)
```

Vegyük észre, hogy a `foTudos3` az előző példánál megadott `halmazL` függvényt használja, ezért ennek kódsorát onnan kell átvenni. A `valogatF` függvény pedig hasonló elgondoláson alapszik, mint az előző példánál megadott `valogat` függvény, csak a listaelemek most `Tudos` típusúak lesznek.

8.5. Kitűzött feladatok

8.1. feladat Írjunk Haskell-programot, amely kiírja lexikografikus sorrendben egy szövegállományba az ábécé első `n` betűjéből képezhető `m` hosszúságú szavakat.

8.2. feladat Írjunk Haskell-programot, amely kiírja egy szövegállományba az ábécé első `n` betűjéből képezhető `m` hosszúságú szavakat, ahol nem megengedett a betűk többszöri felhasználása, és nem számít a betűk sorrendje. Formázzuk úgy a szövegállomány tartalmát, hogy minden sorba négy megoldást írjunk, a megoldások közé *szóköz*#*szóköz*-t, a megoldáselemek közé pedig *szóköz*öket tegyünk.

8.3. feladat Írjunk Haskell-programot, amely meghatározza az `n` királynő probléma `k`-adik megoldását.

8.4. feladat Írjunk Haskell-programot, amely kiírja egy állományba `1`-től `3999`-ig az összes arab számot és a számoknak megfelelő római számot.

8.5. feladat Írjunk Haskell-programot, amely meghatározza a billentyűzetről beolvasott két római szám összegét, szorzatát, különbségét, osztási hányadosát, és az eredményeket római számokként adja meg.

8.6. feladat Egy állományban személynevek és a hozzájuk tartozó születési időpontok vannak tárolva. Írjunk Haskell-programot, amely meghatározza a születési időpontok római számokbeli alakját, és az eredményt a megfelelő nevekkel együtt átírja egy másik állományba.

8.7. feladat Egy szövegállományban egy adott betegről a következő adatok vannak eltárolva: név, születési év, vérnyomás értékpárok, azaz adott a következő adatszerkezet:

```
data Beteg = Beteg {
    bNev :: [Char],
    bVerny :: [(Int, Int)],
    bSzEv :: Int
    } deriving (Show)
```

Írjunk egy Haskell-programot, amely `ByteString`-eket használva dolgozza fel az állományban levő adatokat, létrehoz egy `Beteg` elemtípusú listát és

– rendezi a lista elemeit a `nev` mező alapján ábécésorrendbe,

– bináris keresést alkalmazva meghatározza egy adott személy vérnyomásértékeit, és megállapítja, hogy hányszor volt magas a vérnyomása, ahol egy személynek magas a vérnyomása, ha az első érték `> 160` vagy a második érték `> 140`.

8.8. feladat Egy szövegállományban egy adott filmről a következő adatok vannak eltárolva: filmcím, rendező, megjelenési év, költség, azaz adott a következő adatszerkezet:

```
data Film = Film {
    fCim :: [Char],
    fRendezo :: [Char]
    fEv :: Int,
    fKoltseg :: Int
    } deriving (Show)
```

Írjunk egy Haskell-programot, amely `ByteString`-eket használva dolgozza fel az állományban levő adatokat, létrehoz egy `Film` elemtípusú listát és

– rendezi az adatokat megjelenési év szerint növekvő sorrendbe,
– bináris keresést alkalmazva megállapítja, hogy egy adott évben milyen filmek jelentek meg,
– meghatározza, hogy melyik filmnek volt a legnagyobb a költsége.

8.9. feladat Egy szövegállományban egy adott személyről a következő adatok vannak eltárolva: vezetéknév, keresztnév, azaz adott a következő adatszerkezet:

```
data Szemely = Szemely {
    szVnev :: [Char],
    szKnev :: [Char],
    } deriving (Show)
```

Írjunk egy Haskell-programot, amely `ByteString`-eket használva dolgozza fel az állományban levő adatokat, létrehoz egy `Szemely` elemtípusú listát, majd meghatározza mindegyik személyről, hogy mikor van a névnapja. A névnapok megállapításához használhatjuk a `nevnapok.txt` állományt, amely letölthető a következő linkről:

https://ms.sapientia.ro/~mgyongyi/Funk_Log/Jegyzet/nevnapok.txt

8.10. feladat Írjunk egy Haskell-programot, amely formázza egy adott szövegállomány tartalmát. A formázás alatt azt kell érteni, hogy a következő írásjelek után, ha több van belőlük, szigorúan egyet hagyjunk, illetve egyet

tegyünk, ha nincs egy sem: pont, vessző, felkiáltójel, kérdőjel, kettőspont, pontosvessző.

8.11. feladat Az `iban.txt` állomány IBAN-kódokat tartalmaz, ahol minden IBAN-kód `String` értékkel van megadva. Írjunk egy Haskell-programot, amely beolvassa az állomány tartalmát, rendezi az adatokat, majd bináris keresést használva ellenőrzi, hogy egy megadott IBAN-kód szerepel-e az állományban. Az adatok feldolgozását először lista, majd tömb adatszerkezetet használva végezzük. Használhatjuk az `iban.txt`, illetve `ibanLength.txt` állományokat, amelyek letölthetőek a következő linkekről:

https://ms.sapientia.ro/~mgyongyi/Funk_Log/Jegyzet/iban.txt

https://ms.sapientia.ro/~mgyongyi/Funk_Log/Jegyzet/ibanLength.txt

8.12. feladat Az `iban.txt` állomány IBAN-kódokat tartalmaz, ahol minden IBAN kód egy `String` értékkel van megadva. Írjunk egy Haskell-programot, amely beolvassa az állomány tartalmát, majd megvizsgálja, hogy melyek a helyes IBAN-kódok. Egy IBAN-kódot akkor tekintünk helyesnek, ha csak számjegyeket és angol ábécébeli nagybetűket tartalmaz, illetve fennáll a következő:

- hossza megegyezik az országhoz tartozó hosszal, ahol az országhoz tartozó hosszérték az `ibanLength.txt` állományban található és
- a csoportosítás és helyettesítés után kapott egész szám `97`-tel való osztási maradéka 1, ahol
  - csoportosítás alatt azt értjük, hogy az IBAN-kód első négy karakterét töröljük a kód elejéről, és a kód végéhez fűzzük,
  - helyettesítés alatt azt értjük, hogy minden alfanumerikus karaktert helyettesítünk a következő számértékkel: `A -> 10, B -> 11,..., Z -> 35`, és a kapott karakterláncot egész számnak tekintjük.

Például legyen az IBAN-kód: **`GB82`**`WEST12345698765432`,

- az IBAN-kód hossza `22`, ami megegyezik az előírt hosszértékkel,
- az átcsoportosítás után kapjuk

  `WEST12345698765432`**`GB82`**,

- a helyettesítés után kapjuk:

  `3214282912345698765432161``82`,

- ellenőrizzük, hogy a kapott szám `97`-tel való osztási maradéka `1`.

8.13. feladat Az `autok.json` állomány JSON szerkezetű, személygépkocsik adatait tárolja: gyártmány (`String`), modell (`String`), évjárat (`Int`). Írjunk egy Haskell-programot, amely

- létrehoz egy `gyartmany.json` JSON formátumú állományt, amelybe átírja megadott gyártmányú személygépkocsik adatait, pontosabban a modell és évjárat értékeket, ahol a keresett gyártmány értékét a billentyűzetről olvassuk be,
- létrehoz egy `evjarat.json` JSON formátumú állományt, amelybe átírja megadott évjáratú személygépkocsik adatait, pontosabban a gyártmány és modell értékeket, ahol a keresett évjárat értékét a billentyűzetről olvassuk be,
- kiírja a képernyőre a személygépkocsik adatait, az évjárat szerinti mező alapján rendezve, minden sorba egy gyártmány, modell, illetve évjárat értéket írva.

8.14. feladat A `betegek.json` állomány JSON szerkezetű, betegek adatait tárolja: név (`String`), ország (`String`), születési év (`Int`), betegségek (`[String]`). Írjunk egy Haskell-programot, amely

- létrehoz egy `orszag.json` JSON formátumú állományt, amelybe átírja megadott országú betegek adatait, pontosabban a nevet, születési évet és a betegségek értékeit, ahol a keresett ország értékét a billentyűzetről olvassuk be,
- kiírja egy adott országon belül a betegségeket és a betegségek számát, ahol az országnevet a billentyűzetről olvassuk be,
- meghatározza, hogy melyik országban van a legtöbb fajta betegség.

IRODALOMJEGYZÉK

[1] Steve Awodey, *Category Theory*. Oxford University Press; 2nd edition, 2010

[2] Richard Bird, *Thinking Functionally with Haskell*. Cambridge University Press, 2015.

[3] Alonzo Church, *Introduction to Mathematical Logic*. Princeton University Press; Reprint edition, 1996.

[4] Hal Daume III, *Yet Another Haskell Tutorial*. University of Maryland, 2002–2006.

[5] Diviánszky Péter, *https://people.inf.elte.hu/divip/*.

[6] ELTE Nyelv-leírások, *http://nyelvek.inf.elte.hu/leirasok/Haskell/*.

[7] Mihai Gontineac, *Programare Funcțională – O introducere utilizând limbajul Haskell*. Ed. Al. Myller, Iaşi, 2006.

[8] Graham Hutton, *Programming in Haskell*. Cambridge University Press, 2007.

[9] Király Roland, *Funkcionális nyelvek*. EKF Eger, 2011.

[10] Miran Lipovača, *Learn You a Haskell for Great Good!* No Starch Press, 2011.

[11] Alejandro Serrano Mena, *Beginning Haskell: A Project-Based Approach*. Apress, 2014.

[12] Nyékyné Gaizler Judit, *Programozási nyelvek*. Kiskapu, Budapest, 2003.

[13] Bryan O'Sullivan, John Goerzen and Don Stewart, *Real World Haskell*. O'REILLY, 2008.

[14] Simon Thompson, *Haskell: The Craft of Functional Programming*. Addison-Wesley Professional, 3 ed., 2011.

ABSTRACT

Functional programming, Haskell basics

Functional programming, Haskell basics is an introductory book in one of the most interesting and elegant programming paradigm. The author focuses on the fundamentals of functional programming using Haskell as main language for programming problems. The book is based on the courses that the author has taught at Sapientia Hungarian University of Transylvania since 2007.

The first chapter compares two programming paradigms, namely the imperative and the functional programming, detailing the differences and similarities.

The second chapter provides a general introduction to the principles of Haskell, such as types, type classes, data structures (lists and tuples), definitions, modules, etc.

The third chapter presents in detail the programming features of Haskell, where guards, the layout rule, recursion, pattern matching, different types of expression, function composition, higher-order functions, and the evaluation strategy are highlighted.

The fourth chapter deals with lists, the most important and most frequently used data structure in functional programming. This chapter also presents some sorting algorithms and provides the theory needed to understand the higher-order fold functions.

The fifth chapter introduces the basics of input and output operations, file management, and it presents implementations related to exceptions. It also describes the concept of monads, which is the basic structure for managing side effects in Haskell.

The sixth chapter focuses on types and data structures, such as record types, algebraic data types, parameterized types, and recursive types and presents implementations related to binary trees.

The last chapter presents some implementations of combinatorial problems. The implementation of the binary search algorithm, problems solved with ByteStrings, and the processing of JSON data are also included in this chapter.

The topics of the book are presented in an appropriate didactic and academic style, where at the end of each chapter there are proposed problems related to the presented material.

REZUMAT

Programarea funcțională, elementele de bază Haskell

Programarea funcțională, elementele de bază Haskell este o carte introductivă într-una dintre cele mai interesante și elegante paradigme de programare. Autorul se concentrează pe fundamentele programării funcționale folosind Haskell ca limbaj principal pentru problemele de programare. Cartea se bazează pe cursurile pe care autorul le-a predat la Universitatea Sapientia, România din 2007.

Primul capitol compară două paradigme de programare, și anume programarea imperativă și funcțională, detaliind diferențele și asemănările.

Al doilea capitol oferă o introducere generală a principiilor Haskell, cum ar fi tipuri, clase de tip, structuri de date, definiții, module, etc.

Al treilea capitol prezintă detaliat caracteristicile de programare ale lui Haskell, unde sunt evidențiate gărzile, regula de aliniere, recursivitatea, potrivirea după șabloane, diferite tipuri de expresie, noțiunea de compoziție a funcției, funcții de ordin înalt, strategia de evaluare.

Al patrulea capitol tratează listele, cea mai importantă și cea mai frecvent utilizată structură de date în programarea funcțională. Acest capitol prezintă, de asemenea, câțiva algoritmi de sortare și oferă teoria necesară pentru a înțelege funcțiile de tip fold.

Al cincilea capitol introduce elementele de bază ale operațiilor de intrare și ieșire, gestionarea fișierelor și prezintă implementări legate de excepții. De asemenea, descrie conceptul de monade, care este structura de bază de gestionare a efectelor secundare din Haskell.

Al șaselea capitol acordă atenție tipurilor și structurilor de date, cum ar fi tipurile de înregistrări, tipurile de date algebrice, tipurile parametrizate și tipurile recursive. Folosind tipuri recursive, sunt prezentate implementări legate de arbori binari.

Datorită ușurinței de implementare a problemelor de combinatorică într-un limbaj de programare funcțional, ultimul capitol prezintă probleme legate de combinatorică. Implementarea algoritmului de căutare binară, probleme rezolvate cu `ByteStrings` și procesarea datelor de tip JSON sunt incluse, de asemenea, în acest capitol.

Temele cărții sunt prezentate într-un stil didactic și academic adecvat, iar la sfârșitul fiecărui capitol se află probleme propuse legate de materialul prezentat.

A SZERZŐRŐL

Márton Gyöngyvér egyetemi tanulmányait a kolozsvári Babeș–Bolyai Tudományegyetem Informatika Karán végezte, doktori tudományos fokozatát pedig 2014-ben szerezte Magyarországon, a Debreceni Egyetemen, a műszaki tudományok területén, informatika tudományokban.

1995–2003 között informatikatanár a marosvásárhelyi Bolyai Farkas Elméleti Líceumban.

A Sapientia Erdélyi Magyar Tudományegyetem megalakulásának pillanatától kezdve, 2001-től társult oktatóként tanít a Műszaki és Humántudományok Kar Matematika–Informatika Tanszékén, ahol 2003-tól főállású tanársegéd, majd 2014-től főállású adjunktus.

Egyetemi oktatása kezdetekor több labortevékenységet vezetett, úgymint programozás és programozási nyelvek, számelmélet, algoritmusok és adatszerkezetek, logikai programozás. Jelenleg három előadást és a hozzá tartozó labortevékenységeket vezeti, úgymint diszkrét matematika, funkcionális programozás, információbiztonság és kriptográfia.

Doktori munkája során azt vizsgálta, hogyan lehet létrehozni olyan titkosítási rendszereket, és melyek azok, amelyek biztonságosak a választott rejtjelezett szöveg alapú támadással (chosen-ciphertext attack) szemben.

TÁRGYMUTATÓ

Scientia Kiadó
400112 Kolozsvár (Cluj-Napoca)
Mátyás király (Matei Corvin) u. 4. sz.
Tel./fax: +40-364-401454
E-mail: scientia@kpi.sapientia.ro
www.scientiakiado.ro

Korrektúra:
Szenkovics Enikő

Műszaki szerkesztés:
Márton Gyöngyvér

Tipográfia:
Könczey Elemér

Nyomdai munkálatok:
F&F INTERNATIONAL Kft.
Felelős vezető: Ambrus Enikő igazgató

sapientia
tankönyvek

ISBN 978-606-975-050-6