Funkcionális programozás

Funkcionális programozás

Dr. Horpácsi Dániel

Bereczky Péter

Eötvös Loránd Tudományegyetem
Informatikai Kar
Programozási Nyelvek és Fordítóprogramok Tanszék

Lektorálta **Dr. Bozó István**

Budapest, 2025.

A jegyzet legfrissebb verziója elérhető a https://daniel-h.web.elte.hu/haskell címen.

Mi az a funkcionális programozás?

- Matematikai kifejezések
- Alaptípusok: számok, karakterek, logikai értékek

Függvénykifejezések

- A lambda-kalkulus
- Lambda kifejezések Haskellben
- Curryzés, részleges alkalmazás
- Függvénykompozíció
- Iteráció és rekurzió
- Polimorfizmus
- A lambda-kalkuluson túl

Listakifejezések

- Felsorolással megadott listák
- Lustaság, végtelen listák
- Mintaillesztés és iteráció listákon
- Intervallumkifejezések
- Halmazkifejezések
- A fold függvények

Algebrai adattípusok

- Az algebrai adattípus fogalma
- Mintaillesztés algebrai adatokon
- Nevezetes típusok
- Nevezetes izomorfizmusok

Típusosztályok

- Konkrét típusok nevezetes osztályai
- Példányosítás
- Functor
- Applicative
- Foldable

Ajánlott irodalom

Feladatgyűjtemény

- Bevezetés
- 1. gyakorlat: Ismerkedés a Haskell értelmezővel
- 2. gyakorlat: Műveletek számokkal
- 3. gyakorlat: Mintaillesztés
- 4-5. gyakorlat: Lista literálok, intervallumkifejezések, halmazkifejezések
- 6. gyakorlat - Rekurzió
- 7. gyakorlat - Listák bevezető, rekurzió

8. gyakorlat - Listák, map, filter
9. gyakorlat - Listák, fold, egyéb magasabbrendű függvények
10. gyakorlat - Egyszerű ADT-k
11. gyakorlat - Parametrikus ADT-k
12. gyakorlat - Rekurzív ADT-k

Előszó

Ez a jegyzet az ELTE Informatikai Karán oktatott Funkcionális programozás (BSc) esti tárgyhoz készült. A jegyzet nagy részben lefedi azokat az elméleti és gyakorlati kérdéseket, amelyeket az előadásokon és gyakorlatokon vizsgálunk a hallgatókkal. A jegyzet első fele az előadás kurzus teljesítéséhez ad segítséget, a második felében pedig programozási feladatokat adunk minden egyes témakörhöz, amelyekkel a programozási készség fejleszthető.

Az elmúlt évek során organikusan fejlődött ki a lefedett témák köre és a tárgyalás sorrendje. Fontos megjegyezni, hogy a megközelítés, a tárgyalás módja, a témák sorrendje nem a tipikus "Bevezető a Haskell programozásba" logikát követi: ahelyett, hogy a Haskell nyelvet tanítanánk meg módszeresen, inkább a funkcionális programozás fő fogalmait vesszük sorra, megmutatjuk a megvalósításukat, miközben lépésről lépésre építünk fel egy funkcionális programozási nyelvet a lambda-kalkulusból kiindulva.

> **Megjegyzés**
>
> **Előzetes követelmények**
>
> A leadott anyag elsajátításához elegendőek alapvető matematikai és programozási ismeretek. A funkcionális programozási paradigmába a matematika és az imperatív programozás felől vezetjük be a hallgatót.

Jelen jegyzet a matematikai kifejezések és a lambda-kalkulus felől közelíti meg a funkcionális programozást, nem célunk a teljes Haskell nyelvet bemutatni, inkább az alapfogalmak demonstrálására használjuk fel. A jegyzet a következő részekre tagolódik:

1. **Matematikai kifejezések**. Bevezetjük a logikai és szám alaptípusokat és felírunk velük egyszerű kifejezéseket. Megmutatjuk a kifejezések és a típusok kapcsolatát, és hogy a funkcionális program alapvetése egy kifejezés kiértékelése, normálformára hozása. A funkcionális program esszenciája épp az lesz, hogy melyik részkifejezést hogyan lehet átírni, egyszerűsíteni. A Haskellben tiszta, matematikai kifejezéseket írunk le. Az itt tárgyalt fogalmakat eleminek tekintjük, nem vezetjük őket vissza primitívebb konstrukciókra.
2. **Függvénykifejezések.** Bevezetjük a lambda-kalkulust, mint a funkcionális programozás alapvető számítási modelljét, és megmutatjuk, hogyan lehet függvényeket írni benne, majd erre építve olyan műveleteket megadni, amelyekkel "programokká" formáljuk az

egyszerű függvényeinket (például kompozíció, rekurzió, iteráció, esetszétválasztás). Megmutatjuk, hogy ezek hogyan jelennek meg a Haskell nyelvben (standard függvényként vagy nyelvi elemként) és hogyan valósítjuk meg őket egyszerűbb függvényekkel.

3. **Listakifejezések.** Bevezetjük a paradigma talán legjellegzetesebb összetett adattípusát, a listát, és megmutatjuk, miként tudunk érdekes számításokat leírni végtelen listák és lusta kiértékelés segítségével. Magát a lista típust eleminek tekintjük, de számos műveletet definiálunk rá elemibb eszközökkel. Megmutatunk két fontos konstrukciót, az intervallumkifejezést és a halmazkifejezést, és megmutatjuk, hogyan lehet őket kifejezni az eddigi eszközkészletünkkel. Végül áttekintjük a listák hajtogatásának (redukciójának) módszerét és alkalmazásait.
4. **Összetett típusok.** Megmutatjuk, hogy a listák mintájára számos egyéb típust tudunk definiálni. Bevezetjük az algebrai adattípus fogalmát az összeg és szorzat konstrukciókra építve, és megmutatjuk, hogyan lehet ilyen adatokat megkonstruálni, illetve mintaillesztéssel hozzáférni a bennük tárolt információhoz. Nevezetes példákat tekintünk, melynek során egyre összetettebb szerkezetű adattípusokat adunk meg. Végül a nevezetes típuskonstrukciók között fennálló izomorfizmusokat adunk meg, elmélyítve az összetett típusok működésének megértését.
5. **Típusosztályok.** Az összetett típusok tárgyalását követően bevezetjük a típusosztályok fogalmát és tárgyaljuk a jelentőségét a kód újrafelhasználhatóságának kérdésében. Végigvesszük a nevezetes típusosztályokat, egészen a tetszőlegesen bejárható és hajtogatható konténerekig. Megmutatjuk, milyen tipikus alkalmazások köthetőek a standard típusosztályokhoz, ezzel erősítve a gyakorlati vonatkozást.
6. **Feladatgyűjtemény.** A jegyzet második fele a gyakorlaton használt feladatokat mutatja be. A célunk nem feltétlenül a komplex feladatok tárgyalása és nagyméretű projektek készítése, hanem inkább egyszerűnek tekinthető, gyakorlatias feladatok minél többféle módon történő megoldása, hangsúlyozva a különböző nyelvi absztrakciók és elemek használatának eseteit.

Mi az a funkcionális programozás?

A programozási paradigma határozza meg, hogy egy programozási feladatot hogyan bontunk részfeladatokra és milyen alapvető eszközökkel írjuk le az egyes részfeladatok megoldását és azok kompozícióját. A modern programozási nyelvek mind igyekeznek gazdag nyelvi eszközkészlettel támogatni a programozó produktivitását, elősegítve a megoldóprogram minél magasabb szintű, olvasható, karbantartható leírását. Jó programozóként azonban tisztában kell lennünk azzal, hogy a sokszintű absztrakciók mögött mindig megbúvik egy fundamentális számítási modell, amelyre teljes egészében visszavezethetőek a magas szintű programozási konstrukciók. Ahogyan az imperatív programozási paradigmát meghatározza a Turing gép, úgy határozza meg a funkcionális programozást a lambda-kalkulus. A funkcionális programozás tárgyalása során az egyik célunk épp az lesz, hogy megmutassuk, hogyan és milyen programozási absztrakciók és eszközök származnak ebből a mögöttes számítási modellből.

Összevetve az imperatív programozással

Amikor az imperatív paradigmát bevezetik, általában az utasítás és az algoritmus fogalma szokott előkerülni, és annak ténye, hogy az imperatív program meghatározza, hogyan kell megoldani a programozási feladatot. Ezzel szemben a funkcionális paradigmában a meghatározó fogalmak a kifejezés és a kiértékelés, és a program elsősorban azt határozza meg, hogy mit kell kiszámítani, és a mögöttes stratégia fogja eldönteni, miként végezzük el a számítást.

Tekintsünk például egy egyszerű programozási feladatot, ahol összegezzük egy tömb/lista elemeit. Két alapvetően különböző megközelítés létezik a megoldásra:

- megadjuk, hogy *hogyan* kell kiszámolni az összeget;
- megadjuk, hogy *mit* értünk összeg alatt.

Az első megközelítést **imperatívnak** nevezzük: ez egy algoritmussal leírja, milyen lépéseket kell elvégezni az összeg kiszámításához. Az imperatív megadás pontosan definiálja a vezérlésfolyamot: "inicializáld a tömböt, inicializáld az összegváltozót, aztán ciklusban minden indexre frissítsd az összeg értékét a memóriában". C-ben például a következő kódhoz hasonló értékadásokat, szekvenciákat és ciklust használnánk:

```
int a[5] = {10, 20, 30, 40, 50};
int sum = 0;
for (int i = 0; i < sizeof(a) / sizeof(int); i++) {
  sum = sum + a[i];
}
```

Ezzel szembeállítva a **funkcionális (deklaratív)** megközelítés nem adja meg a lépéseket, csak azt definiálja, hogy mit értünk összeg alatt és hogy melyik összeget akarjuk kiszámítani. A mögöttes számítási modell és kiértékelési stratégia fogja maghatározni, hogy pontosan milyen lépések során jutunk el a bemenő adatoktól az összegig:

```
let sum [] = 0
    sum (x:xs) = x + sum xs
in sum [10, 20, 30, 40, 50]
```

A következő alapvető különbségeket tudjuk azonosítani a két megközelítés között:

imperatív program	**funkcionális program**
utasítások sorozata	egy **kifejezés** (a függvény is csak egy részkifejezés)
a **vezérlésfolyam** explicit, pontosan leírja, hogy milyen sorrendben kell a részszámításokat elvégezni	a vezérlés **implicit**, a kifejezés részkifejezéseit sokszor tetszőleges sorrendben ki lehet értékelni
az összeget **iterációval** (ciklussal) számítja ki	az összeget egy **rekurzív függvénnyel** adja meg
a tömbelemekhez indexeléssel fér hozzá	a listaelemekhez mintaillesztéssel fér hozzá
az utasításoknak **mellékhatása** van: módosítják a memória állapotát	a kifejezések **tiszták**, a függvények tiszta matematikai függvények (nincsenek mellékhatásaik)

A paradigma alapelemei és alapfogalmai

A jegyzetben részletesen tárgyaljuk majd a funkcionális paradigma sajátosságait, de itt a jegyzet elején szeretnénk felhívni a figyelmet és egyben felkelteni az olvasó érdeklődését egy gyors felsoroló áttekintéssel:

- **függvény mint érték (first-class function)**: ahogyan bármely más típusú adat, függvények is lehetnek paraméterek, visszatérési értékek, változókhoz rendelt értékek
- **curryzés, részleges függvényalkalmazás (partial application)**: a látszólag többparaméteres függvények is valójában egy paramétert várnak, viszont az eredményük egy újabb függvény, és ennek köszönhetően a többparaméteres függvényeket lehet részben alkalmazni (az elvártnál kevesebb argumentumot adni nekik), amelynek eredménye szintén függvény lesz
- **magasabb rendű függvények (higher-order function)**: a függvénnyel paraméterezett függvényeket magasabb rendűnek nevezzük, és ilyeneket használunk általánosított algoritmusok/megoldások leírására (ez gyakori programozási minta a Haskell programozásban) — ám általánosabb értelemben akkor is magasabb rendűnek tekintünk egy függvényt, ha függvény típusú az értéke
- **rekurzió és végrekurzió (recursion and tail-recursion)**: ismétlődő számításokat nem ciklusokkal, hanem rekurzív függvényekkel írunk le, és amikor lehetséges, a rekurzív alkalmazás a függvénytörzs fő művelete
- **egyszeri kötés (single assignment)**: a neveket egyszer lehet kötni és nem lehet felülírni, újradefiniálni, tehát a program minden pontján ugyanazt az értéket kell jelöljék
- **algebrai adattípusok és mintaillesztés (ADT, pattern matching)**: összetett típusokat más típusok szorzataiként és összegeként adunk meg, az adattagokhoz való hozzáférés pedig mintaillesztéssel történik, amiben a konstruktorok segítségével tudjuk felbontani az összetett adatokat
- **listák, halmazkifejezések (list comprehension)**: a listák alapvető összetett típusok, amelyeknek megadásához több nyelvi elem is rendelkezésre áll, például a matematikai halmazkifejezésekhez hasonló ún. *list comprehension*

- **polimorfizmus (polimoprhism)**: előnyben részesítjük a típusváltozók használatát, amelyekkel tetszőleges típusra tudunk függvényeket és típuskonstruktorokat alkalmazni, elősegítve a kód újrafelhasználhatóságát

A Haskell nemcsak funkcionális, de *tisztán funkcionális*, amire a következők is jellemzőek:

- **tisztaság (purity)**: a függvények tiszta (matematikai) számításokat írnak le (nincs mellékhatás, ugyanazon argumentumokra alkalmazva egy függvényt mindig ugyanazt az eredményt kapjuk), tehát a függvények nem függnek semmilyen globális állapottól és nem végeznek globális módosításokat
- **hivatkozási helyfüggetlenség (referential transparency)**: a tisztaság és az egyszeri kötés miatt egy kifejezés bárhol is szerepeljen egy programban, mindig ugyanarra az értékre fog kiértékelődni, amely elősegíti a programokról való kompozicionális érvelést (a részek jelentéséből mindig meghatározható az összetett kifejezések jelentése és tulajdonságai)
- **statikus típusozás, erős típusrendszer**: a függvények és kifejezések típusa fordítási időben kerül meghatározásra és ellenőrzésre, valamint a típusok között implicit konverziók nem történnek
- **lusta kiértékelés, végtelen adatszerkezetek**: a kifejezések kiértékelése lustán történik, csak azon részkifejezéseket értékeljük ki, amelyekre szükség van a kifejezés értékének meghatározásához (ennek köszönhetően végtelen adatokkal is lehet számításokat leírni, ha csak véges részükre van szükség)

A funkcionális paradigma tárgyalásához a példanyelvünk a Haskell lesz.

> **Megjegyzés**
>
> **Miért épp a Haskell a tananyag példanyelve?**
>
> A Haskell kutatási nyelvként jött létre, alapvetően matematikusok alkották meg a 80-as évek végén, de manapság a szoftveripar számos területén alkalmazzák. A funkcionális paradigma népszerűségének növekedésével a Haskell szerepe felértékelődött, az ipari szoftverfejlesztésben kifejezetten előnyös a statikus, erős típusozás és a tiszta számítások, hiszen könnyebb kompozicionálisan érvelni a

programok helyes működéséről, nehezebb nehezen felderíthető hibákat vinni a programokba.

A következő oldalakon lehet tájékozódni, hogy hogyan alakult a Haskell nyelv népszerűsége az elmúlt években, és hogy mely nagyobb vállalatok, bankok, szervezetek alkalmazzák a Haskell nyelvet ipari méretű és bonyolultságú szoftverek fejlesztésére:

- http://wiki.haskell.org/Haskell_in_industry
- https://github.com/erkmos/haskell-companies
- https://www.tiobe.com/tiobe-index/

Fontos

A jegyzet alapvetően a Haskell nyelv 2010-es szabványát tárgyalja és használja a példák felírásakor. A nyelvi elemek bemutatásakor a szintaxis és a szemantika is informálisan, példákon keresztül kerül bemutatásra. Arra kérjük az olvasót, hogy pontos szintaxisleírásért és a szemantika precízebb tárgyalásáért látogassa meg a nyelvi specifikációt: https://www.haskell.org/definition/haskell2010.pdf és ebben keresse fel a kapcsolódó fejezetet. Megjegyezzük továbbá, hogy a nyelv újabb, GHC által definiált változatai eltérhetnek az itt tárgyaltaktól és jóval több nyelvi absztrakciót támogatnak.

Érdekesség

Multiparadigma programozás

A Haskell tisztán funkcionális programozási nyelv, de a funkcionális programozási paradigma elemeit más paradigmákkal együtt is lehet alkalmazni, ahogyan teszik ezt a modern multiparadigma nyelvek. Lehet tehát egy nyelv egyszerre imperatív, objektum-orientált és funkcionális, részben vagy egészben valósítva meg a különböző paradigmák elemeit. Pythonban például leírható az összegzés iterációval:

```
for n in a:
 sum += n
```

De ugyanúgy megadható rekurzióval is:

```
def sum(a):
 if len(a)==0: return 0
 else:         return a[0] + sum(a[1:])
```

A különböző paradigmák elemeit szabadon keverhetjük a programunkban, mindig épp azt alkalmazva, amelyik jobban illeszkedik a feladathoz.

Matematikai kifejezések

Ahogy említettük, a funkcionális programozás tárgyalását a megszokott matematikai kifejezések, függvénydefiníciók felől közelítjük, és a lambda-kalkulusra, mint mögöttes számítási modellre vezetjük vissza. Mielőtt megkezdjük a függvénydefiníciók tárgyalását, bevezetünk alapvető típusokat és megmutatjuk, hogyan írhatunk fel velük kifejezéseket, és hogyan értékelődnek azok ki.

A Haskell, ahogyan a legtöbb funkcionális nyelv, képes egyszerű matematikai kifejezések kiértékelésére. Vegyük például a következő formulát:

$$1 + 2$$

A természetes vagy egész számok körében ez a *3* értékkel lenne *egyenlő*. A Haskell értelmezőjében (`ghci`) is kiértékelhető kifejezésként, és az elvárásainknak megfelelően a `3` értékre értékelődik ki:

```
ghci> 1 + 2
3
```

A fenti kódblokkban az első sorban a kiértékelendő kifejezés látható, a második sorban pedig a kiértékelés eredménye.

> **Megjegyzés**
>
> Mindig, amikor az értelmezőben való kiértékelést mutatunk meg, akkor ezt a formátumot használjuk majd, tehát a `ghci>` rész jelöli a kiértékelendő kifejezést, az ezt követő sorokban pedig az eredmény lesz látható.

A matematikai (halmazelméleti) jelöléssel ellentétben a Haskellben minden kifejezésnek fordítási időben meghatározásra és ellenőrzésre kerül a típusa. Úgy mondjuk, hogy a nyelv **statikusan típusozott**. A kikövetkeztetett típust le is lehet kérdezni a `:t` parancs segítségével:

```
ghci> :t 1 + 2
1 + 2 :: Num a => a
```

Az eredmény sorban látható a kifejezésünk, amelyet a `::` jelölés követ (ez a *típusozás* művelete), majd megadásra kerül maga a típus: `Num a => a`. Ez a típus első látásra zavarba ejtő lehet, de egyelőre maradjunk annyiban, hogy ez mindösszesen annyit jelöl, hogy a kifejezés típusa valamilyen szám.

A `::` nem csak az eredménysorban jelenhet meg. A nyelv lehetőséget ad arra, hogy mi magunk megadjuk a kifejezések és részkifejezések típusát. Ha például megjelöljük, hogy az `1` és `2` egész számok, akkor az összeadás eredménye is egész szám lesz:

```
ghci> :t (1::Int) + (2::Int)
(1::Int) + (2::Int) :: Int
```

> **Fontos**
>
> Néhány beépített típustól eltekintve a típusok nevei mindig nagybetűvel kezdődnek.

> **Érdekesség**
>
> A későbbiekben látni fogjuk, hogy a `+` valójában nem nyelvi elem vagy beépített művelet, hanem egy szabványos könyvtárbeli függvény, amelyet `Int` típusú értékekre alkalmazva `Int` típusú értéket kapunk.

Alaptípusok: számok, karakterek, logikai értékek

A jegyzet igyekszik minden programozási szerkezetet visszavezetni elemibb konstrukciókra, ám az egyszerűség kedvéért néhány alapvető típust, azok literáljait és műveleteit adottnak vesszük; így teszünk az egész és lebegőpontos számokkal, a karakterekkel és a logikai értékek típusával is.

Int, Integer. Az egész számokat az `Int` és `Integer` típusok valósítják meg. A legtöbb esetben az `Int` típust fogjuk használni (ez fix bitszélességen ábrázolt gépi egész szám), de matematikai számítások esetén az `Integer` típust tanácsos használni (ez tetszőlegesen nagy számot tárol). A típusok literáljai a megszokott decimális (oktális, hexadecimális) értékek, és a szokásos alapműveletek alkalmazhatóak rájuk (újabb nyelvi verziókban bináris számliterálok is használhatóak).

```
ghci> 0o10 + 0x10 - 10
14
```

Talán az egyetlen érdekesség, hogy az egész osztást a `div` művelet jelöli, amelyet ráadásul fordított félidézőjelek (*backtick*) közé kell tenni:

```
ghci> 22 `div` 10
2
```

> **Megjegyzés**
>
> A fenti kódban mindegyik művelet, beleértve a `div` műveletet is, valójában függvényalkalmazás. Az összeadással, kivonással és szorzással ellentétben a `div` azért kerül fordított félidézőjelek közé, mert nem operátorszimbólum, hanem közönséges függvénynév, és az infix használatát engedélyezzük a jelölés segítségével.

> **Fontos**
>
> A Haskellben két gyakran használt egész típus is van: az `Int` gépi egész, túl tud csordulni, az `Integer` viszont alkalmas tetszőlegesen nagy egészek tárolására és ábrázolására (nincs ábrázolási korlátja). Előjel nélküli egészeket a `Word` típussal írhatunk le, de ez is gépi szám, rögzített bitszélességen ábrázolva, és a gyakorlatban ritkán használjuk.

Float, Double. A lebegőpontosan ábrázolt valós számok típusai a `Float` és a `Double`, literáljaik a megszokottak, és csakúgy mint a legtöbb programozási nyelvben, az ábrázolásuk a IEEE 754 szabvány szerinti. A `/` művelet eredménye törtszám, amellyel demonstrálhatjuk az ábrázolás pontatlanságát:

```
ghci> 0.999999999 :: Double
0.999999999
ghci> 1/9 + 1/9 + 1/9 + 1/9 + 1/9 + 1/9 + 1/9 + 1/9 + 1/9 ::
Double
1.0000000000000002
```

Char. A karakter típus literáljai aposztrófok közé írt karakterek, számos műveletük közül kiemeljük az `ord` és `chr` függvényeket, amelyekkel karakterkóddá alakíthatóak, illetve karakterkódból konstruálhatóak.

```
ghci> import Data.Char
ghci> 'a'
'a'
ghci> ord 'a'
97
ghci> chr 97
'a'
```

> **Érdekesség**
>
> A karakter típus jelentősége abban rejlik, hogy a *String* típus ennek segítségével lesz megadva: a *String* egyszerűen karakterek listája.
>
> A fenti példában látható, hogy a karakter típus műveletei a `Data.Char` modulban érhetőek el, amelyet az `import` parancs segítségével értünk el.

Bool. A logikai értékek típusa a szokott módon két értéket tartalmaz, az igaz és a hamis értéket. Fontos, hogy a literáljaikat nagybetűkkel kell kezdeni, a műveleteik a szokásosak (például negáció, konjunkció, diszjunkció).

```
ghci> True && False
False
ghci> True || not False
True
```

> **Érdekesség**
>
> Azért kell nagybetűvel kezdeni a `True` és `False` literálokat, mert a számliterálokkal ellentétben a logikai típus esetében az értékek nem nyelvi elemek, hanem algebrai adattípus konstruktorai — ezeket később részletesen tárgyaljuk.

> **Érdekesség**
>
> Az alaptípusokról több információt találhatsz a *Base* csomag dokumentációjában: Bool, Int, Integer, Char, Float.

Erős típusrendszer, explicit típuskonverzió

Többször említettük, hogy a Haskell statikusan típusozott és erős típusrendszert (strong type system) használ. Ez azt jelenti, hogy a kifejezések típusozhatóságát/típushelyességét fordítási időben (a kiértékelés megkezdése előtt) ellenőrzi a nyelv implementációja, és futási idejű típushibák lényegében nem tudnak előfordulni, mert implicit típuskonverziók nem történnek.

Meglepő lehet például, hogy habár az `Integer` típus (amelynek nincsen felső korlátja) az `Int` típus (amely fix bitszélességen van ábrázolva) minden elemét magába foglalja (foglalhatná), még sincs a "szűkebb" típusról automatikus (implicit) konverzió a "bővebb" típusra.

```
ghci> (1::Int) + (2::Integer)

<interactive>:97:13: error:
    • Couldn't match expected type ‘Int’ with actual type
‘Integer’
```

Explicit konverzióval azonban orvosolható a típushiba:

```
ghci> toInteger (1::Int) + (2::Integer)
3
```

A nyelv kikényszeríti, hogy a típuskonverziókat mindig explicit elvégezzük, erre figyeljünk oda a gyakorlatok során.

Függvénykifejezések

Az előző fejezetben arra a megállapításra jutottunk, hogy a funkcionális programozásban tulajdonképp matematikai kifejezéseket lehet felírni, és ezeket egy értelmező kiértékeli, meghatározza az értéküket. Az, hogy a funkcionális program egy tiszta, matematikai kifejezés, az egyik legfontosabb megállapításunk, a paradigma másik sarokpontja azonban a függvényközpontúság: minden függvény, még a függvények műveletei is függvények.

A bevezetőben említettük, hogy a funkcionális programozás számítási modellje a lambda-kalkulus, amelyet Alonzo Church 1930-as években alkotott meg számítások függvényekkel való leírására. A lambda-kalkulus felfogható egy olyan formális (matematikai) nyelvnek, amelyben függvényeket írhatunk, épp úgy, mint Haskellben, és bár látni fogjuk, hogy a Haskell egy kifejezetten gazdag nyelv, sokféle szintaxissal, kifejező és sokrétű alaptípusokkal, mégis szinte az egész nyelv visszavezethető a lambda-kalkulus alapvető konstrukcióira.

A lambda-kalkulus

A matematikusok a 17. századtól kezdve használtak a formulákban változókat ismeretlenek jelölésére, és ennek egyfajta továbbfejlesztése a lambda-kalkulus, amelyben a változók bevezetése és eliminálása jól definiált, lehetővé teszi a számítások leírását függvények segítségével. A lambda-kalkulus Alonzo Church munkásságának eredménye az 1930-as évekből, de az 1960-as évektől kezdve épültek rá a fontosabb eredmények.

A lambda-kalkulus nem más, mint egy formális nyelv függvények és függvényalkalmazások leírására. Fontos különbség a szokásos matematikai jelöléshez képest, hogy minden függvény egyparaméteres, és minden függvényalkalmazás prefix formában van megadva. Tehát a szokásos "f(x) = 3x" matematikai jelölés helyett a “λx. * 3 x” leírást kapjuk a kalkulusban. Ezen a jelölésen a Haskell és más programozási nyelvek természetesen lazítanak, de a háttérben ugyanúgy egyparaméteres függvények és prefix függvényalkalmazások kerülnek kiértékelésre.

> **Érdekesség**

> A lambda-kalkulus a λ szimbólumról kapta a nevét, amely a függvényabsztrakció (változó bevezetés) jelölése.

A kalkulus formális definíciója szerint négy konstrukció van a nyelvben: konstans, változó, absztrakció, applikáció.

$$e ::= c \mid x \mid \lambda x.\, e \mid e_1 e_2$$

A nyelv szemantikája egyetlen szabállyal van megadva, amely helyettesítést végez (úgy hívják, hogy beta-redukció). A beta-redukció során a függvény törzsében a változót helyettesítjük az argumentum kifejezéssel, tehát a helyettesítés adja meg a jelentést.

$$(\lambda x.\, e_1) e_2 \Rightarrow e_1[e_2/x]$$

A korábbi példánál maradva vegyük az *x + y* kifejezést, amelyben a változókat absztraháljuk, majd alkalmazzuk az *1* és *2* értékekre, akkor az *1 + 2* kifejezést kapjuk:

$$((\lambda x.\, (\lambda y.\, x + y))1)2 \Rightarrow (\lambda y.\, 1 + y)2 \Rightarrow 1 + 2$$

Amikor egy kifejezésben több helyettesíthető/egyszerűsíthető részkifejezés van (úgynevezett *redex*), akkor a kiértékelési stratégia határozza meg, hogy milyen sorrendben kell haladni.

> **Fontos**
>
> **Kiértékelési stratégia**
>
> Haskellben az alapértelmezett kiértékelési stratégia lusta (egészen pontosan *call-by-need*), a függvényargumentumot csak akkor egyszerűsíti (hozza normálformára), ha szükség van rá a függvénytörzs kiértékeléséhez (egy kifejezés normálformában van, ha nincs benne egyszerűsíthető részkifejezés). A nyelv azonban arra is lehetőséget ad, hogy bizonyos esetekben mohó kiértékelést kényszerítsünk ki, például ha hatékonysági okokból a függvényargumentumot mielőbb érdemes leegyszerűsíteni, hogy ne "szivárogjon" el a memória az egyébként kis méretű értéket leíró nagy méretű kifejezés ábrázolására.

> **Fontos**

Amíg az absztrakció jele a λ szimbólum (Haskellben `\`), addig az applikációnak nincs műveleti jele. Ezt érdemes megszoknunk, mert ez Haskellben is így lesz, tehát a függvény és a paraméterét (paramétereit) egyszerűen egymás mellé írjuk.

Érdekesség

A lambda-kalkulusban minden kiszámítható függvény leírható, kifejezőereje megegyezik a Turing gépével.

Megjegyzés

A lambda-kalkulusról készült összefoglalót a következő forrásból készítettük: link.

Lambda kifejezések Haskellben

A funkcionális programozási nyelvek magjában felfedezhető a nyilvánvaló kapcsolat a lambda-kalkulussal: a Haskell nyelvnek is alapvető elemei a függvényabsztrakció (lambda-függvény) és a függvényalkalmazás. Előbbit a `\` jelöli (ami a λ jelből ered), utóbbinak pedig nincs műveleti jele.

Az alkalmazás művelete, ahogy a lambda-kalkulusban, Haskellben is bináris: minden függvénynek egyetlen paramétere van, és minden függvényalkalmazás egyetlen argumentumot alkalmaz. Ez az úgynevezett *curry forma*.

Ennek következménye, hogy többparaméteres függvények nincsenek. Ha két absztrakciót ágyazunk egymásba, akkor nem kétparaméteres függvényt konstruálunk, hanem egy olyan függvényt, amelyet alkalmazva egy újabb függvényt kapunk. A következő példában egy olyan függvényt adunk meg lambda-jelöléssel, amely két absztrakciót foglal magába, és a belső függvény alkalmazása után az első paraméter értékére egyszerűsödik:

```
\x -> \y -> x -- vagy tömörebben: \x y -> x
```

Ahogyan már említettük, minden kiértékelhető kifejezésnek jól definiált típusa van Haskellben, amelyet az interpreter a `:t` parancsra meg is tud jeleníteni:

```
ghci> :t \x -> \y -> x
\x -> \y -> x :: p1 -> p2 -> p1
```

Az értelmező kikövetkeztette a *legáltalánosabb* (principális) típust a kifejezéshez:

$$p_1 \to p_2 \to p_1$$

Ez azt írja le, hogy a fenti egyszerű lambda-kifejezés típusa függvénybe ágyazott függvény (hasonló, mint egy kétparaméteres függvény, de nem egészen az), az argumentumok típusai tetszőlegesek, az eredmény típusa pedig megegyezik az első argumentum típusával.

> **Fontos**
>
> A fenti típusban a `p1` és `p2` úgynevezett **típusváltozók**, tetszőleges típussal helyettesíthetőek. A típusban a `->` a függvénytípus konstrukciót jelöli és jobbra zárójeleződik.

> **Fontos**
>
> **A `->` különböző jelentései**
>
> A `->` szimbólumot már legalább két különböző kontextusban használtuk. Amikor lambda-absztrakciókat írtunk, akkor a lambda (`\`) jelet követően a változónév és a törzskifejezés elválasztására használtuk, amikor pedig függvények típusát adtuk meg, akkor a függvénytípus jelölésére. Például:
>
> ```
> id :: a -> a
> id = \x -> x
> ```
>
> Vegyük észre, hogy a két nyíl két különböző szintaktikus szerepben van és különböző a jelentésük. Az első nyíl a típuskifejezés része és függvénytípust jelöl, míg a lenti nyíl a függvényabsztrakció kifejezés szintaktikus eleme. Később még ugyanennek a nyílnak további szerepei is lesznek a nyelvben.

A függvénykifejezést alkalmazva konkrét értékekre kiértékelhető, leegyszerűsíthető:

```
ghci> (\x -> \y -> x) 1 2
1
```

Az absztrakciók jobbra, a függvényalkalmazások balra zárójeleződnek. A fenti példa tehát a következőképp zárójeleződik:

```
ghci> ((\x -> (\y -> x)) 1) 2
1
```

Fontos

A függvényalkalmazás balra zárójelezése miatt óvatosnak kell lennünk az olyan összetett kifejezésekkel, mint például a többszöri logikai negáció. Például ha kétszer akarjuk alkalmazni a negációt a hamis értékre, kézzel kell zárójeleznünk az alárendelt részt:

```
ghci> not not False
<interactive>:11:1: error:
• Couldn't match expected type ‘Bool -> t’ with actual type
‘Bool’
ghci> not (not False)
False
```

A hibaüzenet egyébként abból fakad, hogy az értelmező a `not` függvény akarja a `not` függvényre alkalmazni.

Megjegyzés

Ahogyan a fenti példából az nyilvánvaló, az első fejezetben bevezetett alaptípusok (logikai és szám típusok) műveletei mind függvények, alkalmazásuk egyszerű függvényalkalmazás.

Fontos

A függvényalkalmazás művelete

A függvényalkalmazásnak van explicit műveleti jele is, a dollárjel (`$`) operátor. Mivel a dollárjel operátor relatív precedenciája alacsony (és jobb asszociatív), akkor fogjuk a gyakorlatban használni, amikor el akarjuk kerülni a függvénykifejezés vagy az argumentumkifejezés zárójelezését.

Az előző példában látott zárójelezés például elhagyható a függvényalkalmazás művelet használata mellett:

```
ghci> not $ not False
False
```

Ahogyan az alábbi példában az összetett egész kifejezés zárójelezése is elhagyható:

```
ghci> (\x -> x) (1 + 2)
3
ghci> (\x -> x) $ 1 + 2
3
```

Fontos

Kifejezések elnevezése

A saját kifejezéseinknek nevet adhatunk, így egy bonyolult kifejezést (programot) kisebb kifejezésekre (részprogramokra) bonthatunk. A nevekre hivatkozhatunk bármely részkifejezés helyén.

```
ghci> first = \x -> \y -> x
ghci> first 1 2
1
```

Curryzés, részleges alkalmazás

Leszögeztük, hogy minden függvény egyparaméteres, ún. curry-zett formában van. A többszörös függvényabsztrakcióval megadott függvények alkalmazásával egy eggyel kevesebb paraméteres függvényt képzünk; amikor egy ilyen függvénynek nem alkalmazzuk minden absztrakcióját, akkor **részleges függvényalkalmazás**t végzünk.

Egy látszólag kétparaméteres függvényt lehet alkalmazni egyetlen paraméterre, ekkor az eredmény egy egyparaméteres függvény, amelyet részleges alkalmazással kapunk:

```
ghci> first = \x -> \y -> x
ghci> :t first True False
first True False :: Bool
ghci> :t first True
first True :: p2 -> Bool
```

Fontos

A részleges alkalmazást az infix operátorok is támogatják. Ha az összeadásnak csak a baloldalát vagy jobboldalát akarjuk megadni, az úgynevezett "section" szintaxist használhatjuk:

```
ghci> :t (1+)
(1+) :: Num a => a -> a
```

Ezt a fajta jelölést az értelmező természetesen függvény részleges alkalmazássá bontja ki:

```
ghci> :t (+) 1
(+) 1 :: Num a => a -> a
```

Fontos megjegyezni azonban, hogy ez a jelölés a kivonás (`-`) művelet jobboldali operandusának rögzítésére nem alkalmazható, mert az összetéveszthető lenne az aritmetikai negációval.

Valódi többparaméteres függvények viszont n-esekkel szimulálhatóak. A szabványos könyvtárban van támogatás a két reprezentáció közötti váltásra: a `curry` és `uncurry` műveletek. Az előző példában látott függvény például "kétparaméteressé" alakítható, és ilyenkor nem lehet részlegesen alkalmazni (tehát a külső függvényt alkalmazni a belső alkalmazása nélkül):

```
ghci> :t uncurry first
uncurry first :: (c, b) -> c
ghci> uncurry first (1,2)
1
```

Megjegyzés

A fenti példában a `(c, b)` tetszőleges két típus párjának típusa (az n-es kétparaméteres esete). Az összetett típusok tárgyalásánál elő fognak kerülni az n-esek és részletesebben bemutatjuk majd őket.

Fontos

A `curry` és `uncurry` valójában nem nyelvi elemek, csupán szabványos könyvtárbeli függvények:

```
curry   f  x  y  = f (x, y)
uncurry f (x, y) = f  x  y
```

A típusuk pedig a következő:

```
ghci> :t curry
curry :: ((a, b) -> c) -> a -> b -> c
ghci> :t uncurry
uncurry :: (a -> b -> c) -> (a, b) -> c
```

Vegyük észre, hogy a függvénytípusok explicit zárójelezése itt szükséges, hiszen enélkül más típus adódna a jobbra zárójelezés miatt.

Függvénykompozíció

A következő fontos függvényművelet, amelyet megadunk, a függvénykompozíció. Ez az egyik legelemibb módja függvények kombinálásának, és ezzel a funkcionális paradigmában a program építésének: a kompozíció segítségével úgy képzünk két függvényből egy újabbat, hogy az egyik kimenetét a másik bemenetére irányítjuk — ennek imperatív gondolkodásbeli megfelelője a szekvencia.

A kompozíció művelete Haskellben a `.` szimbólummal van jelölve. A komponált függvény típusa természetesen a komponált függvények típusától függ.

```
ghci> inc = \(x::Int) -> x + 1
ghci> dec = \(x::Int) -> x - 1
ghci> :t (inc . dec)
(inc . dec) :: Int -> Int
ghci> (inc . dec) 42
42
```

Fontos megjegyeznünk, hogy a kompozíció sem nyelvi elem Haskellben. Pontosan úgy definiálható, ahogyan lambda-kalkulusban is megadható lenne: függvényabsztrakciókat függvényekkel alkalmazunk, melynek hatására olyan függvényt készítünk, amely "összekapcsolja" a paraméterfüggvényeket:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
(.) = \f -> \g -> \x -> f (g x)
```

Ismétlés

A kompozíció mintájára felírhatunk egy olyan függvényt, amely pontosan háromszor alkalmazza az argumentumfüggvényét:

```
threetimes = \f -> \x -> f (f (f x))
```

Figyeljük meg, hogy itt is függvénytípust következtet ki az `f` paraméterre az értelmező, hiszen függvényalkalmazás baloldalaként szerepel a kifejezésben:

```
ghci> :t threetimes
threetimes :: (t -> t) -> t -> t
```

> **Fontos**
>
> Azokat a függvényeket, amelyeknek az argumentuma függvény típusú, **magasabb rendű függvény**eknek nevezzük.

A `threetimes` függvényt az inkrementálásra és a 10 konstansra alkalmazva az elvárt 13 értéket kapjuk.

```
ghci> threetimes inc 10
13
```

A fenti függvényt természetesen a kompozícióra építve is megadhatjuk:

```
threetimes = \f -> f . f . f
```

> **Megjegyzés**
>
> A Haskell nyelv lehetőséget ad arra, hogy a paramétereket a név mellett soroljuk fel, mintha matematikai függvényt adnánk meg egyenlőséggel — a háttérben azonban a fenti függvénykifejezésre fog kifejtődni:
>
> ```
> threetimes f = f . f . f
> ```
>
> Sőt, akár az `f` függvény paramétere is explicitté tehető:
>
> ```
> threetimes f x = f (f (f x))
> ```
>
> Ez az összefüggés a függvényparaméter elrejtéséről és megjelenítéséről általában is igaz, és *eta-ekvivalenciának* hívják: az `f` kifejezés és a `\x -> f x` kifejezések ekvivalensek egymással (bármilyen kontextusban felcserélhetőek).

Iteráció és rekurzió

Hogyan tudjuk a fenti `threetimes` függvényt általánosítani tetszőleges számú iterációra? Ehhez ciklus szerkezetre lenne szükségünk, ami a funkcionális nyelvekben nem áll rendelkezésre. Van viszont rekurzió, amellyel természetesen ki tudjuk fejezni az ismétlődést. Haskellben a függvénynevek lehetnek rekurzívak (a függvények alkalmazhatják magukat), így megadhatunk egy iteráló függvényt, amely az első argumentuma szerinti sokszor alkalmazza a függvényargumentumát:

```
iter :: Integer -> (Integer -> Integer) -> Integer -> Integer
iter n f x = iter (n-1) f (f x)
```

Azt írjuk tehát le, hogy az *n* lépéses iteráció lényegében egy lépés, majd *n-1* további lépés összekapcsolásaként definiálható. Azonban ahhoz, hogy ez a definíció megfelelően működjön, alapesetre is szükség van, különben a rekurzió sosem terminál.

Esetszétválasztással megadott függvény

Az iterációnak valójában két esete van: az alapeset (*n = 0*), amelyben a definíció nem rekurzív, és a rekurzív eset, amelyet fentebb megadtunk. Ezt a függvényt esetszétválasztással definiálhatjuk:

$$iter(n, f, x) = \begin{cases} x & \text{if} \quad n = 0 \\ iter(n-1, f, f(x)) & \text{otherwise} \end{cases}$$

A fenti definíciót Haskellben úgy írjuk fel, hogy a két esetet egymás alá írjuk:

```
iter :: Integer -> (Integer -> Integer) -> Integer -> Integer
iter 0 f x = x
iter n f x = iter (n-1) f (f x)
```

A fenti definíció két sora teljes egészében megfelel a matematikában használt esetszétválasztásos függvény fogalmának, de a részletekről (mintaillesztés, őrfeltételek) később fogunk beszélni.

> **Fontos**
>
> A több egyenlőséggel definiált függvények esetében fontos az esetek megadásának sorrendje (először írjuk fel a konkrétabb, aztán az általánosabb eseteket).

Megjegyzés

Negatív értékekre nyilvánvalóan hibásan viselkedik a fenti függvény, a későbbiekben ezt őrfeltételekkel fogjuk kivédeni.

Hogyan is fog lépésről lépésre átíródni az `iter` függvény alkalmazása? Definíció behelyettesítések és beta-redukciók történnek. Lássunk egy példát, amikor a `threetimes` függvénnyel megegyező eredményt ad a *3* iteráció megadásakor:

```
iter 3 f x                  =>
iter (3-1) f (f x)          =>
iter 2 f (f x)              =>
iter (2-1) f (f (f x))      =>
iter 1 f (f (f x))          =>
iter (1-1) f (f (f (f x))) =>
iter 0 f (f (f (f x)))      =>
f (f (f x))
```

Az iterációs függvényt megfelelő számú argumentummal alkalmazva az elvárt viselkedést mutatja:

```
ghci> iter 10 inc 1
11
```

Megjegyzés

Amennyiben a függvény típusát kézzel megadjuk és `Int` típusú számokkal dolgozunk, akkor a gyakorlatban is tapasztalhatjuk, hogy ezek gépi egészek (fix méreten ábrázoljuk őket).

```
ghci> iter 30 (\x -> x*x) 20
0
```

Érdekesség

Explicit rekurzió nélkül is kifejezhető a rekurzió (ismétlés) fixpontként való kifejezéssel. Például a faktoriális kiszámításához használt definíció

```
fact 0 = 1
fact n = n * fact (n - 1)
```

kifejezhető rekurzió nélkül a következő függvény fixpontjaként:

```
fact' f 0 = 1
fact' f n = n * f (n - 1)
```

Fontos észrevenni, hogy míg a `fact` függvény rekurzív (önmagát alkalmazza), addig a `fact'` függvény **nem** rekurzív, viszont a `fact'` fixpontja megegyezik a `fact` függvénnyel, és nem kell hozzá rekurzió (sőt, a fixpont leírásához sem kell explicit rekurzió). A fixpontot kiszámító fixpontkombinátor a következőképp adható meg (ad-hoc típusrendszer gyengítéssel):

```
-- y = \f -> (\x -> f (x x)) (\x -> f (x x))
import Unsafe.Coerce
y = \f -> (\x -> f (unsafeCoerce x x)) (\x -> f (unsafeCoerce
x x))
```

Ezek után a faktoriális számítás az explicit rekurzió nélkül is leírható és az elvárt módon viselkedik:

```
fact 5 == (y fact') 5
```

Az Y-kombinátor működését nem tárgyaljuk ebben a jegyzetben, de biztatjuk az olvasót a témában való elmélyedésre. Megjegyezzük továbbá, hogy természetesen a Haskell nyelv és a magnyelve is támogatja a rekurzív definíciókat, mert lényegesen olvashatóbbak és jobban kezelhetőek a fordítás során, viszont jó tudni, hogy a lambda kifejezések is képesek rekurzív függvények leírására.

Az iteráció függvény részleges alkalmazása

A korábbiakban tárgyaltak szerint az `iter` nem háromparaméteres függvény, hanem háromszoros függvényabsztrakció: az iterációszám megadásával egy újabb absztrakciót kapunk, majd az iterált függvény megadásával szintén csak egy függvény az eredmény. Érdemes észrevenni, hogy ahány alkalmazást végzünk, annyi függvénykonstruktor tűnik el a típusból, hiszen az alkalmazás "kioltja" az absztrakciót (pont, mint lambda-kalkulusban):

```
ghci> :t iter 10
iter 10 :: (Integer -> Integer) -> Integer -> Integer
ghci> :t iter 10 (+1)
iter 10 (+1) :: Integer -> Integer
```

Fontos észben tartani, hogy a függvényalkalmazás (amelynek itt nincs külön műveleti jele, egyszerűen csak egymás mellé írjuk a kifejezéseket) balra zárójeleződik. Amennyiben jobbra zárójelezzük az alkalmazásokat, típushibát kapunk:

```
ghci> :t iter (10 (+1))
<interactive>:1:7: error:
No instance for (Num ((Integer -> Integer) -> Integer)) [...]
```

A részleges függvényalkalmazás tehát a többszörös absztrakció valahány szintjének alkalmazásával kapott kifejezés, amely maga is függvény típusú:

```
ghci> :t iter 10 (+1)
iter 10 (+1) :: Integer -> Integer
```

Amennyiben az összes argumentum megadásra kerül, a kifejezés már nem függvény típusú:

```
ghci> :t iter 10 (+1) 50
iter 10 (+1) 50 :: Integer
```

Ezt kiértékelés után már ki is tudjuk írni a képernyőre:

```
ghci> iter 10 (+1) 50
60
```

Polimorfizmus

Vajon az *iter* függvény használható lenne logikai értékeken logikai függvények ismételt alkalmazására? A jelenlegi típusmegjelölése szerint nem:

```
ghci> iter 10 not True
<interactive>:18:9: error:
Couldn't match type 'Bool' with 'Integer' [...]
```

Megtehetjük persze, hogy készítünk belőle egy másik változatot, amely logikai függvényeket tud iterálni:

```
iterBool :: Integer -> (Bool -> Bool) -> Bool -> Bool
iterBool 0 f x = x
iterBool n f x = iterBool (n-1) f (f x)
```

A fejlett típusrendszer azonban megkímél minket a kódduplikálásról. Lehetőségünk van arra, hogy tetszőleges típuson értelmezett homogén függvényt iteráljunk, ha a konkrét típust egy típusváltozóval helyettesítjük:

```
iter :: Integer -> (a -> a) -> a -> a
iter 0 f x = x
iter n f x = iter (n-1) f (f x)
```

A típus általánosításával (de a függvénydefiníció módosítása nélkül) többalakúvá tettük a függvényünket, és most már `Integer` és `Bool` (és valójában bármely) típusú értékeken vett függvény iterálását el tudja végezni:

```
ghci> iter 10 not True
True
ghci> iter 9 not True
False
```

Fontos

Az így kapott `iter` függvény **parametrikusan polimorf**: az iterált függvény bemeneti és kimeneti típusa tetszőleges, a függvényünk viselkedése független tőle. Ez tehát nem túlterhelés, a különböző típusokhoz (pl. `Integer` vagy `Bool`) nem adunk különböző definíciókat, hanem a definíció parametrikus a típus mentén és egyetlen definícióval adtuk meg a függvényt tetszőleges `a` típushoz.

Megjegyzés

Típuskikövetkeztetés

Vajon a fenti a legáltalánosabb típus, amelyet az `iter` függvényhez meg lehet adni? Nos, ha egyáltalán nem adjuk meg a függvény típusát, a Haskell típuskikövetkeztetője fogja kikövetkeztetni a lehető legáltalánosabb típust.

```
ghci> :t iter
iter :: (Eq a, Num a) => a -> (b -> b) -> b -> b
```

Látható, hogy ez az általunk megadott típusnál is általánosabb, mivel az `Integer` helyén az `a` típusváltozó szerepel. Van viszont előtte egy feltétel: az `a` által jelölt típusnak olyannak kell lennie, amelyre értelmezett az egyenlőségvizsgálat (`Eq`) és a számok alapműveletei (`Num`). Az `Eq` és a `Num` úgynevezett típusosztályok, típuscsoportok, amelyekbe konkrét típusok beletartozhatnak (hasonlóak az interfészekhez). Az `Integer` természetesen beletartozik mindkét csoportba, de sok-sok egyéb típus is lehetne az `a` helyén. A típusok csoportjaival (típusosztályokkal) a későbbiek folyamán még részletesebben foglalkozni fogunk.

Vajon miért pont az `Eq` és `Num` típusmegszorításokat következtette ki a típuskikövetkeztető? Nyilván az iterációszámláló használatával van összefüggésben. A `Num` megszorítás abból fakad, hogy eggyel csökkentjük a rekurzív ágban a számlálót, tehát az összeadás-kivonás műveleteket támogatnia kell a számláló típusának. Az `Eq` pedig azért került oda, mert az esetszétválasztásunk (miszerint 0 vagy sem a számláló) egyenlőséget vizsgál a 0 értékkel.

Érdekesség

Az `iter` függvényben látott típusváltozók valójában a függvénynek ugyanúgy paraméterei, mint az értékek, csak a helyükre típusok helyettesítődnek. A típusok mentén történő absztrakció és applikáció a programban alapértelmezetten nem látható, de a GHC `TypeApplications` kiterjesztése lehetővé teszi, hogy megfigyeljük őket. Az identitásfüggvény, amelynek típusa `a -> a`, valójában paraméterezve van az `a` típussal, és ha alkalmazzuk az `Int` konkrét típusra, akkor megkapjuk az `Int` feletti identitásfüggvényt:

```
ghci> :t id
id :: a -> a
ghci> :set -XTypeApplications -fprint-explicit-foralls
ghci> :t id
id :: forall a. a -> a
ghci> :t id @Int
id @Int :: Int -> Int
```

A típusváltozó szerepében: a függvény

Tehát kellően általános típusmegjelöléssel az `iter` függvényt tudtuk `Integer` és `Bool` függvények iterálására is használni, és az iterációszámláló helyén is szerepelhet bármilyen szám típus. Vajon az `iter` arra is használható, hogy függvénytranszformációt iteráljunk vele? Mi lenne, ha olyan függvényt adnánk meg, amely függvényeket alakít át más függvényekké, és ezt iterálnánk? Az identitásfüggvényt például bármennyiszer alkalmazhatjuk az identitásfüggvényen, és ugyanazt a függvényt kapjuk vissza:

```
ghci> (iter 10 id id) 10
10
```

Függvényt függvényre úgy is tudunk képezni, ha a bemeneti függvényt komponáljuk egy másik függvénnyel. Ha értjük a kompozíció műveletét, felhasználhatjuk az iterációs függvényünket arra, hogy egy függvényt n-szer komponáljunk egy másikhoz:

```
ghci> (iter 3 (\f -> f . (+1)) id) 10
13
```

Itt a `\f -> f . (+1)` kifejezés az `f` függvényhez rendel egy olyan függvényt, ami az `f` és a `(+1)` függvény kompozíciója. Ha ezt 3-szor iteráljuk az identitásfüggvényen, akkor megkapjuk azt a függvényt, amely háromszor ad hozzá egyet bármely számhoz:

```
id . (+1) . (+1) . (+1)
```

A lambda-kalkuluson túl

Ebben a fejezetben a függvények megadásakor az alapvető származtatott eszközeink a *kompozíció*, *rekurzió* (iteráció) és az *esetszétválasztás* voltak. Azt megmutattuk, hogy a kompozíció, a rekurzió és az iteráció visszavezethető a lambda-kalkulus elemi műveleteire; az esetszétválasztást nem tárgyaltuk ilyen részletességgel, de hamarosan visszatérünk rá. Az alaptípusokat és literáljaikat is adottnak vettük, bár valójában azokat is be lehetne ágyazni a lambda-kalkulusba, úgynevezett *Church encoding* segítségével (a literálok mind megadhatóak tiszta függvényekkel, lásd itt). A funkcionális programozási nyelvek elviekben szinte teljes egészében beágyazhatóak a lambda-kalkulusba, azonban a legtöbb modern funkcionális nyelv a lambda-

kalkulus általánosabb, minimálisan kiegészített változatára vezeti vissza a konstrukcióikat.

A Haskell esetében ez egészen világosan van kommunikálva és az implementációban is megfigyelhető: a *ghc (Glasgow Haskell Compiler)* köztes nyelve a *Haskell Core*, amely felfogható a lambda-kalkulus kiegészítéseként. A Haskell Core egy olyan programozási nyelv, amelyben a lambda-kalkulus leírásnál olvashatóbb, hatékonyabb programokat lehet leírni, de szintaxisában alig különbözik tőle. Ennek az alapvető, kompakt funkcionális nyelvnek része a lambda-kalkulus, de azon túl tartalmaz lokális definíciókat (let kifejezés), mintaillesztéses esetszétválasztást (`case` kifejezés) és további kiegészítéseket a típusrendszer általánosabbá tételéhez.

Let kifejezések

A korábbi fejezetekben már láttunk olyan példákat, ahol több függvény definícióját követte egy kiértékelendő kifejezés. Ez a funkcionális programok szabványos felépítése, miszerint megadunk névvel ellátott függvényeket, amelyeket felhasználva felírunk egy kifejezést, amelyet értékké egyszerűsítünk. Az *inc-dec* példa bár nem modulszerűen volt megadva, de jól szemlélteti ezt a szerkezetet. Tegyük fel, hogy megadunk két függvényt:

```
inc = \x -> x + 1
dec = \x -> x - 1
```

Majd megadunk egy kifejezést, amelyet ki akarunk értékelni:

```
inc (inc (dec 10))
```

Ez a szerkezet egyetlen kifejezéssel leírható, amelyben a függvények lokális definícióként vannak megadva; ennek a szerkezetnek a neve a `let` kifejezés. A funkcionális magnyelvben két fontos célt is szolgálnak a lokális definíciók: egyrészt lehetővé teszik, hogy részkifejezéseket elnevezzünk és újrahasználjunk (és ennek köszönhetően olvashatóbbá tegyük a funkcionális programot), másrészt a kiértékelés optimalizálható azzal, hogy a többször előforduló részkifejezések értékét csak egyszer határozzuk meg.

Nemcsak a Haskell Core szintjén ábrázolhatóak a definíciók `let` kifejezésekkel, a Haskell konkrétan nyelvi elemként tartalmazza a `let` kifejezést. A fenti példát a következőképp tudjuk leképezni `let` kifejezésekre, szükségtelenné téve a globális definíciókat:

```
ghci> let inc = \x -> x + 1 in let dec = \x -> x - 1 in inc (inc
(dec 10))
11
```

Ebben a példában két `let` kifejezés van egymásba ágyazva, és a sor végén az `in` után lévő kifejezés az, amit az értelmező végül a megelőző definíciókat figyelembe véve kiszámít (leegyszerűsít).

> **Megjegyzés**
>
> Bár a `let` kifejezések a Haskell nyelvben is elérhetőek, a gyakorlatban a programozók az úgynevezett `where` függelékeket használják, amelyekkel meg tudják adni a lokális definíciókat "segédfüggvény" formában, a kiértékelendő kifejezés után (nem pedig előtte):
>
> ```
> r = inc (inc (dec 10)) where { inc = \x -> x + 1 ; dec = \x -
> > x - 1 }
> ```
>
> Ezt több sorba is törhetjük, megfelelő indentálással:
>
> ```
> r = inc (inc (dec 10))
> where
>   inc = \x -> x + 1
>   dec = \x -> x - 1
> ```
>
> A fentiek közül általában az utóbbit preferáljuk (elsősorban az olvashatóság miatt), de az első változat akkor is használható, amikor egyetlen sorban kell leírni a teljes kifejezést (például értelmezőben történő felíráskor).

> **Fontos**
>
> ### Margószabályok
>
> Az előző példában indentálással "alárendelt" szerkezetet az úgynevezett margószabályok segítségével tudjuk létrehozni. A fő kifejezés alá sorakoznak fel a lexikálisan és szintaktikailag is alárendelt lokális definíciók, amelyeknek határait indentálással jelöljük ki.

> A margószabályok lehetővé teszik, hogy fehér szóközökkel alakítsunk ki blokkszerkezetet, beágyazott kódrészleteket a programban, elkerülve a kapcsos zárójelek használatát. Ez egy olyan nyelvi elem, amely számos modern multiparadigma nyelvben elérhető, segítségével olvashatóbb, tömörebb kódot lehet írni. A Haskell esetében a betartandó szabály az, hogy az alárendelt kódrészleteknek bentebb kell kezdődni az indító sornál, illetve azonos szinten lévő elemek felsorolásakor azonos kell legyen a behúzás mértéke.
>
> A Haskell nyelvben négy nyelvi szerkezet támogatja a margószabály segítségével történő kódszervezést: a `let`, a `where`, a `case ... of` és a `do` (utóbbit jelen jegyzet nem tárgyalja). A nyelvi specifikáció vonatkozó fejezete részletesen magyarázza a kapcsolódó lexikai és szintaktikai szabályokat.

Case kifejezések

Az iterációfüggvényt esetszétválasztással definiáltuk, és Haskellben úgy fejeztük ki, hogy két definíciós sort írtunk fel hozzá a két különböző esetre:

```
iter 0 f x = x
iter n f x = iter (n-1) f (f x)
```

Ez azonban egy olyan szerkezet, amelyet a lambda-kalkulusban nem tárgyaltunk és a Haskell Core sem tartalmazza. Azt láttuk, hogy a lambda-kalkulusban semmilyen beépített esetszétválasztás vagy elágazás nincs, de a Haskell Core ezzel szemben tartalmaz esetszétválasztó konstrukciót, és a fenti kódsor valójában erre fejtődik ki. Egészen pontosan lambda-kifejezéseken belül egy *case* kifejezés írja le az esetszétválasztást:

```
iter = \n -> \f -> \x -> case n of 0 -> x ; n -> iter (n-1) f (f
x)
```

Két eset jelenik meg, épp úgy, mint a többszörös definiáló egyenlőségnél: a *0* és az *n* esetet; az *n* helyett természetesen szerepelhetne a `case` kifejezésen belül egy másik változó, például *m*, de így is a megfelelő számítást írtuk le.

A későbbiekben látni fogjuk, hogy minden esetszétválasztás `case` kifejezésekre fog egyszerűsödni a fordítóban, teljesen mindegy, hogy milyen magas szintű konstrukcióval írtuk azt le a Haskell nyelvben.

> **Megjegyzés**

A `case` kifejezés valójában nem egyszerű esetszétválasztás, hanem mintaillesztés az érték "formája", felépítése alapján. A jegyzet későbbi fejezeteiben erre még visszatérünk, és alaposabban tárgyaljuk a mintaillesztés működését.

> **Megjegyzés**
>
> A `case` kifejezésben is használhatunk margószabályt az esetek szétválasztására és strukturálására, de a gyakorlatban általában inkább a függvénydefiníció szintjén fogjuk megcsinálni az esetszétválasztást, mert jobban olvasható kódot eredményez.

Őrfeltételek

Az előző pontban tett állítást megerősítjük egy újabb példával, miszerint a `case` kifejezésekre képződnek le az esetszétválasztások.

Ismétlésül, az iterációt a következő szintaxissal adtuk meg:

```
iter 0 f x = x
iter n f x = iter (n-1) f (f x)
```

Sokszor az ilyen esetszétválasztást, amelyben kettő vagy több egymást kizáró esetet akarunk megadni és logikai feltételekkel is leírhatóak az esetek, az őrfeltétel nyelvi szerkezettel adjuk meg:

```
iter n f x | n == 0 = x
           | n /= 0 = iter (n-1) f (f x)
```

Ez olvashatóbb kódot tud eredményezni, mert az esetek feltételei explicit jelennek meg az esetszétválasztásban. Fontos, hogy ilyenkor az esetek őrfeltételei csak logikai (`Bool`) típusú kifejezések lehetnek.

> **Megjegyzés**
>
> Őrfeltételes definíciókban szokás használni az `otherwise` feltételt az utolsó, "egyéb esetek" sorhoz, azonban jó tudni, hogy az `otherwise` nem kulcsszó, csupán egy szabványos függvény, amely az igaz értékkel van definiálva.
>
> ```
> ghci> otherwise
> True
> ```

Az előző példa tehát a következőképp is felírható:

```
iter n f x | n == 0     = x
           | otherwise = iter (n-1) f (f x)
```

If-then-else. Először is bevezetjük az if-elágazás kifejezést, amely hasonlóan az őrfeltételekhez, `Bool` típusú kifejezés mentén fejez ki esetszétválasztást. Az előző példa felírható az if-then-else szerkezettel, ahol a függvény őrfeltételeiből az elágazás feltételei lesznek:

```
iter n f x = if n == 0 then x else iter (n-1) f (f x)
```

És valóban, az if-then-else is kifejezhető `case` kifejezéssel megadott esetszétválasztásra, hiszen két esetet vizsgálunk: az egyikben igaz a feltétel, a másikban hamis.

```
iter n f x = case n == 0 of True -> x ; False -> iter (n-1) f (f
x)
```

Végül a paraméterek lambda függvényekké alakítása után a következő kifejezést kapjuk:

```
iter = \n -> \f -> \x -> case n == 0 of True -> x ; False -> iter
(n-1) f (f x)
```

Megjegyzés

A többszörös függvényabsztrakció a lambda konstrukció szintjén is összevonható a Haskell szintaxisa szerint:

```
iter = \n f x -> case n == 0 of True -> x ; False -> iter (n-
1) f (f x)
```

Fontos

A tiszta kifejezések körében az if-then-else nem vezérlési szerkezet, sokkal inkább esetszétválasztás, amely a kérdőjel-kettőspont operátorra hasonlít az imperatív nyelvekből; a funkcionális világban azonban a vezérlésfolyam nem definiált, csupán kiválasztás történik a két esetszétválasztás eredményei között.

Érdekesség

A Haskell Core is egy formális nyelvre épül, amely a lambda-kalkulus kiegészítése, úgy hívják, hogy System FC kalkulus. A fenti lambda-kalkulus kiegészítéseken kívül tartalmaz típusabsztrakciót, típusapplikációt, típuskényszerítést, típuskonverziót (és típusok közötti egyenlőségvizsgálatot), amelyekkel a Haskell gazdag típusrendszerének minden eleme kifejezhető.

Megjegyzés

Fontos leszögezni, hogy a Haskell programozási nyelv általunk tárgyalt résznyelve kifejezhető lambda-kalkulusban is (például az előbb tárgyalt `case` és `if` kifejezések is kifejezhetőek a lambda-kalkulus elemi konstrukcióival), tehát elsősorban a hatékonyságot javítja, ha a gyakorlatban nem vezetjük vissza a nyelvi elemeket a legelemibb konstrukciókig. A Haskellben vannak azonban olyan kifejezőbb nyelvi elemek is, amelyeket csak a kiegészített magnyelvre lehet lefordítani (ezeknek a lényege, hogy típusok közötti egyenlőségvizsgálatra van szükség, amelyet a nemcsak a lambda-kalkulusban, de a System F kalkulusban is bonyolult leírni).

Listakifejezések

Ebben a fejezetben foglalkozunk a funkcionális nyelvek egyik nevezetes és érdekes típusával, a listával. A Haskell listái különösen érdekesek: polimorfak (tetszőleges az elemek típusa), homogének (csak azonos típusú elemek lehetnek benne), potenciálisan végtelenek, és számos különböző kifejezéssel meg lehet őket adni. A Haskell lista, mint minden más adatszerkezet a nyelvben, *immutable*, tehát nem lehet megváltoztatni vagy felülírni a benne tárolt értékeket. Először bemutatjuk magát a típust, az alapvető műveleteit, a listákon keresztül bevezetjük a mintaillesztést, majd megmutatjuk, hogyan lehet intervallumkifejezéseket és halmazkifejezéseket rekurzív függvényekkel megadni.

Felsorolással megadott listák

A listák leggyakrabban használt literálja a szögletes zárójelek között, vesszővel felsorolt elemek (ahol az elemek tetszőleges részkifejezések, bármilyen összetettségűek lehetnek):

```
ghci> [1,2,3]
[1,2,3]
ghci> [1,1+1,1+1*2]
[1,2,3]
```

Természetesen az elemekkel felsorolt listában bármely más típusú elemeket is írhatunk, nemcsak számokat, akár listákat is írhatunk a listába:

```
ghci> [[1], [2,3]]
[[1],[2,3]]
```

A lista homogén, tehát különböző típusú elemeket nem adhatunk meg egyazon listában, mert az típushibát eredményezne:

```
ghci> [1::Int, True::Bool]

<interactive>:2:10: error:
    • Couldn't match expected type ‘Int’ with actual type ‘Bool’
```

> **Megjegyzés**

> Az a lista, amelyben számok különböző hosszúságú listáit soroltuk fel, nem okoz típushibát, mert a lista hossza nem része a típusnak.

A listák szerkezete

Jó tudni, hogy a felsorolás (szögletes zárójelek között, vesszővel elválasztva) csak egy jelölés; a listák valójában mindössze kétféle művelettel vannak megkonstruálva: van üres lista és van nemüres lista. Az üres lista konstruktora a `[]`, a nemüres lista konstruktora a `(:)`, ahol utóbbi egy elemet kapcsol össze egy másik (maradék) listával. A jelölés tehát ezen konstruktorműveletek alkalmazását rövidíti:

```
ghci> 1:2:3:[]
[1,2,3]
```

A `:` művelet alkalmazását nemcsak infix jelöléssel adhatjuk meg, hanem prefix jelöléssel is, viszont akkor zárójelek közé kell tenni (ez a prefix jelölés egyébként minden más infix operátor esetén is használható):

```
ghci> (:) 1 [2,3]
[1,2,3]
ghci> (:) 1 ((:) 2 ((:) 3 []))
[1,2,3]
```

Itt azonban már egyre szembetűnőbb, hogy függvényalkalmazásokkal építjük fel a listát. És valóban, a `(:) 3 []` egy függvényalkalmazás, ahol a `:` a függvény, a `3` és a `[]` pedig az argumentumok. Ezek "beépített" jelölések, de "hagyományos" függvényeket jelölnek, és amikor listát készítünk, ezeket a függvényeket alkalmazzuk. A típusaik is lekérdezhetőek a `:t` paranccsal:

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

> **Megjegyzés**
>
> Vegyük észre, hogy itt megint találkoztunk típusváltozóval, tehát a típus polimorf, az listaelem bármilyen típusú lehet. Az `[a]` típusban az `a` tetszőleges típust jelöl, tehát a lista típust bármilyen más típussal specializálhatjuk, és megadhatunk például listák listáját vagy akár függvények listáját is.

> Egy másik fontos észrevétel, hogy a lista típuskonstruktor neve a `[]`, pontosan úgy, mint az üres lista neve is, a kettőt viszont különböző kontextusokban lehet csak használni, így ez nem okoz félreértéseket.

Fejelem, maradéklista

Ahogyan meg lehet konstruálni a listákat a fejelem és a maradéklista megadásával, úgy dekonstruálni is lehet őket ennek mentén. A *head* művelet tetszőleges nemüres lista fejelemét adja meg, a *tail* pedig a nemüres lista maradékát adja meg:

```
ghci> :t head
head :: [a] -> a
ghci> :t tail
tail :: [a] -> [a]
```

Például:

```
ghci> head [1,2,3]
1
ghci> tail [1,2,3]
[2,3]
ghci> head (tail [1,2,3])
2
```

Ezek a műveletek nem beépítettek, mint látni fogjuk, egyszerű mintaillesztéssel definiálhatjuk őket a listák szerkezete szerint.

> **Megjegyzés**
>
> A listákba bele is lehet indexelni (0-tól kezdve), tehát adott indexű eleméhez "közvetlen" hozzáférhetünk:
>
> ```
> ghci> [1,2,3] !! 1
> 2
> ```
>
> Fontos tudni azonban, hogy ez nem konstans, hanem lineáris műveletigényényű művelet, hiszen a listaelemek láncolva vannak tárolva a memóriában; emiatt ezt a műveletet ritkán fogjuk a gyakorlatban alkalmazni.

Lustaság, végtelen listák

A fejezetben több ízben találkozunk majd olyan kifejezésekkel, amelyek végtelen listákat definiálnak. A végtelen adatszerkezetet természetesen nem tudjuk véges időben kiértékelni, de ha csak véges sok elemre van szükség belőle, akkor kifejezetten praktikus lehet bizonyos problémákat végtelen listák felhasználásával megoldani.

Példaként álljon itt egy triviális, végtelen rekurzióval megadott végtelen lista, amely csupa `1` értékeket tartalmaz:

```
ghci> ones = 1:ones
ghci> ones
[1,1,1,1,1,1,1,1, ...
```

Ha megpróbáljuk kiértékelni az értelmezővel, akkor "végtelen sokáig" sorolja fel a listában lévő 1-eseket, tehát valóban végtelen listát definiáltunk. Viszont a Haskell lusta kiértékelési stratégiájának köszönhetően végtelen listák véges prefixeit ki tudjuk értékelni. Ha például a végtelen listának csak a fejelemét próbáljuk kiértékelni, a végtelen sok elem kiszámítása helyett csak az elsőt fogja kiszámítani az értelmező és sikeresen befejeződik a kiértékelés:

```
ghci> head ones
1
```

Vagy akár használhatjuk a `take` műveletet is, amely (maximum) az első argumentumának megfelelő hosszú prefixét veszi a listának (ennyi elemet "vesz ki belőle"):

```
ghci> take 10 ones
[1,1,1,1,1,1,1,1,1,1]
```

Természetesen a `take` sem speciális vagy beépített, hagyományos, rekurzív függvénydefinícióval van megadva a szabványos könyvtárban, és mintaillesztéssel gyűjti a maradéklisták fejelemeit addig, amíg a lista el nem fogyott, vagy a lekérdezendő elemek száma el nem érte a nullát.

Mintaillesztés és iteráció listákon

Az iterációfüggvény esetében láttuk, hogy függvényeket lehet esetszétválasztással definiálni, és azt is megmutattuk, hogy a függvényszintű esetszétválasztás valójában alacsonyabb szinten `case` kifejezésekkel kerül kifejtésre. A `case` alapú esetszétválasztás nemcsak eseteket különböztet meg, de mintaillesztést is végez: megállapítja, milyen szerkezetű az érték, amelyen az esetszétválasztás történik, és összetett adatok esetén "kicsomagolja" az adattagokat.

Azt mondtuk, hogy a listákat kétféleképpen lehet megadni: üres lista konstruktorral és nemüres lista konstruktorral; ezek pontosan a lista szerkezetének lehetséges esetei. Ezen túl pedig a második esetben két adattag határozza meg a konstrukciót: a fejelem és a maradéklista.

Amikor mintaillesztéses esetszétválasztást végzünk listákon, akkor egyrészt megkülönböztetjük a két esetet, másrészt a nemüres esetben kiolvassuk a lista alkotórészeit. Ennek tudatában a `head` és `tail` műveletek megvalósítása függvénnyel nyilvánvalóvá válik, hiszen csak annyit kell tennünk, hogy a függvény bemenetéből kiolvassuk a fejelemet és a maradéklistát, és azokra egyszerűsítjük a listát:

```
head []     = undefined
head (x:xs) = x
```

```
tail []     = undefined
tail (x:xs) = xs
```

> **Fontos**
>
> Ahogyan a korábbi példákban a `:` műveletet használtuk a lista konstruálására, most ugyanezt a műveletet használjuk a dekonstruálásra: az `(x:xs)` minta az `x` és `xs` változókba köti ("menti") a lista fejét és maradékrészét, amelyre a definiáló egyenlőség jobboldalán már tudunk hivatkozni.

> **Megjegyzés**
>
> Az `undefined` egy speciális függvény, amellyel azt tudjuk jelezni, hogy a függvény valamely esetben nem definiált, parciális.

Listaiteráció

Azt már korábban leszögeztük, hogy Haskellben nincsen sem ciklus, sem más iterációs vagy vezérlésfolyam nyelvi elem. Ám azt is láttuk azonban, hogy az iterációt ki tudtuk fejezni rekurzió segítségével. Természetesen adódik a gondolat, hogy írhatunk olyan rekurziót, amelyben a lista első, második, harmadik, negyedik, stb. elemét vesszük (indexeléssel, a `!!` művelettel), de ennél van egy sokkal praktikusabb és pragmatikusabb megoldás. A rekurziót elvégezhetjük magán a listán: mivel minden nemüres lista a `(:)` művelettel készül, ugyanezen konstruktor mentén szét is bontható fejelemre és maradékrészre, majd a maradékelemen újra alkalmazható a feldolgozó függvény. Ez a listákon vett **primitív rekurzió**. Ezzel a sémával megadható például tetszőleges lista hosszát kiszámító függvény:

```
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs
```

Ha ezt a szétválasztást rekurzívan csináljuk, akkor végig tudunk iterálni a teljes listán, majd a rekurziót az üres lista esete zárja.

> **Megjegyzés**
>
> Végtelen listára a `length` kiszámítása természetesen nem terminál, nincsen "végtelen" listahossz.

Számos nevezetes listafüggvény van, amelyet ugyanezzel a rekurziósémával adhatunk meg. Ilyen például a listákat összefűző, listakonkatenáció művelet, amelyben a baloldali listát dolgozzuk fel ilyen módon, és valójában csak annyi történik, hogy a jobboldali listához egyenként hozzáfűzzük a baloldali lista elemeit:

```
append :: [a] -> [a] -> [a]
append []     ys = ys
append (x:xs) ys = x : append xs ys
```

> **Megjegyzés**
>
> A listakonkatenáció műveletének standard megvalósítása a `++` infix operátor.

Ennél egy fokkal bonyolultabb az a művelet, amely egyszerre dolgozza fel a két paraméterlistát: a `zip` segítségével két lista elemeit párosítjuk egy eredménylistába:

```
zip :: [a] -> [b] -> [(a, b)]
zip [] ys = []
zip xs [] = []
zip (x:xs) (y:ys) = (x, y) : zip xs ys
```

> **Megjegyzés**
>
> Vajon hogyan működik a `zip` különböző hosszúságú listák összefésülése esetén?

Nevezetes magasabb rendű listafüggvények

A fenti rekurziósémával számos nevezetes listafüggvényt meg lehet adni. Ahogy korábban már említettük, a funkcionális programozási paradigma egyik legfontosabb pillére a magasabb rendű függvény, amelyet "számítással paraméterezett számításként" említettünk, olyasmi, mint az algoritmikus minták. A listákra vett magasabb rendű függvények általános bejárásokat, szűréseket, transzformációkat írnak le valamilyen konkrét művelet paraméterében. Nevezetes példák a `map` és a `filter`, de a későbbiekben megnézzük majd a "reduce" műveletét is, a hajtogatást (`fold` függvények).

Map. A `map` olyan általános műveletet ír le, ahol iterációban minden listaelemre szeretnénk alkalmazni egy függvényt. Ez egy rendkívül általánosan alkalmazható művelet és sokféle konkrét feladatot fogunk visszavezetni rá. Az rekurzív függvényben itt is két esetet adunk meg: az üres listát üres listára képezzük, a nemüres lista esetében pedig a fejelemre és a maradékrészre is alkalmazzuk a paraméterfüggvényt, majd visszaépítjük a listát:

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

A fejelemre úgy alkalmazzuk a paramétert, hogy egyszerű függvényalkalmazást használunk (`f x`), a maradéklista elemeire pedig a `map` rekurzív alkalmazásával tudjuk alkalmazni a paraméterfüggvényt (`map f xs`).

> **Megjegyzés**

Mivel a rekurzív eset itt is eggyel rövidebb listára alkalmazza rekurzívan a függvényt, véges lista esetén a rekurzió garantáltan terminál.

Fontos

A map paraméterfüggvénye `a -> b`, tehát tetszőleges típusú listákat tud kezelni, a típusparaméterekre nincsen semmilyen megkötés (megszorítás).

A következő példa a `map` és az `even` függvények segítségével egy egész számokat tartalmazó listában minden elemről eldönti, hogy páros-e:

```
ghci> map even [1,2,3]
[False,True,False]
```

Látható, hogy itt az első típusparaméter helyére a számok típusa kerül, a második típusparaméter helyére pedig a logikai értékek típusa helyettesítődik.

Filter. A `filter` is nevezetes magasabb rendű függvény, és a `map`-hez hasonlóan primitív rekurziót végez a listán, de itt az elemekre alkalmazott függvény nem egy tetszőleges leképezés, hanem egy predikátum (tehát a függvényparaméter típusa `a -> Bool`). A `filter` azokat a listaelemeket foglalja az eredménylistába, amelyekre a predikátum teljesül (igazra értékelődik ki):

```
filter :: (a -> Bool) -> [a] -> [a]
filter p []     = []
filter p (x:xs) | p x       = x:filter p xs
                | otherwise = filter p xs
```

Érdekesség

A fenti, őrfeltételekkel megadott definíció természetesen a korábbiakban tárgyaltak szerint kifejtődik egy `if` szerkezetre:

```
filter p []     = []
filter p (x:xs) = if p x then x:filter p xs else filter p xs
```

Amelyből végül `case` esetszétválasztás lesz:

```
filter p []     = []
filter p (x:xs) = case p x of True -> x:filter p xs ; False -
> filter p xs
```

Ahogyan magából a függvényfejben felírt esetszétválasztásból is:

```
filter p xs = case xs of [] -> [] ; (x:xs) -> case p x of
True -> x:filter p xs ; False -> filter p xs
```

A paraméterek pedig lambda absztrakciókban adhatók meg:

```
filter = \p -> \xs -> case xs of [] -> [] ; (x:xs) -> case p
x of True -> x:filter p xs ; False -> filter p xs
```

Megjegyzés

A `zip` általánosításaként megadható a `zipWith`, amely már magasabb rendű; ez a változat nem tényleges párokat képez a listából, hanem a függvényparaméterét alkalmazza az elempárokra.

Intervallumkifejezések

Listát nem csak literállal vagy a konstruktorfüggvényekkel tudunk megadni, vannak egyéb listakifejezések is: intervallumok (range) és listagenerátorok (list comprehension). Fókuszáljunk most az elsőre. Az intervallumkifejezés egy alsó és egy felső korlát közötti elemeket sorolja fel, és egy rendkívül kényelmes módja listák megadásának Haskellben és más nyelvekben is:

```
ghci> [1..4]
[1,2,3,4]
ghci> [1..1]
[1]
ghci> [1..0]
[]
```

Fontos

Amennyiben az intervallum üres (a felső korlát kisebb, mint az alsó), akkor az eredménylista üres.

A korábbi gondolatmenetet követve megmutatjuk, hogy az intervallumkifejezések valójában kifejezhetőek rekurzív függvényekkel, és hogy a Haskell értelmező egyszerűen az intervallumkifejezés esetén ezeket a rekurzív függvényeket alkalmazza.

Kezdjük a kifejezés jelentésének megértésével. Tegyük fel, hogy az intervallum egy jelölés arra, hogy iteratívan hozzáfűzzük az elemeket az üres listához egy alsó korlát és egy felső korlát között:

```
[n..m] === n:(n+1):(n+2):   ...   :(m-2):(m-1):m:[]
```

Ebből jól látszik, hogy az intervallumlistát megadó függvénynek egy ilyen konstruktorsorozatot kell meghatároznia. Most megmutatjuk, hogyan lehet egy egyszerű rekurzív függvénnyel megadni az egyszerű intervallumkifejezések jelentését. A kulcs a rekurzió: fejezzük ki az intervallumot egy eggyel kisebb intervallum segítségével:

```
[n..m] === n:[(n+1)..m]
```

A függvényünk tehát az alsó korláthoz kapcsolja az alsó korlátnál eggyel nagyobb számmal kezdődő intervallumot (a felső határt megtartva). Fontos, hogy a rekurziót "meg kell állítani", ezért kell egy esetszétválasztás, ahol az egyik eset alapeset (tehát nem rekurzív). Itt most a megállás feltétele az, hogy az alsó korlát túllép a felső korláton. Használjunk if-then-else esetszétválasztást a rendezési reláció teljesülése szerint:

```
numericEnumFromTo n m = if n>m then [] else n : numericEnumFromTo
(n+1) m
```

Vegyük észre, hogy a rekurzió mindenképp terminál, mert az alsó korlát mindig el fogja érni a felsőt (hiszen minden rekurziós lépés során eggyel csökken közöttük a differencia).

```
ghci> numericEnumFromTo 1 3
[1,2,3]
ghci> numericEnumFromTo 1 1
[1]
ghci> numericEnumFromTo 1 0
[]
```

Felső korlát nélküli intervallum

Az intervallumkifejezésnek van olyan változata, amely nem ad meg felső korlátot, hanem felsorolja az összes elemet valamilyen alsó korláttól indulva (a lépésköz itt is 1, ahogy a felső korlátos változatnál).

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

A rekurzió sémája gyakorlatilag ugyanaz, mint az előző példánál, csak itt nincs alapeset:

```
[n..] === n:[(n+1)..]
```

Rekurzív függvénnyel felírva a jelölés helyett:

```
numericEnumFrom n = n : numericEnumFrom (n+1)
```

Továbbá az `[n..m]` jelentése visszavezethető az `[n..]` jelentésére:

```
numericEnumFromTo' n m = take (m-n+1) (numericEnumFrom n)
ghci> numericEnumFromTo 1 10
[1,2,3,4,5,6,7,8,9,10]
ghci> numericEnumFromTo' 1 10
[1,2,3,4,5,6,7,8,9,10]
```

> **Megjegyzés**
>
> Azt gondolhatnánk, hogy az `[n..]` formájú kifejezések mindig végtelen listát generálnak, de ez nem feltétlenül igaz. Amennyiben a típusnak a megadott kezdőértéktől indulva már csak véges sok eleme van, akkor az intervallum (lista) is véges (emlékezzünk, hogy az *Int* fix bitszélességen ábrázolt egész, tehát van legnagyobb értéke):
>
> ```
> ghci> [(9223372036854775805::Int)..]
> [9223372036854775805,9223372036854775806,9223372036854775807]
> ```

Intervallumok 1-től eltérő lépésközzel

A nyelv arra is lehetőséget ad, hogy felülbíráljuk a lépésközt: két első elem megadása esetén úgy generáljuk a listát, hogy a két elem közötti különbséget használjuk lépésközként. A felső korlát elhagyható, de most nézzük azt a változatot, amikor megadjuk a felső határt.

```
ghci> [1,3..10]
[1,3,5,7,9]
ghci> [1,3..11]
[1,3,5,7,9,11]
ghci> [1,5..11]
[1,5,9]
```

Ahogy az előzőekben is, rekurzívan fogalmazzuk meg a jelentést, az első elemet leválasztva, a maradék (eggyel rövidebb listát) újabb intervallumként megadva:

```
[n,m..k] === n:[n+(m-n),m+(m-n)..k]
```

A megállási feltételünk (amennyiben *m>n* teljesül) most is az lesz, hogy *n* átlépte a *k* felső határt:

```
numericEnumFromThenTo n m k = if n > k then [] else n :
numericEnumFromThenTo m (m+m-n) k
```

```
ghci> numericEnumFromThenTo 1 5 11
[1,5,9]
```

> **Érdekesség**
>
> A *numericEnumFromThenTo* implementációjának negatív lépésközzel is megfelelően kell működnie, tükrözve az intervallumkifejezés jelentését.
>
> ```
> ghci> [10,9..1]
> [10,9,8,7,6,5,4,3,2,1]
> ```
>
> Gondold át, hogy ehhez hogyan kell megváltoztatni a definíciót.

Polimorf intervallumok

Azt gondolhatnánk, hogy a fenti függvények csak a számok kontextusában használhatóak, de ez nem így van. Egyrészt az intervallumkifejezésekhez megadott függvényeink nagyrészt általánosak (a `+1` művelet lecserélhető az általánosabb, rákövetkezőt megadó `succ` műveletre), másrészt lényegében bármely típusra alkalmazható a felsorolás, ha megadjuk, hogyan képezhetőek az elemei az `Int` típus elemeire.

Például ha a logikai hamisat megfeleltetjük a 0-nak, az igazat pedig az 1-nek, akkor a logikai értékek felsorolása visszavezethető a számok felsorolására:

```
ghci> [False ..True]
[False,True]
```

Vagy ha a karaktereket megfeleltetjük a karakterkódjuknak, akkor a karakterek felsorolása szintén visszavezethető a számok felsorolására:

```
ghci> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"
```

De mi történik a háttérben? Nos, explicit konverzió a kérdéses típusról az `Int` típusra, majd a felsorolás után visszafelé irányú konverzió. Minden olyan típus, amelyre ez értelmezett, beletartozik az `Enum` típuscsoportba (típusosztályba). Az explicit konverziós függvények pedig:

```
ghci> :t fromEnum
fromEnum :: Enum a => a -> Int
ghci> :t toEnum
toEnum :: Enum a => Int -> a
ghci> fromEnum 'a'
97
ghci> toEnum 97 :: Char
'a'
```

A kettőt összekapcsolva tehát bármely felsorolható típus intervallumkifejezése lényegében visszavezethető a fentebb definiált rekurzív függvényeinkre:

```
ghci> [fromEnum 'a'..fromEnum 'z']
[97,98,99,100,101,102,103,104,105,106,107,108,109,110,111,112,113,
114,115,116,117,118,119,120,121,122]
ghci> map toEnum [fromEnum 'a'..fromEnum 'z'] :: [Char]
"abcdefghijklmnopqrstuvwxyz"
```

És a törtszámok intervallumai?

Na azok egy kicsit bonyolultabbak. A fenti módszer sajnos nem úgy adja meg az intervallumlistát, mint ahogy a `Float` vagy `Double` típusokra működnek az intervallumkifejezések:

```
ghci> [0.0..0.9]
[0.0,1.0]
ghci> map toEnum [fromEnum 0.0..fromEnum 0.9] :: [Float]
[0.0]
```

Ez azért van, mert a levegőpontos számok esetében másképp van definiálva a felsorolás: a felső határt meghaladó érték is bekerülhet az eredménylistába, amennyiben a lépésköz felénél nem több a különbség (részleteket lásd a nyelvi specifikációban):

> *For Float and Double, the semantics of the enumFrom family is given by the rules for Int above, except that the list terminates when the elements become greater than e3 + i/2 for positive increment i, or when they become less than e3 + i/2 for negative i.*

Az intervallumkifejezés jelentése a fenti leírás alapján természetesen finomítható, aminek köszönhetően a lebegőpontos számokat is helyesen kezeljük:

```
numericEnumFromThen     n m   = n : numericEnumFromThen m (m+m-n)
numericEnumFromThenTo' n m k = takeWhile (<= k+((m-n)/2))
(numericEnumFromThen n m)
```

> **Érdekesség**

A fenti definíciók, intervallumkifejezések nemcsak megmutatják, hogy hogyan lehet az intervallumokat elemi funkcionális programozási eszközökkel (rekurzióval, esetszétválasztással) függvényekként definiálni, hanem egészen közel állnak ahhoz, ahogy a Haskell értelmező és fordító kezeli az intervallumkifejezéseket (a fordítás során valóban ilyen, majdnem pontosan így definiált függvényekre fejtődnek ki).

Halmazkifejezések

Ahogy az imént láttuk, intervallumkifejezésekkel (pl. `[1..10]`) véges és végtelen listákat is tudunk generálni, amennyiben a lista elemtípusa felsorolható. Listákat generálni azonban nemcsak intervallumokkal lehet, hanem úgynevezett halmazkifejezésekkel is. A halmazkifejezés (vagy listagenerátor kifejezés) lehetővé teszi, hogy egy vagy több listát bejárva és megszűrve, egy újabb listát állítsunk elő.

A következő kifejezés például a 10-nél kisebb, páros, pozitív egész számok kétszeresét sorolja fel:

```
[ 2*x | x <- [1..10] , even x ]
  ---   ------------   ------
   E          G           F
```

Megjegyzés

Az `even` függvény logikai értéke igaz, ha az argumentumszám páros, egyébként pedig hamis értékre egyszerűsödik.

A fenti példában felfedezhető egy generátor (`G`), amely felsorolja az egészeket 1 és 10 között (az előző fejezetben látott intervallumkifejezéssel), kiválogatja közülük a párosakat egy szűrővel (`F`) és végül kiértékeli a generált és válogatott elemekre a fő kifejezést (`E`), amely kétszerezi a kiválasztott számokat.

```
ghci> [2*x | x <- [1..10], even x]
[4,8,12,16,20]
```

Ahogyan az intervallumkifejezések esetén, itt is visszavezethető az egyébként bonyolultnak tűnő konstrukció jelentése elemi kifejezésekre. A halmazkifejezés kibontásához egy rekurzív algoritmust használunk, és a benne foglalt generátor kifejezések listáin rekurzív függvénnyel fogunk végigiterálni.

> **Megjegyzés**
>
> Fontos: amennyiben több generátor van a listában, azok szekvenciálisan, egymásba ágyazva kerülnek bejárásra, nem pedig párhuzamosan (egy időben). Továbbá a filter (szűrő, válogató) kifejezések (amelyek *Bool* típusúak) az összes tőlük balra eső generált értékre mondhatnak ki feltételeket.

```
ghci> [x*y | x <- [1..10], even x, y <- [1..5], x+y<5]
[2,4]
```

A halmazkifejezés kibontása

A Haskell 2010-es nyelvi dokumentációja tárgyalja, hogy a következő egyszerű szabályrendszer szerint kell leegyszerűsíteni (kibontani) a halmazkifejezéseket:

- Ha a kifejezés `[e | True]` alakú, tehát csak az azonosan igaz szűrő szerepel benne, a szűrő elhagyható és az `[e]` listakifejezést kapjuk.
- Ha a kifejezés `[e | q]` alakú (ahol `q` tetszőleges generátor vagy szűrő), akkor kiegészíthető az azonosan igaz szűrővel, kanonikus alakra hozva: `[e | q, True]`.
- Ha a kifejezés `[e | b , Q]` alakú (ahol `b` egy szűrő, `Q` pedig generátorok és szűrők sorozata), akkor a szűrő jelentése kifejezhető egy elágazással, amelyben a halmazkifejezés további generátorait és szűrőit csak a kiválogatott elemekre alkalmazzuk: `if b then [e | Q] else []`.
- Ha a kifejezés `[e | x <- [n..m], Q]` alakú (`Q` generátorok és szűrők sorozata), akkor a generátor jelentése kifejezhető egy olyan iterációval, ahol az egyes elemekre alkalmazzuk a halmazkifejezés maradékrészét és összefűzzük az eredményeket: `(f n) ++ ... ++ (f m)` ahol `f x = [e | Q]`.

> **Megjegyzés**

> Emlékeztető: a `++` a listák összefűzésének művelete.

Valójában az utolsó esetet általánosabban is megadhatjuk: a listagenerátor baloldalán nemcsak változó, hanem tetszőleges minta szerepelhet (a mintákról majd később még értekezünk), a jobboldalán pedig tetszőleges listakifejezés. A listán való iterációt és összefűzést egy rekurzív, magasabb rendű függvénnyel valósítjuk meg (`concatMap`), amely a lista összes elemére alkalmaz egy listát eredményező függvényt, és az eredménylistákat összefűzi.

A szabályrendszer tehát egészen pontosan a következőképp néz ki:

```
1. [ e | True ]       = [e]
2. [ e | q ]          = [ e | q, True ]
3. [ e | b, Q ]       = if b then [ e | Q ] else []
4. [ e | p <- l, Q ]  = let f p = [ e | Q ]
                            f _ = []
                        in concatMap f l
```

A 4. szabály tehát annyival bonyolultabb, hogy az *f* függvényt lokálisan definiálja (az *f* definíciója csak a `let` kifejezésen belül érvényes) és a `concatMap` függvényt használja az általános listakifejezés elemeinek bejárásához.

Egy konkrét példa kifejtése

Nézzük, hogy miként néz ki a motivációs példaként megadott halmazkifejezés kibontása a fenti szabályokkal. A kiinduló kifejezésünk tehát a következő:

```
[2*x | x <- [1..10], even x]
```

Látható, hogy az 1, 2 és 3 átírási szabályok nem alkalmazhatóak rá, de a 4-es szabály igen. Az átírás a generátort kibontja (eltünteti), a helyébe kerül a lokális függvény és a `concatMap` segítségével megadott iteráció:

```
let f x = [2*x | even x]
    f _ = []
in concatMap f [1..10]
```

Látszik, hogy a halmazkifejezésünk bonyolultsága csökkent, megszabadultunk egy generátortól. A következő lépésben az 1. szabály nem, de a 2. alkalmazható, és hozzáadunk egy triviális `True` feltételt. Ez a lépés technikai, nem csökkenti a kifejezésünk méretét vagy komplexitását, de

lehetővé teszi, hogy alkalmazhassuk a 3., majd azt követően az 1. szabályt.

```
let f x = [2*x | even x, True]
    f _ = []
in concatMap f [1..10]
```

Most már alkalmazhatjuk a 3. szabályt, amellyel eliminálni tudjuk a szűrőt a halmazkifejezésből, kibontva és kifejezve a jelentését az elágazással:

```
let f x = if even x then [ 2*x | True ] else []
    f _ = []
in concatMap f [1..10]
```

Az utolsó lépésben pedig használhatjuk az 1-es szabályt és eltörölhetjük a triviális szűrőt a halmazkifejezésből. Ezzel a halmazkifejezést teljesen leegyszerűsítettük és az eredménykifejezésben már nincsen halmazkifejezés:

```
let f x = if even x then [2*x] else []
    f _ = []
in concatMap f [1..10]
```

Mivel az *x* változó mint minta minden listaelemre illeszkedni fog, az *f* függvény második esete (amit a szabály generált oda) egyszerűen törölhető:

```
let f x = if even x then [2*x] else [] in concatMap f [1..10]
```

Az eredményből jól látszik, hogy a halmazkifejezésünk valójában az eredeti lista minden elemét egy egyelemű vagy egy üres listára képezi le, majd ezeket a listákat összefűzve az üres listák "eltűnnek" és csak a szűrt eredményeket kapjuk meg.

Az összefűzés. A listagenerátor kifejtésekor egy olyan segédfüggvényre alapoztunk, amely végigiterál a listán, alkalmaz minden elemre egy függvényt, és az eredményeket összefűzi — ez a `concatMap`. A korábban látott `map` és a `concatMap` valójában csak annyiban tér el, hogy utóbbi olyan függvényt alkalmaz a listaelemeken, amelynek eredménye is lista lesz, az eredménylistákat pedig az append (`++`) művelettel összefűzzük:

```
concatMap :: (a -> [b]) -> [a] -> [b]
concatMap f [] = []
concatMap f (x:xs) = f x ++ concatMap f xs
```

Látható, hogy a definíció kísértetiesen hasonlít a `map` megadására, az egyetlen különbség az eredmények összefűzésében van.

> **Megjegyzés**
>
> A listagenerátorok valójában két alapvető műveletet végeznek: szűrnek és (feltételesen) transzformálnak iterációban. Ezeket a műveleteket valósítják meg azok a standard függvények, amelyeket még a fejezet elején tárgyaltunk: a `map` és a `filter`. Nem meglepő tehát, hogy a fent tárgyalt halmazkifejezés kifejezhető a `map` és a `filter` kombinációjaként is:
>
> ```
> map (\x->2*x) (filter even [1..10])
> ```
>
> Ez a fajta átalakítás, tehát a `map` és `filter` függvényekkel való kifejezés, majdnem minden halmazkifejezés esetén megvalósítható, de a fordító és az értelmező ettől függetlenül a közvetlen rekurzív függvénydefinícióra és a `concatMap` használatára fejti ki őket.

A halmazkifejezésekről még néhány gondolat

Megmutattuk, hogyan lehet az intervallumkifejezéseket és a halmazkifejezéseket rekurzív függvényekkel kifejezni, lényegében megadva ezek jelentését az elemibb nyelvi konstrukciókkal — megmutattuk tehát, hogy ezek cukorkák, "kibonthatóak" egyszerűbb kifejezésekre.

A halmazkifejezések esetén a következő kifejtési szabályokat írtuk fel:

```
1. [ e | True      ] = [e]
2. [ e | q         ] = [ e | q, True ]
3. [ e | b,      Q ] = if b then [ e | Q ] else []
4. [ e | p <- l, Q ] = let f p = [ e | Q ]
                           f _ = []
                       in concatMap f l
```

Ahogyan a fenti átírási szabályokból is látszik, a szűrőfeltétel `if` kifejezésre, míg a generátor egy lokális függvényre és a `concatMap` függvény alkalmazására íródik át. Ezen nyelvi elemek szemantikája (jelentése) meghatározza a halmazkifejezések szemantikáját, és van egy-két érdekesség, amit érdemes megállapítani.

Először is, a generátorok egyesével fognak kifejtődni, szekvenciálisan balról jobbra, és egymásba ágyazott `let` kifejezéseket (egymásba ágyazott lokális függvényeket és `concatMap` alkalmazásokat) eredményeznek. A következő példában nem összezipzárazódik a két lista, hanem először az `1`-et párosítjuk össze a `[6..10]` lista elemeivel, majd a `2`-t, és így tovább:

```
ghci> [(x,y) | x <- [1..5], y <- [6..10]]
[(1,6),(1,7),(1,8),(1,9),(1,10),(2,6),(2,7),(2,8),(2,9),(2,10),
(3,6),(3,7),(3,8),(3,9),(3,10),(4,6),(4,7),(4,8),(4,9),(4,10),
(5,6),(5,7),(5,8),(5,9),(5,10)]
```

A halmazkifejezés kifejtéséből világos, hogy két `concatMap` alkalmazás van egymásba ágyazva, aminek a hatására a két listabejárás (listaiteráció) egymásba ágyazódik, nem pedig párhuzamosan, szimultán történik meg.

Hasonló megállapítást tudunk tenni a szűrőfeltételek esetén is, tehát ha a szűrő a generátor után van, akkor az nem a generátort fogja megszorítani (nem a listaiterációhoz kapcsolódik feltétel), hanem az iteráción belül, minden listaelemre történik meg a feltétel ellenőrzése és a feltételt nem kielégítő elemek kihagyása. Ez egészen világosan látható a következő példából, ami az intuíció ellenére végtelen kiértékelést eredményez:

```
ghci> [x | x <- [1..], x < 5]
[1,2,3,4
```

Megállapíthatjuk, hogy végtelen listából generálva végtelen listát készítünk, hiszen a `concatMap` a teljes (végtelen) listán végig fog iterálni, függetlenül attól, hogy a lokális függvényben egyébként ott van egy feltétel, ami minden további elemre üres listát generál.

Generátorváltozó elfedése. Az egymásba ágyazott `let` kifejezések szemantikájának van még egy érdekes vonzata: az egymásba ágyazott lokális függvények paraméterei elfedhetik egymást.

Például a következő halmazkifejezésben az eredményt nem befolyásolják az első listagenerátor elemei:

```
ghci> [(x, x) | x <- [1..5], x <- [6..10]]
[(6,6),(7,7),(8,8),(9,9),(10,10),(6,6),(7,7),(8,8),(9,9),(10,10),
(6,6),(7,7),(8,8),(9,9),(10,10),(6,6),(7,7),(8,8),(9,9),(10,10),
(6,6),(7,7),(8,8),(9,9),(10,10)]
```

Analóg példaként, leegyszerűsítve a kifejtés menetét, a következő kifejezéssel tudjuk szemléltetni az elfedést:

```
ghci> let f x = (let f x = (x, x) in f 1) in f 2
(1,1)
```

További érdekesség, hogy ugyan minden halmazkifejezésben kell legyen fejkifejezés és legalább egy generátor vagy egy szűrőfeltétel, tudunk egészen egyszerű és szélsőséges példákat is felírni:

```
ghci> [True | True]
[True]
ghci> [False | False]
[]
ghci> [1 | 1 <- [1..5]]
[1]
```

Utóbbi példa pedig azt a témát pedzegeti, hogy a generátor baloldalán valójában nem egy változó kell álljon, hanem egy minta (a `let` kifejezésben megadott lokális függvény paramétere). Ha számokat generálunk, akkor a minta nemcsak változó lehet, hanem egy konkrét szám is, ahogy a fenti példában látható, és amennyiben a generált szám megegyezik a mintában szereplő számmal, akkor fogjuk az iterációból megtartani az adott elemet.

Mivel listákra való mintákat is láttunk a nevezetes listafüggvények megadásánál, felírhatunk egy olyan halmazkifejezést, amelyben listákat generálunk és azokra írunk fel mintát:

```
ghci> [x | x <- [[1..10], [2,3,4]] ]
[[1,2,3,4,5,6,7,8,9,10],[2,3,4]]
ghci> [x | (x:xs) <- [[1..10], [2,3,4]] ]
[1,2]
ghci> [xs | (x:xs) <- [[1..10], [2,3,4]] ]
[[2,3,4,5,6,7,8,9,10],[3,4]]
ghci> [x | (y:x:xs) <- [[1..10], [2,3,4]] ]
[2,3]
ghci> [x | (y:(x:xs)) <- [[1..10], [2]] ]
[2]
ghci> [42 | [] <- [[1..10], [2]] ]
[]
ghci> [42 | [] <- [[1..10], []] ]
[42]
```

Mintaillesztést még részletesebben fogjuk tárgyalni a következő fejezetekben.

A fold függvények

Mostan láttuk a `map` és `filter` függvények definícióját, és szemléltettük a halmazkifejezésekkel való kapcsolatukat is. A halmazkifejezésekkel leírt listák tulajdonképp ezen műveletek kombinációi, sokszor csak a programozói ízlés dönti el, hogy melyiket használjuk egy-egy feladat megoldására.

A `map` és `filter` fontos, nevezetes magasabb rendű függvények, ahogy fontos és hasznos szerkezet a halmazkifejezés, de nem lehet velük minden feladatot megoldani. Vegyük példaként az összegzést — össze akarjuk adni egy számokat tartalmazó lista elemeit, a megoldó kifejezésnek legyen az értéke az elemek összege. A `map`/`filter` és a halmazkifejezések nem képesek ezt a műveletet leírni, mert a listaiteráció közben nem tudnak aggregálni. Bevezetünk tehát egy újabb nevezetes magasabb rendű függvényt, a hajtogatást (redukciót).

Példa. Rekurzív függvényt persze tudunk írni az összegzésre, hasonló stílusban, mint ahogyan a `length` függvényt írtuk; a különbség csak annyi, hogy itt minden elem esetén az elemet vesszük az összegkifejezésbe, nem pedig az 1-et:

```
sum :: [Int] -> Int
sum []     = 0
sum (x:xs) = x + sum xs
```

Észre kell venni, hogy egy ilyen függvény valójában tényleg nem csinál mást, mint előállítja az összegkifejezést (a listában lévő számok összeadásának kifejezése), amelyet későbbi lépésekben értékre lehet redukálni:

```
sum [1,2,3] =>
1 + sum [2,3] =>
1 + (2 + sum [3]) =>
1 + (2 + (3 + sum [])) =>
1 + (2 + (3 + 0)) =>
...
6
```

Érdemes észrevenni, hogy a kifejezés jobbra zárójeleződik — ezt figyelembe kell venni, ha olyan bináris művelettel végezzük az aggregációt, amely nem asszociatív. Az összegzés esetében viszont megfelelő a kifejezés, amit előállítottunk.

Mielőtt tovább haladunk, végezzünk egy általánosítási lépést, szervezzük ki a kezdeti értéket (ami itt a 0) egy függvényparaméterbe:

```
sum' :: Int -> [Int] -> Int
sum' z []     = z
sum' z (x:xs) = x + (sum' z xs)
```

Csupán annyit változtattunk, hogy felvettük a `z`-t paraméterként, és a `0`-t `z`-vel helyettesítettük. A `sum' 0 xs` kifejezés ugyanarra a kifejezésre fog kifejtődni, mint a `sum xs`:

```
sum' 0 [1,2,3] =>
1 + sum' 0 [2,3] =>
1 + (2 + sum' 0 [3]) =>
1 + (2 + (3 + sum' 0 [])) =>
1 + (2 + (3 + 0)) =>
...
6
```

Itt felmerül, hogy vajon át tudjuk-e fogalmazni ezt az összegző függvényt úgy, hogy az előálló összegkifejezés ne jobbra, hanem balra legyen zárójelezve, de továbbra is egyszerű rekurziót végezzünk a listán. Nos, a válasz igen, de a koncepciónkat teljesen meg kell változtatni: azt a függvényparamétert, amiben a kezdeti összeget "tároltuk", most arra kell használnunk, hogy a bejárt listaelemek összegét (összegkifejezését) írja le:

```
sum'' :: Int -> [Int] -> Int
sum'' z []     = z
sum'' z (x:xs) = sum'' (z+x) xs
```

Vegyük észre, hogy a függvény első esetén (ami az üres listához tartozik) nem változtattunk, és továbbra is a listán végzünk egyszerű rekurziót (mindig a maradékrészre alkalmazzuk rekurzívan a függvényt), az összeadás azonban a rekurzív alkalmazáson kívülről az alkalmazáson belülre került, így a lista balról jobbra való bejárása esetén az előálló kifejezés is balról jobbra lesz zárójelezve:

```
sum'' 0 [1,2,3] =>
sum'' (0 + 1) [2,3] =>
sum'' ((0 + 1) + 2) [3] =>
sum'' (((0 + 1) + 2) + 3) [] =>
(((0 + 1) + 2) + 3) =>
...
6
```

Érdemes megfigyelni, hogy az első változatunk (ami jobbra zárójelez) definíciójában a jobboldali kifejezés fő művelete az összeadás, míg ez utóbbi változat (ami balra zárójelez) közvetlenül a rekurzív függvényalkalmazásra íródik át (minden sor a `sum''` alkalmazásával kezdődik egészen a rekurzió lezárásáig). Ebből azonnal látszik az is, hogy míg az első változatnál minden iterációban (minden átírással) megjelenik az eredménykifejezésben egy összeadás, addig a második a második változat az akkumulátor változóban kezdi felhalmozni az összeadás műveletek alkalmazásait. Ennek fontos következménye, hogy a második változatban csak akkor lehet az aggregáló műveleteket elkezdeni elvégezni, amikor már a teljes listát bejártuk, tehát végtelen lista esetén biztosan végtelen lesz a kiértékelés is. Másrészről viszont a második változat végrekurzív (tail-recursive), amely lehetővé teszi a rekurzív függvényalkalmazások optimalizációját (hiszen az aktivációs rekord újrafelhasználható), tehát véges listák esetén tárban hatékonyabb tud lenni az első változatnál.

A különbség demonstrálható olyan bináris művelettel, amelynek definíciója bizonyos esetekben nem hivatkozik a jobboldali operandusra, tehát rövidzáras (short-circuit). Ilyen például a logikai "és":

```
ghci> False && length [1..] > 0
False
```

A fenti példában a végtelen lista hosszára vonatkozó állítás eldöntése végtelen sokáig tartana, de a konjunkció baloldalán lévő hamis érték miatt a jobboldal nem releváns, tehát a lusta kiértékelés miatt az interpreter el sem kezdi redukálni. Ez azért érdekes gondolat, mert ezek szerint egy végtelen konjunkció kiértékelése lehet véges:

```
ghci> False && True && False && True && ...
False
```

Ezt a rövidzár kiértékelést ki tudja használni a jobbra zárójelező aggregáló függvény, de nem tudja kihasználni a balra zárójelező változat. Ezt tudjuk demonstrálni egy olyan "összegzéssel", amely a számokat nem összeadja, hanem azt összegzi (dönti el), hogy mindegyik páros-e:

```
alleven :: [Int] -> Bool
alleven []     = True
alleven (x:xs) = even x && alleven xs
```

A kezdeti értéket az összegzés mintájára kiemelve:

```
alleven' z []     = z
alleven' z (x:xs) = even x && alleven' z xs
```

Látható, hogy az `even x && ...` műveletek szépen felsorolásra kerülnek, ám az összeadás művelettel ellentétben a konjunkció jobboldalát nem feltétlenül kell kiértékelni az érték meghatározásához:

```
alleven' True [0..] =>
even 0 && alleven' True [1..] =>
True && alleven' True [1..] =>
alleven' True [1..] =>
even 1 && alleven' True [2..] =>
False && alleven' True [2..] =>
False
```

Ezzel szemben a balra zárójelező változat (mivel végrekurzív) ezt a fajta "rövidítést" nem tudja megtenni, hiszen mindig magára íródik át a függvény:

```
alleven'' :: Bool -> [Int] -> Bool
alleven'' z []     = z
alleven'' z (x:xs) = alleven'' (z && even x) xs
```

És ennek hatására végtelen listára alkalmazva mindig végtelen sok átírást fog eredményezni:

```
alleven'' True [0..] =>
alleven'' (True && even 0) [1..] =>
alleven'' ((True && even 0) && even 1) [2..] =>
alleven'' (((True && even 0) && even 1) && even 2) [3..] =>
...
```

foldr és foldl

A `sum` és az `alleven` függvények valójában ugyanazt a rekurziósémát (mintát) követik, mind a balra zárójelező, mind a jobbra zárójelező változatban. Ez az általános logika különválasztható a konkrét aggregáló művelettől és kezdeti értéktől, amellyel egy nevezetes magasabb rendű műveletet kapunk: a "fold" függvényt. Ahogyan az összegzésnek, úgy a foldnak (hajtogatásnak) is van jobbra és balra zárójelező változata: `foldr` és `foldl`.

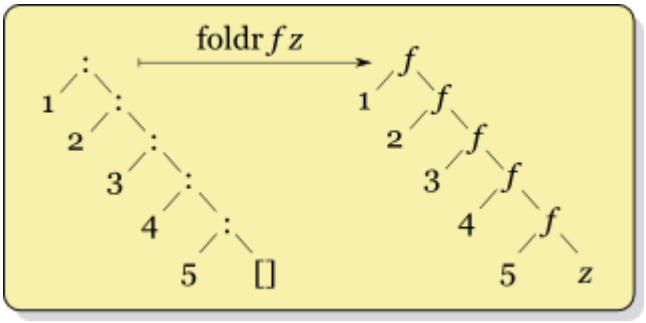
A kezdőérték és a bináris művelet (függvényparaméter) kiemelése után a következő általánosan alkalmazható definíciókat kapjuk:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z [] = z
foldr f z (x:xs) = x `f` foldr f z xs
```

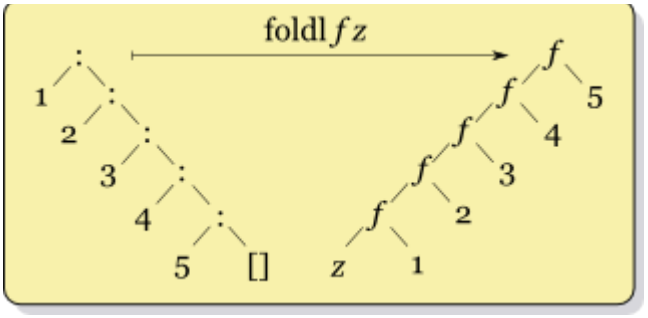
```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f z [] = z
foldl f z (x:xs) = foldl f (z `f` x) xs
```

A `sum` és az `alleven` is közvetlenül visszavezethető mindkét hajtogató függvényre.

A `foldr` a "jobbra hajtogatás", a `foldl` pedig a "balra" hajtogatás magasabb rendű függvénye. Szemléletesen úgy lehet elképzelni, hogy a `foldr` a `(:)` listakonstruktort helyettesíti a paramétermű velettel és az üres lista konstruktort a kezdeti akkumulátor értékkel (tehát a kezdeti érték a lista "végére" kerül):



A `foldl` esetében hasonló történik, csak közben a fa dőlésének irányát megfordítjuk (ennek hatására lesz balra zárójelezve az eredménykifejezés, és a kezdeti érték így a lista "elejére" kerül):



Érdekesség, hogy a `foldl` segítségével például triviálisan felírható a tetszőleges listát megfordító függvény (`reverse`).

foldl'

Van azonban a végrekurzív változatnak egy komoly előnye, amelyet véges listák esetén ki is tudunk használni. Vegyük a korábban látott `sum''` függvény kiértékelését:

```
sum'' 0 [1,2,3] =>
sum'' (0 + 1) [2,3] =>
sum'' ((0 + 1) + 2) [3] =>
sum'' (((0 + 1) + 2) + 3) [] =>
(((0 + 1) + 2) + 3) =>
...
6
```

Látható, hogy a kiértékelés során teljes, egyszerűsíthető összegkifejezések állnak össze a lista prefixeihez. Ezeket a részösszegeket valójában nem kell kifejezésként "cipelnünk" a rekurzív alkalmazásokban, hanem rávehető az értelmező, hogy mohón értékelje ki a köztes összegeket, ilyen egyszerűsítést eredményezve:

```
sum''' 0 [1,2,3] =>
sum''' (0 + 1) [2,3] =>
sum''' 1 [2,3] =>
sum''' (1 + 2) [3] =>
sum''' 3 [3] =>
sum''' (3 + 3) [] =>
3 + 3 =>
6
```

Utóbbi megközelítésnek lényegesen kisebb a memóriaigénye, hiszen nagy listák esetén nem kell nagy méretű kifejezést előállítani.

> **Megjegyzés**
>
> A mohó `foldl` függvényt a standard könyvtárban a `foldl'` implementálja. Bővebb magyarázat található a fold variánsokról a haskell.org wiki oldalain.

Visszavezetések

A redukcióra az egyik példánk számokat tartalmazó lista elemeinek összegzése volt (`sum`), a másik példa pedig eldöntötte, hogy egy számokat tartalmazó lista minden eleme páros-e (`alleven`). Kétféleképp is megfogalmaztuk az aggregálás műveletét, az egyik esetben jobbról balra építve az összegkifejezést, a másik esetben pedig balról jobbra (végrekurzívan megfogalmazva). Ezeket a műveleteket viszonylag magától értetődő módon vissza tudjuk vezetni az imént bevezetett `foldr` és `foldl` általános hajtogató függvényekre.

A `sum'` és `alleven'` függvények hasonlóképp vannak felírva és hasonlóképp működnek: két esettel adjuk meg őket, az üres lista esetén a kezdőértéket (alapértéket) eredményezik, a nemüres lista esetén pedig alkalmazzák a feldolgozó műveletet, amely az aktuális elemet összekapcsolja a lista maradékának redukáltjával. Ezekből a függvényekből az aggregáló művelet is kiáltalánosítható:

```
sum' :: Int  -> [Int] -> Int
(+) :: Int -> Int -> Int
alleven' :: Bool -> [Integer] -> Bool
even :: Integer -> Bool
(&&) :: Bool -> Bool -> Bool
(&&) . even :: Integer -> Bool -> Bool
```

És ezek után megkapjuk a nevezetes "hajtogató" magasabb rendű függvényeket:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z [] = z
foldr f z (x:xs) = x `f` (foldr f z xs)

foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f z [] = z
foldl f z (x:xs) = foldl f (z `f` x) xs
```

A foldr a redukáló művelet mentén általánosítja a `sum'` és `alleven'` függvényeket, míg a `foldl` pedig a `sum''` és `alleven''` függvényeket. Valóban, a `sum` és az `alleven` is visszavezethető:

```
sum = foldr (+) 0
alleven = foldr (\x p -> even x && p) True
-- vagy kompozícióval: alleven = foldr ((&&) . even) True
```

Nevezetes visszavezetések

A korábban mutatott ábrán jól látszik, a (listákra értelmezett) `foldr` művelet valójában úgy is tekinthető, mint az üres lista konstruktor z-re, illetve a nemüres lista konstruktor f-re való cseréje a listaszerkezetben. Ezt elnézve nem meglepő, hogy a listákra értelmezett `foldr` egyik legegyszerűbb specializációja, amikor az üres listát üres listára, a nemüres lista konstruktort pedig a nemüres lista konstruktorra cseréljük, ezzel megadva a listákon vett identitásfüggvényt:

```
id' :: [a] -> [a]
id' = foldr (:) []
```

A hajtogatást valóban redukcióra használva az egyik legegyszerűbb visszavezetés a lista hosszának kiszámítása, ami az üres listát a `0`-ra, a nemüres lista konstruktorát pedig a `(+1)` műveletre képezi:

```
length' :: [a] -> Int
length' = foldr (\x n -> n + 1) 0
-- vagy egyszerűbben: foldr (\_ -> (+1)) 0
```

Ami viszont már érdekesebb, hogy a `foldr` valóban több mint csak redukció, a hajtogatásra visszavezethető például a `map` (elemenkénti műveletalkalmazás a lista elemein) is:

```
map' :: (a -> b) -> [a] -> [b]
map' f = foldr (\x xs -> f x : xs) []
-- vagy kompozícióval: foldr ((:) . f) []
```

És visszavezethető rá a `filter` is:

```
filter' :: (a -> Bool) -> [a] -> [a]
filter' p = foldr (\x xs -> if p x then x:xs else xs) []
```

Vegyük észre azt is, hogy a `foldl` viszont úgy "cseréli le" a lista konstruktorait a kezdőértékre és a műveletre, hogy a lista elemeit fordítva fűzi össze. Ennek következménye, hogy a `foldl`-t az identitásfüggvényhez hasonlóan paraméterezve (annyi különbséggel, hogy a `(:)` paramétereit megcseréljük) a listát megfordító műveletet kapjuk:

```
reverse' :: [a] -> [a]
reverse' = foldl (flip (:)) []
```

Ennek természetesen végtelen lista esetén nincs értelme, de véges listákra úgy fog kiértékelődni, ahogyan elvárjuk tőle a korábbi definíciók alapján.

> **Megjegyzés**
>
> A flip függvény nem csinál mást, mint megcseréli a bináris művelet jobb és bal argumentumát:
>
> ```
> ghci> :t flip
> flip :: (a -> b -> c) -> (b -> a -> c)
> ghci> (:) 1 [2,3]
> [1,2,3]
> ghci> (flip (:)) [2,3] 1
> [1,2,3]
> ```

foldr1 és foldl1

A hajtogatás műveleteknek létezik valódi redukáló változata is a szabványos könyvtárban, amelyeknek nem kell kezdeti értéket megadnunk:

```
ghci> :t GHC.List.foldr1
GHC.List.foldr1 :: (a -> a -> a) -> [a] -> a
ghci> :t GHC.List.foldl1
GHC.List.foldl1 :: (a -> a -> a) -> [a] -> a
```

Ezekre is vissza tudjuk vezetni az összegzést:

```
ghci> foldr1 (+) [1..10]
55
ghci> foldl1 (+) [1..10]
55
```

A fold ezen változatai a redukálást úgy végzik, hogy a hajtogatásban nem szerepel kezdőérték, helyette a lista első (illetve utolsó) elemét használják. Ennek nyilvánvaló következménye, hogy ezeket a változatokat (az általános hajtogatással ellentétben) üres listára nem lehet alkalmazni, és azt is érdemes észrevenni, hogy kevésbé általános a típusuk.

> **Megjegyzés**
>
> Ahogyan a jegyzet elején említettük, az ezen jegyzetben tárgyalt nyelvi elemek mind beágyazhatóak a lambda-kalkulusba, ez így van az alaptípusokkal és a listákkal is: a listáknak is van úgynevezett *Church-encoding* megvalósításuk, amelyben tisztán függvényekkel írjuk le őket.
>
> Ennél kevésbé komplex azonban a listák tényleges megvalósítása. Ahogyan látni fogjuk, a listák algebrai adattípusként vannak megadva, ahol két alapvető típuskonstrukcióra támaszkodunk: az összegre és a szorzatra. A következő fejezet részletesen tárgyalja, hogyan adunk meg egyszerű összetett adattípusokat, mik a nevezetes esetek, és közöttük tárgyaljuk a listákat is.

Algebrai adattípusok

Az előző fejezetben részletesen tárgyaltuk a homogén, nem változtatható (*immutable*) listákat, mint a (tisztán) funkcionális programozási nyelvek alapvető összetett típusát. Valóban, sokféle és érdekes nyelvi elemet tudtunk lefedni csupán egész számok, logikai értékek és listák használatával, de ezeken túl más, a funkcionális paradigmára jellemző (összetett) típusok is elérhetőek Haskellben. Sőt, saját összetett típusok megadására is van lehetőség. Ebben a fejezetben részletesebben tárgyaljuk a nevezetes típusokat, miközben megmutatjuk a szokásos algebrai típusdefinícióikat és a rajtuk értelmezett műveleteket.

> **Megjegyzés**
>
> Típusszinonimákat a `type` kulcsszó segítségével lehet bevezetni, ám ezek csak *típusneveket* adnak meg (például a `String` a `[Char]` szinonimája). Ezekkel olvashatóbbá tehetjük függvények típusleírását, de új típust nem tudunk velük definiálni. A jegyzet a `type` segítségével megadott típusnevekkel nem foglalkozik részletesen, további információ a nyelvi specifikáció vonatkozó fejezetében fellelhető.

Az algebrai adattípus fogalma

Felsorolási és összetett típusokat algebrai adattípusokkal adunk meg. Az algebrai adattípus leegyszerűsítve nem más, mint szorzattípusok összegtípusa, avagy Descartes-szorzatok címkézett uniója. Egyelőre képzeljük el úgy, mint alternatív rekordok, amelyekben tetszőleges típusú adatokat "tárolhatunk", és ahol minden alternatívának egyedi neve van. Az előbb említett egyedi neveket *adatkonstruktornak* fogjuk hívni, az egyes alternatívákat *variánsnak* nevezzük, a variáns szorzattípusában "tárolt" adatokat, amelyek az adatkonstruktorok argumentumai, *mezőknek* vagy adattagoknak hívjuk.

> **Érdekesség**
>
> Az adatkonstruktorok injektív függvények, variánsok típusértékei mindig diszjunkt halmazok.

> **Érdekesség**

> Habár az ilyen típusokat is be lehet ágyazni a lambda kalkulusba, a gyakorlatban a modern funkcionális nyelvek magnyelvét inkább olyan kalkulusra építik, amelyben lehetőség van algebrai adattípusokat megadni, azok értékeit megkonstruálni, és az értékekre mintát illeszteni, így nincs szükség bonyolult beágyazásra csupán függvények használatával; ahogyan már említettük, a Haskell magnyelve a System FC kalkulusra épül.

Amikor algebrai adattípust adunk meg Haskellben, a típus definícióját a *data* kulcsszó jelöli, a variánsokat pedig függőleges vonallal (unió, alternatíva) választjuk el egymástól, a mezők típusait pedig felsoroljuk a konstruktor neve után.

Felsorolás

Az algebrai adattípusok legegyszerűbb esete a felsorolási típus. Az ilyen algebrai adattípusban nincsen szorzattípus, a konstruktoroknak nincsenek argumentumai; tulajdonképp "üres" variánsokkal adjuk meg, amelyek egymástól megkülönböztethetőek, "eseteket" írnak le.

Egy korábban már bemutatott típus épp felsorolással adható meg: a Haskell logikai típusának elemeit nagybetűvel kellett felírni (`True` és `False` jelölték a típus elemeit), aminek az volt az oka, hogy a `Bool` típus valójában algebrai adattípusként van megadva, és az *adatkonstruktorai* a nyelv szabályai szerint nagybetűvel kell kezdődjenek. A logikai típus definíciója így írható fel:

```
data Bool = False | True
```

A felsorolási típusnak annyi eleme van, ahány konstruktora; következésképp a `Bool` típusnak két eleme van.

> **Megjegyzés**
>
> A konstruktorok sorrendje a típusértékhalmaz megadása szempontjából nem releváns, de lesznek olyan származtatott függvények, amelyek a konstruktorok sorrendjét fogják például rendezéshez felhasználni.

> **Megjegyzés**
>
> Léteznek egyelemű felsorolások is, amelyek közül kiemeljük a *unit* típust. A következőképp adható meg:

```
data () = ()
```

Mivel ennek a típusnak csak egy konstruktora van, pontosan egy eleme is van.

Rekord

Amíg a felsorolási típusnál több, paraméter nélküli adatkonstruktort adunk meg (tehát csak összeg van, szorzat nincs), addig a másik véglet az, amikor nincs összeg (unió), csupán szorzat van. Itt egyetlen eset (variáns) van a típusban, annak viszont vannak paraméterei, és így "nevesített" rekordként tárolja a paraméterekben megadott értékeket.

Megadható például egy síkbeli pont típus, amelynek két `Int` adattagja van:

```
data Point = Point Int Int
```

Ennek a típusnak eleme minden olyan érték, amelyet a `Point` konstruktorral alkotunk meg. Az értékek megadása lényegében a konstruktor alkalmazása a megfelelő mezőértékekkel. A `Point 1 2` egy `Point` típusú érték, amelyben az `1` és `2` értékek kerülnek tárolásra.

Miután megadtuk a `Point` típusdefiníciót, már típusváltozók helyén is szerepelhet, így például a lista típuskonstruktor is paraméterezhető a `Point` típussal, pontok listáját eredményezve:

```
[Point 1 2, Point 3 4]
```

Fontos

A fenti példában a `Point` a típusnak (típuskonstruktornak) és az adatkonstruktornak is a neve. Természetesen ezeknek nem kell megegyezni, de fontos tudni, hogy megegyezhetnek. Hasonlót láttunk a unit típusnál, ahol a típuskonstruktor és az adatkonstruktor neve is `()`.

Megjegyzés

Az adatkonstruktor egy függvényszimbólum, következésképp részlegesen is alkalmazható. Felírható tehát például egy "részlegesen kitöltött" adatkonstruktor:

```
Point 42 :: Int -> Point Int Int
```

> Itt a `Point 42` egy olyan függvényt ír le, amelyet egy `Int` értékre alkalmazva (amely a `Point` második argumentuma), megkapjuk a `Point` típusú értéket.

Mintaillesztés algebrai adatokon

A `Point` lényegében két `Int` típusú mezővel megadott rekord. Hogyan férhetünk hozzá a benne tárolt értékekhez? Más nyelvekben megszokott "setter" és "getter" függvények itt nincsenek, a megoldás itt is a mintaillesztés lesz, ahogyan a listák esetében már láthattuk.

Minták általában

A `Point` név egyszerre konstruktor (függvény) és destruktor (minta) is. Ha meg akarunk adni egy pontot, akkor a konstruktorfüggvényt alkalmazzuk, és amikor ki akarjuk nyerni a rekord mezőit, akkor a `Point` címkével írunk fel egy mintát, illesztve a konstruktor paramétereire. A `Point 1 2` tehát egy érték, amelyből a mezőket (`1` és `2`) nem ún. "getter" függvényekkel érjük el, hanem az adat szerkezetének megfelelő minta illesztésével.

A minták, túl a primitív vagy beépített típusok értékein, az adott típus valamely konstruktorának megnevezésével, és az adattagokra vonatkozó minták megadásával írhatók fel. Amennyiben valamely adattag értékét szeretnénk "kiolvasni", akkor egy változót (változónév mintát, argumentum mintát) kell illeszteni rá, amelynek hatására az adattag értéke hozzárendelődik a változóhoz. A `Point x y` minta illesztése a `Point 1 2` értékre például azt eredményezi, hogy az `x` az `1` értékhez, az `y` pedig a `2` értékhez kerül kötésre.

> **Megjegyzés**
>
> Ha egy adattag értéke nem releváns a mintában, akkor a 'wildcard' mintát használjuk, amelynek `_` a jelölése. A 'wildcard' és a *változó* minták mindig illeszkednek (irrefutable pattern), a konstruktorral készített minták viszont csak akkor, ha a minta konstruktora megegyezik az illesztett érték konstruktorával.

Mintaillesztéssel rendkívül egyszerűen lekérdezhető a `Point` első eleme, amit függvénybe is foglalhatunk:

```
firstElement :: Point -> Int
firstElement (Point x _) = x
```

A mintában a konstruktor neve és paraméterei zárójelbe vannak foglalva. Minden paraméterre kell mintát megadni, és a mezők pozíció szerint kerülnek elérésre (részleges minták tehát nincsenek). A függvény fejében történő mintaillesztés a korábban a listáknál látott módon `case` esetszétválasztásra fejtődik ki. A mintáknak lineárisnak kell lennie, tehát ugyanaz a változó nem szerepelhet többször egy mintában; a `(Point x x)` minta például hibás, a Haskell értelmező (és fordító) visszautasítja.

> **Megjegyzés**
>
> A Haskell nyelv támogat úgynevezett rekord szintaxist is, ami lehetővé teszi, hogy a rekord mezőit elnevezzük és a nevükkel hivatkozzunk rájuk a mintaillesztésben is, ezzel kiküszöbölve a pozíció szerinti hivatkozást. Ennek a nyelvi elemnek a megítélése igencsak vegyes, vannak, akik szerint nem igazán illik a Haskell nyelvbe, de a Haskell 2010 szabvány tartalmazza, és a nyelvi specifikáció vonatkozó fejezete részletesen tárgyalja a rekord szintaxis különböző használati módjait.

Vegyük észre, hogy a `(Point x _)` valójában tekinthető egymásba ágyazott (*nested*) mintának, hiszen a fő (külső) minta a `Point` konstruktorra (variánsra) illeszkedik, és az abban lévő mezőkre illeszkedik az `x` és a `_`, amik szintén minták (bizonyos definíciók csak azokat a mintákat tekintik egymásba ágyazottnak, amelyekben a belső minták nem triviálisak). A továbbiakban sok-sok példán keresztül fogjuk látni, hogy milyen sokféle mintát lehet felírni Haskellben, de a minták pontos definíciójával és a mintaillesztés algoritmusával nem foglalkozunk, azokat részletesen tárgyalja a nyelvi specifikáció vonatkozó fejezete.

Bool függvények megadása mintaillesztéssel

Mintaillesztéssel természetesen nemcsak rekordmezőket tudunk kiolvasni, hanem például a felsorolási típusok eseteit is meg tudjuk különböztetni. A `Bool` esetében mintaillesztéssel adhatjuk meg az alapvető logikai műveleteket:

```
not :: Bool -> Bool
not False = True
not True = False
```

Mivel a típusnak csak két típusértéke van, a második (másodlagos) esetben akár olyan mintát is használhatunk, ami megengedőbb (több értékre, ebben az esetben mindkét értékre illeszkedik). A függvény jelentése nem változik meg, mivel *az eseteket mindig fentről lefelé próbálja alkalmazni a kiértékelés*:

```
not False = True
not x = False
```

> **Megjegyzés**
>
> Nem meglepő, hogy amennyiben az első eset mintáját változtatjuk változóra (ami minden értékre illeszkedik), akkor a függvény jelentése megváltozik, hiszen mindkét logikai értékhez a `True` értéket fogja rendelni:
>
> ```
> not x = True
> not True = False
> ```

> **Megjegyzés**
>
> A `not` függvény kétszeri alkalmazásával, dupla negációt végezve visszakapjuk az eredeti logikai értéket.
>
> ```
> ghci> not (not True)
> True
> ```
>
> Vegyük viszont észre, hogy itt nem hagyható el a zárójelezés az alkalmazás művelet balra zárójeleződése miatt: a `not not True` kifejezés a `not` függvényt alkalmazná a `not` függvényre, majd az eredményt a `True` értékre, ami típushibát fog eredményezni.

Hasonlóan megadhatjuk a konjunkció függvényét is:

```
andBool :: Bool -> Bool -> Bool
andBool False x = False
andBool True  x = x
```

A mintaillesztést itt az első paraméteren végezzük el, annak a kiértékelése szükséges ahhoz, hogy kiderüljön, melyik eset (ág) kerül alkalmazásra. Ha az első paraméter hamis, akkor a hamis értékre egyszerűsödik a függvényalkalmazás, ha pedig igaz, akkor a második paraméterre.

A nyelv lusta kiértékelésének köszönhetően a fenti definíció rövidzárasan kerül értelmezésre, tehát amennyiben az első paraméter hamis, a másodikat már ki sem értékeli az értelmező (tehát akár végtelen sok lépésből álló számítás is lehetne, hiszen nem kerül kiértékelésre).

> **Megjegyzés**
>
> Ahogy már korábban említettük, a fenti példában az első `x` változóminta helyére az _ minta is írható, ami a változóhoz hasonlóan az összes típusértékre illeszkedik. A későbbiekben használni fogjuk a változómintát és a 'wildcard' mintát is az ilyen esetekben, és az olvasóra bízzuk, hogy melyiket preferálja.
>
> ```
> andBool False _ = False
> andBool True  x = x
> ```

Ha nem akarjuk, hogy rövidzár működésű legyen a művelet, akkor felírhatunk olyan mintaillesztést, ami kikényszeríti mindkét operandus kiértékelését az eset kiválasztása előtt.

```
andBool True True = True
andBool _ _ = False
```

Itt az első mintaillesztés elvégzéséhez (az első ág illeszkedésének eldöntéséhez) mindkét argumentumot ki kell értékelni, mert csak ezután dönthető el, hogy mindkettő illeszkedik-e a `True` mintára. A minták megfelelő megválasztásával tehát befolyásolható, hogy mely argumentumokat (és milyen mértékben) értékeli ki az értelmező a függvény alkalmazása előtt.

Az `andBool` függvény az elvárások szerint viselkedik mindkét definíció mellett:

```
ghci> andBool True True
True
ghci> andBool True False
False
ghci> andBool False True
False
ghci> :t andBool False
andBool False :: Bool -> Bool
```

Az `andBool False` kifejezés az `andBool` függvény részleges alkalmazását mutatja be, ahol az egyik operandust megadtuk (és rögzítettük), így a kifejezés egy olyan függvényre értékelődik ki, amely már `Bool` típusú értékekhez (amelyek a konjunkció jobboldalát adják meg) `Bool` típusú értéket rendel.

Értékek megjelenítése

Az előző kódblokkokban csaltunk kicsit, ugyanis az algebrai adattípusaink elemeit alapértelmezetten csak megkonstruálni lehet, megjeleníteni azonban nem. A típust megadva már értelmezhető a definíció, tudunk függvényeket definiálni rajta, de ahhoz, hogy ki tudjuk írni a képernyőre ezeket az elemeket, meg kell adnunk, hogyan képezhetőek a `String` típusra (más nyelvekben ezt neveznék a típus "to string" metódusának):

```
showBool :: Bool -> String
showBool False = "hamis"
showBool True  = "igaz"
```

A típusértékeket megjelenítő függvényt automatikusan is generálhatjuk, ahol az alapértelmezett megjelenítés egyszerűen szöveggé alakítja a konstruktorok neveit és a mezőket. Ehhez Haskellben a `deriving` kulcsszót tudjuk használni. A `Bool` típust így kiegészítve már megjeleníthetővé válnak a típus elemei.

```
data Bool = False | True deriving Show
```

A `Show` valójában egy nevezetes típusosztály, típusok egy csoportja, családja. Azok a típusok tartoznak a `Show` osztályhoz, amelyeket meg lehet jeleníteni (szöveggé lehet alakítani). A `deriving Show` hatására a `show` (ad-hoc polimorf) függvény kerül túlterhelésre a típusunkhoz, alapértelmezett implementációval. A típusunk osztályba való bevitelét, példányosítását, saját szöveggé alakító implementációval is elvégezhetjük, erről később fogunk részletesebben értekezni.

Nevezetes típusok

A felsorolási és rekord típusok "kombinálásával" kapjuk azokat a típusokat, amelyek több variánssal, és azokon belül adattagokkal vannak megadva. Ezekkel már valóban tudunk összetett adatokat reprezentálni, és felhasználni őket komplexebb problémák típushelyes megoldásának felírására. Mielőtt saját típusokat definiálnánk (például bináris fák, adatrekordok, programozási

feladatok állapotának leírására), sorra vesszük a paradigma nevezetes típusait.

Maybe

A `Bool` kételemű felsorolási típus és a `Point` egyszerű rekord típus után a következő lépés a típusunk bonyolításában az, ha szorzat és összeg konstrukció is felhasználásra kerül egyazon típus megadásában. Erre lehet példa a következő típus, ami opcionális egészértéket tud reprezentálni:

```
data MaybeInt = NoInt | JustInt Int deriving Show
```

Ezt felhasználhatjuk például úgy, hogy amennyiben egy `Int` típusú értékeket tartalmazó lista fejeleméhez akarunk hozzáférni, az üres listán vett sikertelen művelet értéke lehet a `NoInt`, a sikeres pedig a `JustInt`, megvalósítva ezzel egyfajta egyszerű hibakezelést/kivételkezelést:

```
maybeIntHead :: [Int] -> MaybeInt
maybeIntHead []     = NoInt
maybeIntHead (x:_) = JustInt x
```

Ekkor:

```
ghci> maybeIntHead []
NoInt
ghci> maybeIntHead [1..10]
JustInt 1
```

Ez az opcionális típus és "extremális elem" megközelítés természetesen általánosítható `Int` típusról tetszőleges típusra, típusparaméter használatával. Ekkor megkapjuk a nevezetes `Maybe` típust, amelyet széles körben használnak tiszta nyelvekben:

```
data Maybe a = Nothing | Just a deriving Show
```

Az általánosított, immáron parametrikus típussal felírhatjuk a polimorf fejelem függvényt, amely bármilyen típusú elemeket tartalmazó listára értelmezhető.

```
maybeHead :: [a] -> Maybe a
maybeHead []    = Nothing
maybeHead (x:_) = Just x
```

Hasonlóan az előző példához, üres listához a `Nothing` értéket rendeljük, egyébként pedig "becsomagoljuk" a fejelemet a `Just` konstruktorral (amelyből majd mintaillesztéssel lehet "kicsomagolni").

```
ghci> maybeHead []
Nothing
ghci> maybeHead [1..10]
Just 1
```

Érdemes észrevenni, hogy a `maybeHead` függvényhez hasonlóan maga a `Just` is egy polimorf (adatkonstruktor) függvény:

```
ghci> :t Just
Just :: a -> Maybe a
ghci> :t Just True
Just True :: Maybe Bool
```

Megjegyzés

A `Maybe` nem adatkonstruktor, nincs típusa:

```
ghci> :t Maybe
<interactive>:1:1: error:
• Illegal term-level use of the type constructor ‘Maybe’
```

A `Maybe` egy típuskonstruktor, aminek fajtája (kind-ja) van. A fajta megadja, hogy hány típustól függ a kérdéses típus:

```
ghci> :k Maybe
Maybe :: * -> *
ghci> :k Bool
Bool :: *
```

A `Maybe` tehát egy egyparaméteres típuskonstruktor, amelyet egy típusra alkalmazva egy újabb típust kapunk.

A `Maybe`-t nemcsak opcionális visszatérési értékként használhatjuk, hanem természetesen függvényparaméter is lehet. Megadhatunk például egy olyan függvényt, amely a `Maybe`-n belül megduplázza egy szám értékét, amennyiben van érték, és üresen hagyja, ha nincs érték (pontosabban `Nothing` az érték):

```
doubleMaybe :: Maybe Int -> Maybe Int
doubleMaybe Nothing = Nothing
doubleMaybe (Just x) = Just (2*x)
```

Fontos észrevenni, hogy ez a függvény a `Nothing`-ból `Nothing`-ot, `Just`-ból pedig `Just`-ot csinál. Kicsomagol, elvégzi a műveletet, visszacsomagol. Épp, mint a listán megadott duplázás művelet, itt is van üres és nemüres eset, csak itt a nemüres esetben egyetlen értéket kell csak módosítani. Emlékeztetőül a `map` mintájára felírt `doubleList` így nézett ki:

```
doubleList [] = []
doubleList (x:xs) = (:) (2*x) (doubleList xs)
```

Továbbá a `doubleList` általánosítása, a `map` nevezetes függvény így nézett ki az `f` függvénnyel paraméterezve:

```
map f [] = []
map f (x:xs) = (f x) : (map f xs)
```

Nem nehéz elképzelni, hogy a ennek mintájára lehetne egy `mapMaybe` függvényt írni, ami akkor alkalmazza az `f` függvényt, ha `Just` értékkel van dolgunk:

```
mapMaybe f Nothing = Nothing
mapMaybe f (Just x) = Just (f x)
```

Ezt a gondolatot tovább fogjuk vinni és tetszőleges összetett típusra értelmezni fogjuk a "mapping" műveletét (a "map-elhető" típusok csoportját úgy hívjuk majd, hogy `Functor`).

Either

A `Maybe` típustól eggyel tovább lépve a típus összetettségében megadjuk a másik egyszerű, nevezetes opciót kifejező algebrai adattípust, az `Either` típust (típuskonstruktort). Itt már mindkét variáns paraméterezve van egy-egy értékkel, és itt már két különböző típusú érték lehet a mezőkben. A klasszikus definíció így szól:

```
data Either a b = Left a | Right b deriving Show
```

Ez is sokmindenre használható, többek között hibakezelésre: itt is az egyik eset sikeres, a másik a sikertelen, de utóbbi mellé információ csatolható a "kivétel" okáról. Az egyik legegyszerűbb eset, ha a `String` típust használjuk a hibás eset leírására; vegyük az előző fejelem lekérdező műveletet példaként:

```
getHead :: [a] -> Either a String
getHead []    = Right "cannot take the head of an empty list"
getHead (x:_) = Left x
```

A `Maybe` esetében tárgyalt hibakezelési és hibapropagációs megoldás itt is hasonlóan tud működni (sikertelen műveletek eredményén a további műveletek "automatikusan" sikertelenek), és ennek általános kezelését a következő fejezetben be is fogjuk mutatni a típusosztályok tárgyalásakor.

Lista

A listákkal már egy egész fejezeten keresztül foglalkoztunk, de "beépítettként" kezeltük őket, nem adtuk meg a definíciójukat. Most, hogy már tudjuk, hogyan adjunk meg összetett típusokat, a listákat is felírhatjuk a korábban látott `[]` és `(:)` (üres és nemüres lista) konstruktorok felsorolásával. A listák típusa azonban lényegesen bonyolultabb, mint a `Maybe`, `Either` vagy `Point` típusok. Habár itt is "csak" két variáns és két adattag van, a nemüres lista konstruktora lényegileg különbözik az eddigiektől: rekurzív.

A lista típusa tehát a lista típusával van definiálva; ezt a Haskell támogatja, tehát ez nem kell beépített legyen, mi magunk is meg tudunk adni egy saját, a standard listát utánzó típust:

```
data List a = Nil | Cons a (List a)
```

Itt azt kell megérteni, hogy minden nemüres lista tartalmaz egy újabb listát, balról jobbra láncolva. Minden véges lista legbelső listája a `Nil`. Azok a listák, amelyekben nincs `Nil`, végtelennek tekinthetőek; utóbbiakat csak lustán tudjuk kiértékelni.

> **Megjegyzés**
>
> A korábbi fejezetben látott lista típus (`[a]`) izomorf, megfeleltethető a `List a` típussal, tulajdonképp csak a jelölés más. A `[]` konstruktornak a `Nil` felel meg, a `(:)` konstruktornak pedig a `Cons` . A Haskell nyelv azért vezeti be a jelöléseket, hogy könnyebben olvashatóak legyenek a listák literáljai és mintái.

Tuple (rendezett n-es)

Van egy további nevezetes összetett típus, amelyet majdnem minden modern programozási nyelv támogat: a rendezett n-es (tuple). Az n-esek segítségével párokba, hármasokba, négyesekbe szervezhetünk összetartozó adatokat, anélkül, hogy rekord típust kellene megadnunk hozzájuk (lényegében anonim rekordként tudjuk használni). Egész számok párja például használható a `Point` típus helyett:

```
ghci> :t (1::Int,2::Int)
(1::Int,2::Int) :: (Int, Int)
```

Az n-esek érdekessége, hogy a listákkal és más nevezetes típusokkal ellentétben rendkívül körülményes az algebrai definíciójuk: minden elemszámra (n-re) meg kellene adnunk egy típust és egy konstruktort, például `data (a, b) = (,) a b` párokhoz, `data (a, b, c) = (,,) a b c` hármasokhoz és így tovább. A nyelvi implementáció(k) ezt általában nyelvi szinten kezelik, és limitálják a maximális elemszámot.

> **Érdekesség**
>
> Az n-esek konstruktorai a vesszők számának növelésével mind külön-külön elérhetőek függvény szimbólumként:
>
> ```
> ghci> :t (,,,,)
> (,,,,) :: a -> b -> c -> d -> e -> f -> (a, b, c, d, e, f)
> ```
>
> A maximális elemszám implementációfüggő, de általában 62 vagy 64 a GHC megvalósításban:

```
ghci> :t
(,,,
```

paramétertípusok számossága összeszorozódik, a függvénytípus esetében pedig hatványozódik.

Ezen példákat kombinálva most kiemelünk néhány nevezetes izomorfizmust.

Maybe vs. Either

Amikor az `Either` típus második típusparaméterének számossága 1, akkor megfeleltethető a `Maybe` típusnak; egy ilyen eset a unit típus, ahol a `Right ()` lesz a `Nothing` megfelelője.

```
maybeToEitherUnit :: Maybe a -> Either () a
maybeToEitherUnit Nothing = Left ()
maybeToEitherUnit (Just x) = Right x

eitherUnitToMaybe :: Either () a -> Maybe a
eitherUnitToMaybe (Left ()) = Nothing
eitherUnitToMaybe (Right x) = Just x
```

Nyilván ha definiálunk egy másik egyelemű típust, amely izomorf a unit típussal, akkor hasonló megfeleltetés írható fel.

A két transzformáció kompozíciója az identitás, tehát `maybeToEitherUnit . eitherUnittoMaybe = id`; a függvények egyenlősége minden bemeneti értékre való egyenlőségükből következik (functional extensionality).

Rendezett pár vs. Either

Amikor a pár típus (n-es, ahol n = 2) valamelyik paramétertípusa kételemű (pl. `Bool`), akkor az elemszáma a másik paramétertípus elemszámának kétszerese; ez igaz arra az `Either` típusra is, amelyet ugyanazzal a típussal paraméterezünk, hiszen akkor kétszer vesszük a paramétertípus elemeit (egyszer a `Left` konstruktorba, másodszor a `Right` konstruktorba "csomagolva").

```
pairWithBoolToEither :: (Bool,a) -> (Either a a)
pairWithBoolToEither (False, x) = Left x
pairWithBoolToEither (True, x) = Right x

eitherToPairWithBool :: Either a a -> (Bool,a)
eitherToPairWithBool (Left x) = (False, x)
eitherToPairWithBool (Right x) = (True, x)
```

A megfeleltetés lényege, hogy a kételemű típus két elemét (`Bool` esetén a két konstruktorát) feleltetjük meg az `Either` két konstruktorának.

Rendezett pár vs. függvény

Ha a pár típus két típusparamétere megegyezik, akkor az elemszáma a paramétertípus elemszámának négyzete lesz, hiszen az elemszámot szorozzuk magával; hasonlóan a típusértékek számának négyzetét kapjuk, ha a függvénytípus értékkészletének elemszáma 2 (pl. `Bool`).

```
pairToFunctionFromBool :: (a,a) -> (Bool -> a)
pairToFunctionFromBool (x, y) False = x
pairToFunctionFromBool (x, y) True = y

functionFromBoolToPair :: (Bool -> a) -> (a,a)
functionFromBoolToPair f = (f False , f True)
```

Itt a megfeleltetés megvalósítása annyiból érdekes, hogy a függvényre nem tudunk mintát illeszteni, így egy változómintát illesztünk rá és a definíció jobboldalán kiértékeljük a két típusértékre, visszanyerve a két "becsomagolt" értéket.

Típusosztályok

Az előző fejezetben említettük, hogy a típusok értékeinek megjelenítése (szöveggé alakítása) nem történik meg automatikusan — ahogyan semmilyen más automatikus típuskonverzió sem történik az erősen típusos nyelvekben — ezért ha meg akarjuk jeleníteni egy típus értékeit, akkor meg kell adni egy olyan függvényt, amely szöveggé alakítja az adatot. Mivel ez egy általános művelet, amelyet minden "megjeleníthető" típusra meg szeretnénk adni, praktikus lenne túlterheléssel definiálni. A Haskell (és más funkcionális nyelvek) megoldása erre a típusosztály, amely egyfajta interfész: a megvalósítandó függvények egy csoportját deklarálja, és minden típus, amely példánya (tagja) az osztálynak, megvalósítja az osztályban leírt műveleteket.

> **Megjegyzés**
>
> Egy típus tetszőlegesen sok típusosztály/csoport tagja lehet, de egy adott típusra csak egyetlen példányosítást lehet megadni.

> **Érdekesség**
>
> A típusosztályok valójában függvénykollekcióként vannak implementálva, a típusosztályokra való hivatkozások az osztályban foglalt függvények megvalósításaira vonatkozó referenciák. Minden olyan esetben, amikor egy típusváltozóra típusosztály megszorítás van érvényben, egy extra argumentumban kerül átadásra a típusosztály megvalósítását adó függvénykollekció.

A típusosztály megvalósításával a típus túlterhelheti a típusosztályban deklarált szignatúrákat, ami egyszerűbb, *olvashatóbb* kódokat eredményez. Továbbá a típusosztályok megjelenhetnek függvények, típuskonstruktorok vagy más típusosztályok szignatúráiban (típusváltozók megszorításaként, ún. típuskényszerként), így konkrét típusok megadása helyett sokszor egy megszorított, de *általános* típust tudunk használni a szignatúrában, újrafelhasználhatóbb kódot eredményezve. Az általános, megszorított típus előnye, hogy olyan típusokat is megenged, amelyeket adott esetben csak később fogunk elkészíteni (és példányosítani rájuk az osztályt).

> **Fontos**

> A típusosztályoknak a túlterhelésen túl a legfontosabb feladata az ad-hoc polimorfizmus megvalósítása a típusosztályok használatával a szignatúrákban. Mivel Haskellben és a hasonló funkcionális nyelvekben nincsenek altípusok és altípusos polimorfizmus, olyan esetekben, ahol a függvény viselkedése nem teljesen parametrikus, valamilyen kellően általános típust kellene megjelölni, amelynek "altípusai" használhatóak a konkrét alkalmazásokban. Ezt valósítjuk meg az úgynevezett típuskényszerekkel (típusmegszorításokkal), amelyek a típusosztályt használják a lehetséges típusok körének meghatározására az altípusosság helyett.

Konkrét típusok nevezetes osztályai

Show

Az egyik nevezetes típusosztály épp a "megjeleníthető" típusok csoportja; a neve Haskellben `Show`, és az egyik művelet, amit előír, a `show :: a -> String`, amely a típus elemeit képezi szövegekre. A `Show` típusosztály *leegyszerűsített* változata így lenne megadható Haskellben:

```
class Show a where
  show :: a -> String
```

A `class` kulcsszó után megadjuk a típusosztály szignatúráját, majd a `where` kulcsszó után felsoroljuk az osztály műveleteit.

Fontos megjegyezni, hogy a tényleges implementációban a `Show` osztály további függvényeket is megvalósít: elemek egy listájának megjelenítése például az egyes elemek megjelenítéséből származtatható, tehát ha tudjuk, hogyan kell megjeleníteni a `Point` egy elemét, akkor annak segítségével meg tudjuk jeleníteni pontok egy listáját is. A példányosítás során azonban felüldefiniálhatjuk a származtatott műveleteket is, ha egyedi megoldást akarunk megvalósítani (például a pontok listáját egyedi logika szerint megjeleníteni a képernyőn).

> **Megjegyzés**
>
> A típusosztályok mindig megadják, hogy az összes bennük foglalt függvény közül melyeket kötelező megvalósítani (ez az úgynevezett *MINIMAL* rész), és mely függvények esetében opcionális a megvalósítás.

> **Fontos**

> A típusosztályoknak a függvénynevek túlterhelésén túl épp a származtatott, "kényelmi" műveletek adják a lényegét; ezekhez "default" megvalósítás adható a kötelezően definiálandó függvényekre építve, tehát "ingyen" megkapjuk őket az elemi műveletek megadása után. (Némi hasonlóság érezhető az OO nyelvek interfészeinek "default" metódusaival, de erre nem szeretnénk ráerősíteni.)

Eq

Nevezetes típusosztály azon típusok köre, amelyek elemeinek egyenlősége vizsgálható. A megvalósítandó művelet itt a két típusbeli elem egyenlőségének eldöntése, melynek szignatúrája `(==) :: a -> a -> Bool` (alternatívaként a nem egyenlőség is megadható, hiszen ezek egymás tagadásából adódnak).

Ord

Szintén nevezetes azon típusok csoportja, amelyekben az elemek rendezettek egy "kisebb vagy egyenlő" reláció szerint. Ebbe az osztályba akkor tartozhat bele egy típus, ha megvalósítja a `(<=) :: a -> a -> Bool` műveletet (vagy alternatívaként a `compare` műveletet, amely két elemről eldönti, hogy kisebb, egyenlő, vagy nagyobb relációban vannak).

Enum

Az `Enum` típusosztállyal már találkoztunk az intervallumkifejezések tárgyalásakor, hiszen a felsorolható (enumerálható) típusok támogatják a `..` kifejezéseket. Ebben a típusosztályban két függvényt kell megvalósítani, amelyek az egész számokra képezik (`fromEnum :: a -> Int`) és az egész számokhoz rendelik (`toEnum :: Int -> a`) a típus elemeit.

Számtípusok osztályai

A Haskell számtípusait a numerikus típusosztályok fogják össze. A legtöbb számtípus beletartozik a `Num` osztályba, amely előírja az alapműveletek megvalósítását (pl. összeadás, kivonás, szorzás), az egészek az `Integral` osztályba (pl. egészosztás, maradékképzés), a törtek a `Fractional` osztályba (pl. osztás, reciprok), a levegőpontos típusok pedig a `Floating` osztályba (olyan műveletekkel, mint `pi`, `exp`, `log`, `sqrt`).

Típuskonstruktorok osztályai

A `Show`, `Eq`, `Ord` és `Enum` mind olyan osztályok, amelyekbe konkrét típusok tartoznak (olyanok, amelyeknek nincs paramétere); a `Bool` és a `Point` a fenti példákban konkrét típusok voltak, a `Maybe` vagy a `List` viszont paraméteres típusok (a fajtájuk `* -> *`, vagyis típussal paraméterezett típusok), ezekre nem tudjuk közvetlenül példányosítani az előbb említett típusosztályokat, először alkalmazni kell a típuskonstruktort konkrét típusokra vagy megszorított típusváltozókra. A `Show` például példányosítható a `Maybe Int` konkrét típusra, vagy az általános `Maybe a` típusra, de utóbbi esetben az `a` típusváltozóra megszorítást kell alkalmazni (például feltenni, hogy a paramétertípus megjeleníthető).

Vannak olyan típusosztályok is, amelyek kifejezetten paraméteres típusokat, típuskonstruktorokat foglalnak csoportba. Például a `Functor` azon "konténerek" osztálya, amelyeknek elemein alkalmazhatunk egy transzformáló függvényt. Ebbe az osztályba bele tud tartozni a `Maybe` és a `List` is, mindkettő olyan típuskonstruktor, amelynek egyetlen paramétertípusa (elemtípusa) van. A fejezet további részeiben részletesen tárgyaljuk a `Functor` osztályt és példányait.

Típusosztályok kiterjesztése

A típusosztályok bővíthetőek, kiterjeszthetőek. Az `Ord` például az `Eq` osztályból van "származtatva", így minden olyan típus, ami az `Ord` osztályba tartozik, beletartozik az `Eq` osztályba is, tehát minden rendezett típus elemei között vizsgálható az egyenlőség. Ezzel az osztályok között alosztály kapcsolatokat tudunk kialakítani, egyszerűbb szignatúrákat tudunk használni, és a kód végül rövidebb és egyszerűbb lesz.

Példányosítás

Egy típusosztályba úgy kerülhet be egy típus, ha példányosítja azt, tehát megvalósítja (definiálja) a típusosztály által előírt függvényeket. Ennek két alapvető útja van: az egyik esetben a függvényekhez a típus szerkezete szerint automatikusan állítunk elő definíciókat (ez az ún. "deriving", vagyis automatikusan levezetett példány), a másik esetben pedig definíciók sorozatával, "kézzel" adjuk meg a függvények típusspecifikus túlterhelését.

Deriving

Sok típus és típusosztály esetében "nyilvánvaló" a példányosítás. A korábban látott `Point` típus esetében (melynek egyetlen `Int -> Int -> Point` konstruktora volt), az `Eq` például triviálisan definiálható oly módon, hogy két pont akkor egyenlő, ha a koordináták egyenlőek. Hasonlóan, két `Maybe Int` akkor egyenlő, ha mindkettő `Nothing`, vagy mindkettő `Just` és a bennük tárolt számok egyenlőek. Továbbá két lista akkor tekinthető egyenlőnek, ha ugyanannyi elemük van, és az elemek sorra egyenlőek. Az ilyen "default" példányok automatikusan legenerálhatóak számos nevezetes típusosztályhoz.

- A `Show` például veszi a konstruktor nevét szövegesen, és konkatenálja a szöveggé alakított paraméterekkel.
- Az `Ord` a konstruktorok megadásának sorrendjét veszi alapul, azon belül pedig a rendezett paraméterek alapján rendez.

> **Megjegyzés**
>
> A GHC Generics használatával nemcsak nevezetes típusosztályok példányosíthatóak automatikusan, hanem saját típusosztályokhoz is megadhatóak alapértelmezett (automatikusan generált) implementációk a típus szerkezete szerint.

Instance

A típusosztály példányosítása a szükséges függvények megadásával is történhet; az osztály által elvárt összes szignatúrához meg kell adni a típusra specifikus megvalósítást az `instance` kulcsszót követő `where` szakaszban.

A `Bool` típusra (amelynek a `True` és a `False` a konstruktorai) a következőképp példányosítható a `Show` típusosztály:

```
-- data Bool = False | True
instance Show Bool where
  show False = "hamis"
  show True = "igaz"
```

A `deriving` itt nem jelenik meg, helyette az `instance` kulcsszó után adjuk meg a kívánt példányt (`Show Bool`), majd a `where` után definiáljuk a `show` függvényt, amelyet elvár a `Show` típusosztály. Ennek hatására túlterheltük a `show` függvényt a `Bool` típusra:

```
ghci> show True
"igaz"
```

És ezzel egy időben a `Show` származtatott műveleteit is megkaptuk, például a `showList` függvényt:

```
ghci> showList [False, True] ""
"[hamis,igaz]"
```

> **Érdekesség**
>
> Ahogyan már említettük, a `Show` példányosítása során például elegendő a `show` függvényt megadni. A `showList` definiálása nem kötelező (használhatjuk a `show`-ra épülő megadását), de lehetséges, és akkor az értékek listájának megjelenítése a típusra specializálható. A karakterek típusának esetében például a `[Char]` megjelenítése speciális, hiszen ez felel meg a `String` típusnak, így elválasztójelek nélkül, idézőjelek között jelennek meg a karakterek.

A `Show` csak konkrét típusra példányosítható, típuskonstruktorra nem, így a `Maybe` vagy `Either` esetében vagy konkretizáljuk a típust, vagy megszorítást alkalmazunk. Az `Either` konstruktort alkalmazva az `Int` és `String` típusokra például már konkrét típust kapunk, amelyre megadható a `Show`:

```
instance Show (Either Int String) where
  show (Left x) = "[left] " ++ show x
  show (Right x) = "[right] " ++ show x
```

Egy másik példában a `Maybe` esetén feltehetjük, hogy a típusparaméter megjeleníthető (a `Show` példánya), így megadható a `Show` a típusváltozóval alkalmazott típuskonstruktorra:

```
instance Show a => Show (Maybe a) where
  show Nothing = "[nothing]"
  show (Just x) = "[just] " ++ show x
```

Functor

A típusosztályok egészen általános műveletek túlterhelésére használhatóak, és akkor válnak igazán érdekessé, amikor nem konkrét típusokra, hanem típuskonstruktorokra vonatkoznak. Talán a legnevezetesebb ezek közül a `Functor`, amellyel konténer típusok (kontextusok) "tartalmára" (benne foglalt elemeire) alkalmazhatunk függvényeket.

A `Functor` típusosztály művelete a listákon megadott `map` függvény általánosítása, típusa `fmap :: (a -> b) -> f a -> f b`; itt `f` a konténer, `a` és `b` a (nem feltétlenül különböző) elemtípusok, és a művelet lényege, hogy a konténer elemein végigiterálva azokat egyesével transzformálja az `a -> b` típusú függvénnyel. Fontos, hogy az `fmap` mindig úgy alakítja át az összetett adatot, hogy a konténer szerkezete nem változik (például a lista esetében nem változik a hossza, fa esetében nem változik az alakja).

> **Megjegyzés**
>
> A típusosztályok sokszor informális specifikációkat fogalmaznak meg, amelyeket érdemes betartani a példányosítás során. A `Functor` esetében például a következő két egyenlőségnek kell teljesülnie az `fmap` függvényre:
>
> - Identitás: `fmap id == id` (az identitásfüggvényt alkalmazva a konténer minden elemére az eredeti konténert kapjuk vissza)
> - Kompozíció: `fmap (f . g) == fmap f . fmap g` (ha két függvény kompozíciójával transzformálunk, ugyanazt érjük el, mintha a két függvénnyel való transzformációk kompozícióját vesszük)
>
> Az ilyen megkötéseket a típusosztályok kódja nem, de a dokumentációja tartalmazza. A fordítóprogram és az értelmező nem kényszerítik ki ezeket, de a ráépülő kódok feltételezhetik a teljesülésüket.

Hibakezelés

Klasszikus alkalmazása a `Functor` típusosztálynak a `Maybe` és az `Either` opcionális értéktípusok példányaival való hibakezelés. A `Maybe` esetében a `Nothing` konstruktor használható arra, hogy egy számítás sikertelenségét/hibáját reprezentálja, melyhez kapcsolódóan a hibakezelést/hibapropagálást a Functor függvényalkalmazás művelete el tudja

rejteni — és hasonlót tudunk kivitelezni az `Either` típussal is, viszont ott a `Left` konstruktorral nemcsak a hiba tényét, de annak okát is képesek vagyunk reprezentálni. Ilyen jellegű hibakezelésre akkor lehet szükség, ha egy potenciálisan sikertelen számítás eredményén további számításokat akarunk végezni, és szükséges a "kivételes" eset kezelése, propagálása. Ezt a `Functor` függvényalkalmazása automatikusan el tudja végezni az értékek `Maybe`-be csomagolása esetén.

Vegyünk egy tetszőleges `z` kifejezést, amelynek típusa `Maybe Int`. Tegyük fel, hogy a `z` értékén az `even` függvényt szeretnénk alkalmazni, de nem tudjuk, hogy az `Int` érték kiszámítása sikeres volt-e. Ennek a megoldása történhet egy `case` esetszétválasztással, ahol az `even` alkalmazása előtt megvizsgáljuk, hogy sikeres volt-e a szám kiszámítása (`Just` eset) és csak ebben az esetben alkalmazzuk az `even` függvényt, a `Nothing` esetben pedig visszaadunk egy hamis értéket:

```
case z of { Nothing -> False ; (Just x) -> even x }
```

A probléma ezzel az, hogy a hamis érték esetében nem tudjuk, hogy a hibát jelzi vagy pedig a szám sikeres kiszámítása esetén értékelődött ki az `even` függvény hamisra, mert páratlan volt a szám. Jobb megközelítés, ha az esetszétválasztás után is "továbbvisszük" az opcionalitást, és így a hibajelzést. A `Just`-ból tehát `Just` lesz, a `Nothing`-ból pedig `Nothing`:

```
case z of { Nothing -> Nothing ; (Just x) -> Just (even x) }
```

Képzeljük el, hogy mindahányszor egy további műveletet szeretnénk végrehajtani az opcionális számítás eredményén, újabb és újabb esetszétválasztást kell beépítenünk, olvashatatlanná téve a kódot. Ez a hibakezelési logika elrejthető egy magasabb rendű függvénybe, amely megkapja a függvényt és az esetleges paramétert, majd elkészíti az esetleges függvényértéket:

```
(Int -> Bool) -> Maybe Int -> Maybe Bool
```

Általánosan:

```
(a -> b) -> Maybe a -> Maybe b
```

Ez pontosan a `Functor` osztályban szereplő `fmap` típusa a `Maybe` típuskonstruktorra specializálva, amelyet a `Maybe`-re a következőképp terhelünk túl:

```
instance Functor Maybe where
  fmap _ Nothing = Nothing
  fmap f (Just x) = Just (f x)
```

Ez a példány valóban megtartja a `Maybe` szerkezetét (`Just`-ból `Just` lesz, `Nothing`-ból `Nothing`), és a kontextusban lévő értékre alkalmazásra kerül a paraméterfüggvény.

A `Functor Maybe` már lehetőséget ad arra, hogy a fentebb tárgyalt hibakezelést elrejtsük az `fmap` alkalmazásba, melynek infix operátora a `<$>` (emlékeztető: a függvényalkalmazás operátora a `$` volt). Az előző példánkat tehát lényegesen leegyszerűsíti a `Functor` használata, hiszen az esetszétválasztás logikáját elrejti az `fmap` függvénybe:

```
case z of { Nothing -> Nothing ; (Just x) -> Just (even $ x) }
```

helyett egyszerűen írhatjuk azt, hogy

```
even <$> z
```

Az `Either` esetében a hibakezelés úgy történik, hogy a hibát a `Left` konstruktor keretezi, míg sikeres alkalmazás esetén a `Right` csomagolja az értéket. A `Functor` tehát itt úgy példányosítható, hogy amennyiben a `Right` (sikeres) eset áll fenn, alkalmazzuk a paraméterfüggvényt, egyébként továbbvisszük a hibát:

```
instance Functor (Either a) where
  fmap _ (Left x) = Left x
  fmap f (Right x) = Right (f x)
```

Iteráció

A `Functor` legfőbb alkalmazása azonban nem a hibapropagáció, hanem az iteráció. Megismételjük a műveletének (amely egy magasabb rendű függvény) típusát: `(a -> b) -> f a -> f b`. Ha `f`-re tetszőlegesen sok elemet tartalmazó konténerként gondolunk, világossá válik, hogy az osztály

műveletének lényege a paraméterfüggvény alkalmazása minden konténerelemre, az elemeken való iterációval.

> **Megjegyzés**
>
> Ez a fajta logika természetesen értelmezhető a `Maybe` konstruktorra is, hiszen az csak egy speciális konténer, ami maximum 1 elemet tartalmaz.

A hibakezeléses példában az `even` függvényt akartuk alkalmazni egy opcionális értékre. Ennek a gondolatnak az általánosítása, hogy amennyiben nem 0 vagy 1, hanem több érték van a konténerben, mindegyikre szeretnénk alkalmazni a függvényt. A lista típusunk esetében a lista elemein való iterációval (amelyet rekurzív függvénnyel valósítunk meg) valósítjuk meg a transzformációt, amiben minden értékről eldöntjük, hogy páros-e:

```
-- List a = Nil | Cons a (List a)
mapEven Nil = Nil
mapEven (Cons x xs) = Cons (even x) (mapEven xs)
```

Látható, hogy a fenti függvényből kirajzolódik az általános logika, amelyet a listákkal foglalkozó fejezetben már láttunk: primitív rekurzióval bejárjuk a listát, és az aktuálisan kibontott elemre alkalmazzuk a függvényt, majd "visszacsomagoljuk" (a `Cons` konstruktorba). Ez természetesen megfogalmazható az elemeken elvégzett művelet általánosításával, amivel megérkezünk a `map` függvényhez, amely nem más, mint a listákra vett `fmap`:

```
instance Functor (List a) where
  fmap _ Nil = Nil
  fmap f (Cons x xs) = Cons (f x) (fmap f xs)
```

És valóban, ez is megtartja az adat szerkezetét, hiszen `Nil`-ből `Nil`, `Cons`-ból pedig `Cons` lesz; ez a listák esetében azt implikálja, hogy a `map` sosem változtatja meg a lista hosszát.

A `mapEven` a példányosítás után már kifejezhető explicit rekurzió és esetszétválasztás nélkül, támaszkodva a `Functor` műveletére:

```
mapEven :: Functor f => f Int -> f Bool
mapEven = fmap even
```

Vegyük észre, hogy ez a függvény már nemcsak `Maybe` vagy `List` esetén használható a párosság eldöntésére, hanem tetszőleges konténerre alkalmazható lesz, amelyre példányosítjuk a `Functor` típusosztályt, tehát egy generikus, újrafelhasználható megoldást készítettünk.

Applicative

Hibakezelés

A `Functor` segítségével megvalósítottunk az opcionális, `Maybe` értékekre egyfajta hibakezelést. Hasonló hibakezelés megvalósítható a kiértékelendő függvényre vonatkozóan is. Ekkor valójában olyan esetszétválasztást kell megadni, amely megvizsgálja, hogy a függvény és az argumentum is hibamentesen kiértékelődött, és csak ebben az esetben alkalmazzuk a függvényt, különben továbbvisszük a hibát.

```
Maybe (a -> b) -> Maybe a -> Maybe b
```

Vagy általánosan, `f` típuskonstruktorra:

```
f (a -> b) -> f a -> f b
```

Ez a Functor általánosításaként is felfogható, jelölése a `<*>` és az `Applicative` típusosztály példányosításával lehet túlterhelni:

```
instance Applicative Maybe where
  pure x = Just x
  Nothing <*> _ = Nothing
  _ <*> Nothing = Nothing
  Just f <*> Just x = Just (f x)
```

Vagy rövidebben:

```
instance Applicative Maybe where
  pure = Just
  Nothing <*> _ = Nothing
  Just f <*> m = fmap f m
```

Ez a fajta hibakezelés természetesen általánosítható az `Either` típusra is, ahol nemcsak a hibát, de a hiba okát is tudjuk tárolni és propagálni. Érdemes megnézni, hogyan van példányosítva az `Applicative` az `Either` típusra:

```
instance Applicative (Either a) where
  pure =  Right
  Left x <*> _ = Left x
  Right f <*> x = fmap f x
```

Látható, hogy az `Applicative` megvalósításához felhasználjuk a `Functor` megvalósítását (az `fmap` alkalmazásával). Amennyiben az alkalmazandó függvény adott, akkor az `fmap` fogja meghatározni az eredményt, amiről tudjuk, hogy megőrzi az `x` szerkezetét.

```
ghci> Right even <*> Right 42
Right True
ghci> Left "no fun" <*> Right 42
Left "no fun"
ghci> Right even <*> Left "no value"
Left "no value"
```

Amint látható, a `Functor` és az `Applicative` fokozatosan vezeti be az opcionális értékekre a hibakezelést. A hagyományos függvényalkalmazás (`$`) sem a függvény, sem a paraméter esetén nem tudja kezelni az opcionalitást. A `Functor` a paraméter esetén már igen, az `Applicative` pedig mind a függvény, mind a paraméter esetében kezeli az opcionális értékeket, és a hibás esetben meg sem próbálja elvégezni az alkalmazást.

Általánosan, tetszőleges `f` típuskonstruktorra a következőképp alakulnak ezek a függvényalkalmazási műveletek:

```
($)    ::                     (a -> b) ->   a ->   b
(<$>) :: Functor f =>         (a -> b) -> f a -> f b
(<*>) :: Applicative f => f (a -> b) -> f a -> f b
```

Érdekesség: Applicative függvény

A függvény típuskonstruktor fajtája `* -> * -> *`, tehát ha a paraméter vagy az eredmény típusát rögzítjük, `* -> *` fajtájú típust (egyetlen típussal paraméterezett típust) kapunk, amelyre példányosítható a `Functor` és az `Applicative`. Vegyük a következő függvényeket:

```
(&&) :: Bool -> Bool -> Bool
(>0) :: Int -> Bool
even :: Int -> Bool
```

Tegyük fel, hogy szeretnénk egy számról eldönteni, hogy pozitív (`>0`) és páros-e (`even`). Ehhez alkalmazni kell rá mindkét predikátumot, majd az eredmények konjunkcióját kell venni. A következő függvény definiálja ezt a predikátumot:

```
\n -> (&&) ((>0) n) (even n)
```

Ez a számítás azonban felfogható úgy is, hogy egy "becsomagolás" egy `Functor` példányba, ahol a kontextus az `(Int ->)` függvénytípus, tehát `Int -> a` valamely `a` típusra. Erre a funktorra az `fmap` a kompozíció, a `(<*>)` operátor segítségével pedig a részben alkalmazott, funktorban lévő függvényt tudjuk alkalmazni:

```
(&&) <$> (>0) :: Int -> Bool -> Bool
(&&)  .  (>0) :: Int -> Bool -> Bool
even          :: Int -> Bool
```

A `Bool -> Bool` függvény bekerült az `(Int ->)` funktorba, továbbá az `even` is ugyanebben a funktorban van, csak `Bool` típusparaméterrel (de ugyanaz a kontextusuk). A `Bool -> Bool` pedig már alkalmazható a `Bool` értékre (a funktoron belül), aminek az eredménye `Int -> Bool`, továbbra is a funktoron belül, az `Applicative` osztály `<*>` műveletével:

```
(&&) <$> (>0) <*> even :: Int -> Bool
```

Ez a kifejezés tehát, a `Functor` és `Applicative` műveleteit kihasználva leírja azt a számítást, amely a konjunkciót az `Int` predikátumok körében (funktorában) végzi el. Rövidebben:

```
liftA2 (&&) (>0) even
```

A `liftA2` függvény általánosan írja le azt a műveletet, hogy a bináris függvényalkalmazást bevisszük az applikatív funktorba:

```
liftA2 :: (a -> b -> c) -> f a -> f b -> f c
```

További magyarázat és példák találhatóak az Applicative dokumentációjában.

Foldable

Ahogyan a listák `map` műveletét is tudtuk általánosítani, és típusosztályba foglalni a hasonló típusokat, ezt megtehetjük a listák `foldr` műveletével is. Utolsó nevezetes típusosztály példánk a `Foldable`, amely a listák hajtogatását általánosítja tetszőleges összetett adatszerkezetre, amely valamilyen típusú elemeket tárol és azokat redukálni tudja.

Foldable List

Korábbi fejezetben megmutattuk, hogy a listák összegzése, produktuma, vagy a lista elemeinek megszámlálása mind ugyanazt a rekurziósémát követik, így ezeket általánosíthatjuk egyetlen hajtogató műveletté, a `foldr` függvénnyé. Azt is végignéztük, hogyan vezethetőek vissza ezek a nevezetes műveletek a `foldr` függvényre, tehát csak egyszer kell megadni a hajtogatás "tételét", és erre visszavezethetjük feladatok széles körét.

Ha a listákon megadott `foldr` függvényből kiabsztraháljuk a lista típuskonstruktort, akkor egy általános hajtogató függvényt kapunk:

```
foldr :: (a -> b -> b) -> b -> t a -> b
```

Itt `t` a konténer, amelynek az elemeit hajtogatni akarjuk, ezen típuskonstruktorok csoportját fogjuk `Foldable` osztálynak nevezni. Ezek után a listákon megadott `foldr` függvény valójában csak az általános logika alkalmazása a listára: az elemeken való iteráció, és azok redukálása egy értékre. Ezt nem csak listákra, hanem tetszőleges konténerre értelmezhetjük, így a listákra vonatkozóan ez csupán egy specializáció (példány):

```
instance Foldable List where
  foldr f z Nil = z
  foldr f z (Cons x xs) = x `f` (foldr f z xs)
```

A `Foldable` az a típusosztály, amely tartalmazza az összes olyan konténert (típuskonstruktort), amelynek elemei egymásra hajtogathatóak egy tetszőleges művelettel. Ez a típusosztály a `foldr` műveletre építve számos egyéb függvényt is definiál, amelyeket "ingyen" megkapunk a példányosítás során:

```
ghci> foldr (+) 0 (Cons 1 $ Cons 42 Nil)
43
ghci> sum (Cons 1 $ Cons 42 Nil)
43
ghci> product (Cons 1 $ Cons 42 Nil)
42
ghci> length (Cons 1 $ Cons 42 Nil)
2
```

Foldable Tree

A `Foldable` tárgyalása során a másik klasszikus példány a (bináris) fa adattípus példánya, amelyen ugyanilyen könnyen definiálhatjuk a hajtogatást. A listánál valamivel bonyolultabb típus a bináris fa; a fő különbség, hogy a második konstruktorban kétszer szerepel rekurzívan a típus, így valósítva meg a többszörös összetettséget:

```
data Tree a = Leaf | Node (Tree a) a (Tree a) deriving Show
```

Mielőtt megadjuk a `Foldable` példányt, vegyük a `Functor` példányt is erre a típusra, gyakorlásképp. A szerkezete alapján a fa típusra könnyen megadható a `Functor`, csupán arra kell figyelni, hogy mindkét rekurzív "ágra" rekurzívan kell alkalmazni a "transzformáló" függvényt:

```
instance Functor Tree where
  fmap _ Leaf = Leaf
  fmap f (Node l x r) = Node (fmap f l) (f x) (fmap f r)
```

Itt tehát az utolsó sorban az `fmap` duplán rekurzív, mindkét részfára rekurzívan alkalmazza az `f` függvényt, szemben a listáknál látott egyszeres rekurzióval. Ezzel a megoldással már a fán belül is tudunk függvényeket alkalmazni a node-ok elemeire:

```
ghci> even <$> Node Leaf 42 (Node Leaf 1 Leaf)
Node Leaf True (Node Leaf False Leaf)
ghci> id <$> Node Leaf 42 (Node Leaf 1 Leaf)
Node Leaf 42 (Node Leaf 1 Leaf)
ghci> id $ Node Leaf 42 (Node Leaf 1 Leaf)
Node Leaf 42 (Node Leaf 1 Leaf)
```

Vegyük észre, hogy a két utolsó példában a `$` és a `<$>` különböző hatással bír: az első nem járja be a fát, míg a második igen, és elemenként veszi az identitást.

A `Functor` után megadjuk a `Foldable` példányt is:

```
instance Foldable Tree where
  foldr f z Leaf = z
  foldr f z (Node l x r) = x `f` (foldr f (foldr f z l) r)
```

Itt a hajtogatás műveletét először a bal részfára végezzük el, majd a jobb részfára, tehát balról jobbra haladunk a hajtogatással.

Hasonlóan az előző példához, összegezhetővé és megszámlálhatóvá válnak a fák:

```
ghci> sum $ Node Leaf 42 (Node Leaf 1 Leaf)
43
ghci> product $ Node Leaf 42 (Node Leaf 1 Leaf)
42
ghci> length $ Node Leaf 42 (Node Leaf 1 Leaf)
2
ghci> length $ Node Leaf 42 (Node Leaf 1 (Node Leaf 45 Leaf))
3
ghci> sum $ Node Leaf 42 (Node Leaf 1 (Node Leaf 45 Leaf))
88
```

A fontos észrevétel azonban itt egyértelműen az, hogy a `Foldable` mint "típus tulajdonság" jelenik meg a programszövegben, ezzel lehetővé téve, hogy a fejlesztés során megadhassunk olyan műveleteket, amelyek hajtogatható típusokra értelmezhetőek. Ezek a műveletek alkalmazhatóak lesznek minden olyan típusra, amelyekről korábban már megmutattuk, hogy hajtogathatóak, vagy később megmutatjuk, hogy beletartoznak az osztályba. Ezzel a megoldással a Haskell lehetővé tette, hogy generikus, újrafelhasználható típusokat és függvényeket készítsünk, amelyeket a típusosztályok mint szerződések mentén tudunk komponálni, típushelyes programokat eredményezve.

Ajánlott irodalom

- Nyékyné Gaizler Judit. Programozási nyelvek, Kiskapu, 784 oldal, 2003. ISBN: 9789639301467.
- Judit Nyéky-Gaizler et al. Advanced Programming Languages, Eötvös Loránd University, 2014. ISBN 978-963-284-450-3.
- Miran Lipovaca. Learn You a Haskell for Great Good!: A Beginner's Guide
- Simon Thompson. Haskell: The Craft of Functional Programming (3rd ed.)
- Bryan O'Sullivan, John Goerzen, and Don Stewart. Real World Haskell (1st ed.). O'Reilly Media, Inc. 2008.
- Graham Hutton. Programming in Haskell (2nd ed.). Cambridge University Press. 2016.

Feladatgyűjtemény

Bevezetés

Ez a feladatgyűjtemény az ELTE Informatikai Karán oktatott Funkcionális programozás (BSc) esti tárgyhoz készült. A feladatgyűjtemény a tárgy jegyzetében leírt témakörökhöz nyújt további gyakorló feladatokat. Ezen dokumentum tartalma követi a gyakorlati kurzus felépítését, így a fő fejezetek strukturája a kurzus tematikáját követi. Minden fejezetben vannak alapvető feladatok, amelyeket a gyakorlatok során oldanak meg a hallgatók, illetve vannak házi (gyakorló) feladatok, amelyek további példákkal szolgálnak, illetve a mélyebb megértést szolgálják.

> **Megjegyzés**
>
> Az itt leírt feladatokhoz a Glasgow Haskell Compilation System (GHC) használata javasolt. A feladatokhoz tartozó függvényszignatúrákat és teszteseteket a GHC 9.4.7-es verziójával készítettük el.
>
> A feladatok megoldását egy `.hs` kiterjesztésű fájlban javasoljuk elkészíteni (amelybe érdemes importálni a `Data.List`, `Data.Char` és `Data.Foldable` modulokat). A megoldás elkészítéséhez először másold be a fájlba a megadott függvényszignatúrát (amennyiben van), majd add meg a feldatban kért definíciót. A kész megoldásokat tartalmazó fájlt töltsd be a Haskell interpreterbe (`ghci`), majd másold be a teszteseteket is ide. A feltüntetett egységtesztek mindegyike `True` értéket ad eredményül helyes megoldás esetén.
>
> A feladatok nehézségét ⭐-okkal jelöli a feladatgyűjtemény. Minél több ⭐ szerepel a feladat szövegében, annál bonyolultabb megoldást igényel. A különösképpen nagy kihívást jelentő feladatokat a 🔥 karakter jelöli.

Források, további feladatgyűjtemények

A feladatgyűjtemény elkészítése során több feladatot felhasználtunk a következő forrásokból:

- ELTE Funkcionális programozás jegyzet, 2012: http://lambda.inf.elte.hu/Index.xml

- Poór Artúr, Funkcionális programozás gyakorlatok, 2020: https://people.inf.elte.hu/poor_a/

Ezúton is javasoljuk ezen feladatgyűjtemények tanulmányozását és feladatainak megoldását.

Jogi nyilatkozat

A feladatsorban szereplő minden adat kizárólag szemléltetési célt szolgál. A megadott nevek, helyszínek, események és egyéb információk kitaláltak, illetve véletlenszerűen generáltak. Bármilyen hasonlóság valós személyekkel, szervezetekkel vagy eseményekkel csupán a véletlen műve.

1. gyakorlat: Ismerkedés a Haskell értelmezővel

Az első gyakorlaton kizárólag a Glasgow Haskell Compilation System értelmezőjével (interpreterével), a Haskell szintaxisával és alapvető típusaival ismerkedünk. Az interaktív értelmező használatakor a következő parancsokat használjuk a gyakorlatok során:

- `:l` modul betöltése
- `:q` az interpreter leállítása
- `:r` betöltött modulok újratöltése
- `:t` egy kifejezés típusának kikövetkeztetése
- `:i` egy típusról, függvényről, konstruktorról vagy típusosztályról információk megjelenítése (pl. definíció, asszociativitás, típus, stb.)
- `:set +s` memóriaigény és futásidő monitorozásának bekapcsolása

A következő kérdéseket beszéljük meg a gyakorlaton, példákkal:

- Milyen alapvető típusok vannak Haskellben? Hogyan lehet ezen típusok értékeit leírni (számliterálok, karakterliterálok, stb.)?
- Milyen alapvető aritmetikai műveleteket lehet ezen típusokkal végezni?
- Hogyan működik a hatványozás Haskellben (természetes, egész és tört kitevő esetén)?
- Hogyan kell (standard) függvényeket alkalmazni?
- Hogyan kell függvényt definiálni az interpreterben?

Gyakorló feladatok

Az első gyakorlaton elhangzottak alapján a már forrásoknál említett, interaktív jegyzetben található "Grafika függvényekkel" feladatsor megoldását javasoljuk.

2. gyakorlat: Műveletek számokkal

Ezen a gyakorlaton az alapvető számtípusokkal (`Int`, `Integer`, `Float`, `Double`) ismerkedünk meg mélyebben. Ezenfelül, szó esik alapvető logikai műveletekről (pl. konjunkció, diszjunkció, negáció) és a pár további típusról (pl. `Char`, `Bool`, rendezett pár) és számokhoz kötődő típuscsoportokról (ún. típusosztályokról, pl. `Num`) is.

Alapvető feladatok

1. feladat (⭐): Definiáld a `double` függvényt, amely megszorozza a paraméterét 2-vel!

   Tesztek:

   ```
   double 1 == 2
   double 0 == 0
   double 100 == 200
   double 10.1 == 20.2
   ```

2. feladat (⭐): Add meg az `even` függvényt, amely eldönti a paraméteréről, hogy páros-e! Lehetséges-e alkalmazni ezt a függvényt?
3. feladat (⭐): Megszerkeszthető-e egy háromszög a három megadott oldala alapján?

   ```
   isATriangle :: Int -> Int -> Int -> Bool
   ```

   Tesztek:

   ```
   isATriangle 1 1 1
   isATriangle 4 3 2
   isATriangle 5 3 4
   isATriangle 10 4 10
   not (isATriangle 1 2 3)
   not (isATriangle 1 2 1)
   ```

4. feladat (⭐): Add meg az `isPythagoreanTriple` függvényt, amely eldönti, hogy a három paramétere pitagoraszi számhármast alkot-e!

   Tesztek:

```
isPythagoreanTriple 3 4 5
isPythagoreanTriple 8 6 10
isPythagoreanTriple 13 12 5
not (isPythagoreanTriple 1 2 3)
not (isPythagoreanTriple 10 25 2)
```

5. feladat (⭐ ⭐): Tegyük fel hogy egy racionális számot párként reprezentálunk. Add meg az `addRational` függvényt, amely két racionális számot (párt) összead!

```
addRational :: (Int, Int) -> (Int, Int) -> (Int, Int)
```

   Tesztek:

```
addRational (1,2) (1,2) == (4,4)
addRational (1,4) (1,2) == (6,8)
addRational (1,7) (1,10) == (17,70)
addRational (3,2) (5,7) == (31,14)
```

Gyakorló feladatok

1. feladat (⭐): Add meg a kör területét kiszámító függvényt! Legyen a függvény neve `circleArea`!

```
circleArea :: Floating a => a -> a
```

   Tesztek:

```
round (circleArea 3) == 28
round (circleArea 2) == 13
round (circleArea 1) == 3
```

2. feladat (⭐): Add meg a téglatest területét kiszámító függvényt! Legyen a függvény neve `cuboidArea`!

```
cuboidArea :: Num a => a -> a -> a -> a
```

   Tesztek:

```
cuboidArea 1 1 1 == 6
cuboidArea 1 4 2 == 28
cuboidArea 10 10 10 == 600
```

3. feladat (⭐): Add meg a téglatest térfogatát kiszámító függvényt! Legyen a függvény neve `cuboidVolume`!

```
cuboidVolume :: Num a => a -> a -> a -> a
```

Tesztek:

```
cuboidVolume 1 1 1 == 1
cuboidVolume 1 4 2 == 8
cuboidVolume 10 10 10 == 1000
```

4. feladat (⭐ ⭐): Szorozz össze két racionális számot! (`mult` legyen a feladatot megoldó függvény neve!)

```
mult :: (Int, Int) -> (Int, Int) -> (Int, Int)
```

Tesztek:

```
mult (1,2) (1,2) == (1,4)
mult (2,3) (3,2) == (6,6)
mult (10,3) (2,1) == (20,3)
```

5. feladat (⭐): Add meg két szám egymással vett egész osztásának maradékát és egész osztásának eredményét egy párba rendezve!

```
modDiv :: Integer -> Integer -> (Integer, Integer)
```

Tesztek:

```
modDiv 10 5 == (0,2)
modDiv 11 5 == (1,2)
modDiv 7 7 == (0,1)
modDiv 7 8 == (7,0)
modDiv 7 1 == (0,7)
```

6. feladat (⭐): Add meg egy helyvektor hosszát!

```
len :: (Double, Double) -> Double
```

Tesztek:

```
len (3, 4) == 5.0
len (4, 3) == 5.0
len (5, 12) == 13.0
```

7. feladat (⭐): Add meg egy helyvektor nyújtását konstansszorosára!

```
stretch :: Num p => (p, p) -> p -> (p, p)
```

Tesztek:

```
stretch (2, 4) 2 == (4, 8)
stretch (1, 6) 3 == (3, 18)
```

8. feladat (⭐⭐): Add meg koordinátákkal megadott két pont távolságát!

```
distance :: (Double, Double) -> (Double, Double) -> Double
```

Tesztek:

```
distance (2, 5) (5, 9) == 5.0
distance (-4, 3) (1, -9) == 13.0
```

9. feladat (⭐ ⭐): Állapítsd meg, hogy illeszthető-e két dominó! A dominókat egy-egy párban reprezentáljuk, amelyek akkor illetszhetők, ha van olyan oldaluk, amelyen azonos szám szerepel.

```
matches :: (Integer, Integer) -> (Integer, Integer) -> Bool
```

Tesztek:

```
matches (2, 4) (4, 6)
matches (4, 2) (4, 6)
matches (6, 2) (4, 6)
not (matches (2, 8) (4, 6))
```

3. gyakorlat: Mintaillesztés

Ezen a gyakorlaton a hallgatók megismerkednek az alapvető számtípusok közötti konverziós függvényekkel (`fromInteger`, `fromIntegral`, `round`, `ceiling`, `floor`, stb.). A gyakorlat fő témája viszont az esetszétválasztással (mintaillesztéssel) megadott függvények, amelyeket a jól ismert logikai

műveletek definiálásával vezetünk be.

Alapvető feladatok

1. feladat (⭐): Add meg egy szám duplájának a gyökét! Adj meg helyes definíciókat az alábbi típusokkal!

```
f :: Int -> Double
g :: Double -> Int
```

Tesztek:

```
f 8 == 4.0
f 2 == 2.0
f 3 < 2.5 && f 3 > 2.4
f 4 < 2.9 && f 4 > 2.8
g 8 == 4
g 2 == 2
g 3 == 2
g 4 == 3
```

2. feladat (⭐⭐): Add meg két síkbeli pont távolságát, de most legyen más a bemeneti paraméter típusa!

```
distanceInt :: (Int, Int) -> (Int, Int) -> Double
```

Tesztek:

```
distanceInt (2, 5) (5, 9) == 5.0
distanceInt (-4, 3) (1, -9) == 13.0
```

3. feladat (⭐): Add meg a logikai tagadás műveletét!

```
notb :: Bool -> Bool
```

Tesztek:

```
notb False == True
notb True  == False
```

4. feladat (⭐): Add meg a logikai konjunkció műveletét! Meg lehet adni a definíciót két egyenlettel (a standard `&&` függvény felhasználása nélkül)?

```
andb :: Bool -> Bool -> Bool
```

Tesztek:

```
andb True  False == False
andb True  True  == True
andb False False == False
andb False True  == False
```

5. feladat (⭐): Adj meg egy függvényt, amely a sorvége karaktert (`\n`) szóközre képezi (minden más karaktert pedig önmagára)!

```
swapEndLine :: Char -> Char
```

Tesztek:

```
swapEndLine '\n' == ' '
map swapEndLine ['a'..'z'] == ['a'..'z']
```

Gyakorló feladatok

1. feladat (⭐ ⭐): Definiálj egy egyszerű számológépet egy függvény segítségével. A függvény két szám között végezzen el egy aritmetikai műveletet, amelyet a 2., karakter típusú paraméter határoz meg (lásd tesztek)!

```
calc :: Int -> Char -> Int -> Int
```

Tesztek:

```
calc 1 '+' 2 == 3
calc 3 '+' 1 == 4
calc 1 '*' 1 == 1
calc 6 '*' 2 == 12
calc 1 '-' 3 == -2
calc 10 '-' 5 == 5
```

2. feladat (⭐ ⭐ ⭐): Hogyan lehetne az előző feladatban megadott számológépet kiegészíteni osztás (`/`) művelettel?

3. feladat (⭐): Add meg a logikai diszjunkció műveletét!

```
orb :: Bool -> Bool -> Bool
```

Tesztek:

```
orb False False == False
orb True True == True
orb False True == True
orb True False == True
```

4. feladat (⭐): Add meg a logikai "kizáró vagy" műveletét! Lehetséges egyetlen egyenlettel definiálni ezt a műveletet?

```
```haskell
xorb :: Bool -> Bool -> Bool
```

Tesztek:

```haskell
xorb False False == False
xorb True True == True
xorb False True == True
xorb True False == False
```
```

4-5. gyakorlat: Lista literálok, intervallumkifejezések, halmazkifejezések

Ezen gyakorlatok témája a listakifejezések bevezetése, listák felépítése, és a végtelen listák használata. A gyakorlatok fókuszában a halmazkifejezések és az intervallumkifejezések állnak, de szó esik néhány szabványkönyvtárbeli, listákat feldolgozó függvényről is (pl. `length`, `nub`, `take`, `drop`, `head`, `tail`), viszont ezeknek a definícióját későbbi gyakorlatokon tárgyaljuk. A gyakorlatokon különösen nagy hangsúlyt helyezünk arra, hogy mennyire változatos problémák oldhatóak meg halmazkifejezések segítségével.

1. feladat (⭐): Találd ki az alábbi kifejezések típusát (amennyiben típusozhatóak)!

- `[[]]`
- `[True, [False]]`
- `[[], [[]]]`
- `[[(True, 'a')], [(True, 'a'), (False, 'b')]]`

- `([], [])`

2. feladat (⭐): Adj meg egy függvényt, amely felsorolja az egész számokat az első paraméterétől a másodikig!

```
fromTo :: Int -> Int -> [Int]
```

Tesztek:

```
fromTo 1 5 == [1,2,3,4,5]
fromTo (-5) 0 == [-5,-4,-3,-2,-1,0]
fromTo 10 3 == []
```

3. feladat (⭐): Megoldható az előző feladat, ha a függvények paraméterei valamilyen lebegőpontos számtípusúak (pl. `Float`)?

4. feladat (⭐): Adj meg egy függvényt, amely felsorolja a páros egész számokat az első paraméterétől a másodikig!

```
evensBetween :: Int -> Int -> [Int]
```

Tesztek:

```
length (evensBetween 0 10) == 6
evensBetween 3 7 == [4,6]
evensBetween (-10) 0 == [-10,-8,-6,-4,-2,0]
```

5. feladat (⭐): Adj meg egy függvényt, amely egy lista összes páros elemét megduplázza, a páratlanokat pedig kiszűri!

```
doubleEvens :: [Int] -> [Int]
```

Tesztek:

```
doubleEvens [1,2,3,4,5,6] == [4,8,12]
all even (doubleEvens [0..1000])
```

6. feladat (⭐⭐): Adj meg egy függvényt, amely egy lista összes páros elemét megduplázza! (A páratlan számok módosítás nélkül kerüljenek be az eredménybe!)

```
doubleEvens2 :: [Int] -> [Int]
```

Tesztek:

```
doubleEvens2 [1,2,3,4,5,6] == [1,4,3,8,5,12]
not (all even (doubleEvens2 [0..1000]))
length (doubleEvens2 [1..1000]) == 1000
```

7. feladat (⭐): Hány páratlan eleme van egy listának?

```
odds :: [Int] -> Int
```

Tesztek:

```
odds [1..10] == 5
odds [2,4..1000] == 0
take 100 (odds [1,3..]) == 100
```

8. feladat (⭐): Add meg az összes óra-perc lehetséges kombinációt!

```
clock :: [(Int, Int)]
```

Tesztek:

```
all (`elem` clock) [(0,1),(2,1),(0,3),(0,5),(23,59),(0,0),
(10,43)]
not ((4, 60) `elem` clock)
not ((24, 6) `elem` clock)
```

9. feladat (⭐⭐): Add meg azt a legkisebb 2 hatványt halmazkifejezéssel, amelyik már nagyobb, mint 10^{20}!

```
smallest :: Integer
```

Tesztek:

```
smallest == 147573952589676412928
```

10. feladat (⭐): Döntsd el, hogy egy lista kizárólag pozitív elemeket tartalmaz-e!

```
allPositive :: [Int] -> Bool
```

Tesztek:

```
allPositive [1..100]
allPositive [100,99..1]
allPositive []
not (allPositive [0..100])
not (allPositive [10,4,7,1,-1,23,4])
```

11. feladat (⭐ ⭐): Sorold fel az összes prímszámot! Tipp: Ehhez a feladathoz érdemes egy segédfüggvényt definiálni, amely eldönti egy számról, hogy prím-e.

```
primes :: [Integer]
```

Tesztek:

```
take 10 primes == [1,2,3,5,7,11,13,17,19,23]
primes !! 100 == 541
primes !! 1000 == 7919
```

12. feladat (⭐): Adott egy kulcs-érték párokból álló asszociatív lista. Add meg az `1` kulcshoz tartozó értéket!

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

Tesztek:

```
ones [] == []
ones [(1,'a'),(2,'b'),(1,'l'),(4,'a'),(1,'m'),(1,'a')] ==
"alma"
ones [(10,'a'),(2,'b'),(3,'l'),(4,'a'),(15,'m'),(2,'a')] ==
""
```

Gyakorló feladatok

1. feladat (⭐): Állítd elő a 2 hatványait növekvő sorrendben `1`-től `2^10`-ig!

```
twos :: [Integer]
```

Tesztek:

```
last twos == 1024
length twos == 11
head twos == 1
```

2. feladat (⭐): Sorold fel az első 100 négyzetszám kétszeresét!

```
squares :: [Integer]
```

Tesztek:

```
take 3 squares == [2,8,18]
all (\x -> 2* (round (sqrt (fromIntegral x / 2)))^2 == x)
squares
length squares == 100
```

3. feladat (⭐⭐): Állítsd elő olyan 10 hosszúságú listát, amely váltakozva tartalmazza a `False` és `True` értékeket!

```
bools :: [Bool]
```

Tesztek:

```
length bools == 10
bools !! 0 == False
bools !! 1 == True
take 3 bools == [False, True, False]
bools !! 9 == True
```

4. feladat (⭐ ⭐ ⭐): Állítsd elő a következő listát: `[1,2,2,3,3,3,4,4,4,4,...]` ! Az i szám i-szer szerepeljen a listában!

```
incr :: [Int]
```

Tesztek:

```
all (\x -> length (filter (x==) (take 2000 incr)) == x)
[1..50]
```

5. feladat (⭐ ⭐): Adj meg egy szöveget tömörítő függvényt, amely karakter-darabszám párokként tárolja a szöveget, ahol a darabszám mondja meg, hogy az adott karakter hányszor szerepel egymás után. Alaposan nézd meg a teszteket, hogy pontosan miről is van szó! Tipp: használd a `group` függvényt!

```
compress :: String -> [(Int, Char)]
```

Tesztek:

```
compress "aaaabccaadeeee" == [(4,'a'), (1,'b'), (2,'c'),
(2,'a'), (1,'d'), (4,'e')]
compress "oh hello!!" == [(1,'o'),(1,'h'),(1,' '),(1,'h'),
(1,'e'),(2,'l'),(1,'o'),(2,'!')]
compress "" == []
```

6. feladat (⭐ ⭐): Add meg az előző feladatban megadott függvény inverzét! Azaz, adott egy lista karakter-darabszám párok formájában, és állítsd elő a reprezentált szöveget!

```
decompress :: [(Int, Char)] -> String
```

Tesztek:

```
decompress [(4,'a'), (1,'b'), (2,'c'), (2,'a'), (1,'d'),
(4,'e')] == "aaaabccaadeeee"
decompress [(1,'o'),(1,'h'),(1,' '),(1,'h'),(1,'e'),(2,'l'),
(1,'o'),(2,'!')] == "oh hello!!"
decompress [] == ""
```

Adatbázissal kapcsolatos feladatok - 1. rész

Adott egy kereskedés használtautó adatbázisa, amely márkánként tárolja, hogy az egyes autó típusokból (a feltüntetett üzemanyag/hajtáslánc mellett) hány áll rendezésre.

```
cars :: [(String, [(String, String, Int)])]
cars =
  [ ("KIA", [("Optima", "plug-in", 5), ("Soul", "electric", 2),
("CEE'D", "plug-in", 2), ("CEE'D", "gasoline", 12), ("XCEE'D",
"plug-in", 3)])
  , ("Renault", [("Zoe", "electric", 4), ("Megane", "plug-in", 2),
("Megane", "gasoline", 5), ("Captur", "plug-in", 1)])
  , ("Nissan", [("Leaf", "electric", 6), ("Qashqai", "mild-
hybrid", 1), ("X-Trail", "gasoline", 2)])
  , ("Opel", [("Corsa", "gasoline", 2), ("Corsa", "electric", 3)])
  , ("Toyota", [("RAV4", "hybrid", 4), ("Camry", "hybrid", 2),
("Corolla", "hybrid", 9), ("Prius", "plug-in", 3)])
  , ("Ssangyong", [("Korando", "diesel", 4), ("Rexton", "diesel",
2)])
  , ("Mazda", [("MX-30", "electric", 1), ("3", "mild-hybrid", 2),
("6", "gasoline", 1)])
  ]
```

1. feladat (⭐): Add meg, hogy hány elektromos autót forgalmaz a kereskedés!

   ```
   countECars :: [(String, [(String, String, Int)])] -> Int
   ```

   Tesztek:

   ```
   countECars cars == 16
   ```

2. feladat (⭐): Add meg, hogy összesen hány autót forgalmaz a kereskedés!

   ```
   countCars :: [(String, [(String, String, Int)])] -> Int
   ```

   Tesztek:

   ```
   countCars cars == 78
   ```

3. feladat (⭐⭐): Add meg, hogy hány olyan márka van, amelyből legalább 6 autója van a kereskedésnek!

   ```
   atLeast6 :: [(String, [(String, String, Int)])] -> [String]
   ```

   Tesztek:

```
atLeast6 cars ==
["KIA","Renault","Nissan","Toyota","Ssangyong"]
```

4. feladat (⭐): Add meg, hány `"plug-in"` autót forgalmaznak!

```
plugIns :: [(String, [(String, String, Int)])] -> Int
```

Tesztek:

```
plugIns cars == 16
```

5. feladat (⭐⭐): Add meg, hogy hány `'C'` karakterrel kezdődő típusú (nem márkájú!) autót forgalmaznak!

```
ctypes :: [(String, [(String, String, Int)])] -> Int
```

Tesztek:

```
ctypes cars == 31
```

6. feladat (⭐ ⭐): Add meg, hogy márkánként hány autót forgalmaz a kereskedés!

```
carsPerBrand :: [(String, [(String, String, Int)])] ->
[(String, Int)]
```

Tesztek:

```
carsPerBrand cars == [("KIA",24),("Renault",12),("Nissan",9),
("Opel",5),("Toyota",18),("Ssangyong",6),("Mazda",4)]
```

7. feladat (⭐⭐⭐): Adott egy környezetvédelmi mutató az üzemanyagtól függően (`"diesel": 6, "gasoline": 5, "mild-hybrid": 4, "hybrid": 3, "plug-in": 2, "electric": 1`, tipp: erre felírhatsz egy segédfüggvényt). Add meg márkánként, hogy mi a környezetvédelmi összeg (azaz minden autótípusnál az üzemanyagnak megfelelően szorozd össze a darabszámot a fenti mutatóval)!

```
greennessPerBrand :: [(String, [(String, String, Int)])] ->
[(String, Int)]
```

Tesztek:

```
greennessPerBrand cars  == [("KIA",82),("Renault",35),
("Nissan",20),("Opel",13),("Toyota",51),("Ssangyong",36),
("Mazda",14)]
```

8. feladat (⭐⭐): Hány különböző autótípust forgalmaz a kereskedés?

```
differentTypes :: [(String, [(String, String, Int)])] -> Int
```

Tesztek:

```
differentTypes cars == 20
```

Adatbázissal kapcsolatos feladatok - 2. rész

Adott egy használtautó-kereskedés adatbázisa, amely márkánként csoportosítva tartalmazza a forgalmazott autókat, típusukat, üzemanyagukat, és árukat feltüntetve. Az autók ára millió Ft-ban értendő.

```
carPrices :: [(String, [(String, String, Double)])]
carPrices =
  [ ("KIA", [("Optima", "plug-in", 6.7), ("Soul", "electric",
4.5), ("CEE'D", "plug-in", 8.3), ("CEE'D", "gasoline", 6),
("XCEE'D", "plug-in", 7)])
  , ("Renault", [("Zoe", "electric", 4.2), ("Megane", "plug-in",
8.6), ("Megane", "gasoline", 6.1), ("Captur", "plug-in", 7.2),
("R-5", "electric", 13)])
  , ("Nissan", [("Leaf", "electric", 6.2), ("Qashqai", "mild-
hybrid", 5.3), ("X-Trail", "gasoline", 7.1)])
  , ("Opel", [("Corsa", "gasoline", 4.3), ("Corsa", "electric",
6.2)])
  , ("Toyota", [("RAV4", "hybrid", 10.1), ("RAV4", "plug-in",
8.6), ("Camry", "hybrid", 12.3), ("Corolla", "hybrid", 6.4),
("Prius", "plug-in", 4.5)])
  , ("Ssangyong", [("Korando", "diesel", 7.5), ("Rexton",
"diesel", 6.8)])
  , ("Mazda", [("MX-30", "electric", 6.5), ("3", "mild-hybrid",
7.1), ("6", "gasoline", 11.2)])
  ]
```

A következő három feladatban az eredményben szereplő autókat "<márka>: <típus>" alakú szövegként add meg!

1. feladat (⭐): Ha valaki maximum 7 millió forint értékben keres elektromos (`"electric"`) autót, milyen lehetőségei vannak?

```
electric7 :: [(String, [(String, String, Double)])] ->
[String]
```

Tesztek:

```
electric7 carPrices ==
["KIA:Soul","Renault:Zoe","Nissan:Leaf","Opel:Corsa","Mazda:M
X-30"]
```

2. feladat (⭐): Melyik autó a kereskedés legdrágább plug-in hibrid (`"plug-in"`) autója?

```
mostExpensive :: [(String, [(String, String, Double)])] ->
String
```

Tesztek:

```
mostExpensive carPrices == "Renault:Megane" || mostExpensive
carPrices == "Toyota:RAV4"
```

3. feladat (⭐ ⭐ ⭐): Tegyük fel, hogy a következő évben a benzin (`gasoline`) ára rögzített `594` Ft/l, és a dízel ára `606` Ft/l. Adottak a következő fogyasztási becslések:

```
"gasoline" -> 7.5 liter/100km (benzin)
"plug-in" -> 4 liter/100km (benzin)
"hybrid" -> 5 liter/100km (benzin)
"diesel" -> 6.5 liter/100km (dízel)
"mild-hybrid" -> 7 liter/100km (benzin)
"electric" -> 0 liter/100km
```

Továbbá adottak a becsült éves szervízköltségek (millió Ft-ban):

```
"gasoline" -> 0.8
"plug-in" -> 1.2
"hybrid" -> 1.1
"diesel" -> 0.9
"mild-hybrid" -> 1
"electric" -> 1.4
```

Tegyük fel, hogy egy vásárló havonta 300 km-t utazik (tehát évente 3600 km-t). Melyik autót tudná legalacsonyabb költségen megvenni 1 évre számolva? A költséget a vételár, éves üzemanyag és éves szervízköltség adja ki. Az eredményt "<márka>:<típus>-<üzemanyag>" formában jelenítsd meg!

```
mostExpensive :: [(String, [(String, String, Double)])] ->
String
```

Tesztek:

```
mostEconomic carPrices == "Opel:Corsa-gasoline"
```

Adatbázissal kapcsolatos feladatok - 3. rész

Adott egy tánciskola adatbázisa, amely táncstílusonként tartalmazza, hogy kik vannak az adott stílusra beiratkozva. Egy táncosról a következő adatokat tárolja az alábbi lista: vezetéknév, keresztnév, keres-e táncpartnert, és a táncos neme.

```
dancers :: [(String, [(String, String, Bool, String)])]
dancers = [("salsa", [("Kiss", "Samuel", True, "ferfi"),
                      ("Kiralyi", "Evelin", True, "no"),
                      ("Kovacs", "Bela", False, "ferfi"),
                      ("Nemeth", "Csilla", True, "no"),
                      ("Solymos", "Klara", True, "no"),
                      ("Mars", "Bruno", False, "ferfi"),
                      ("Kovacs", "Agoston", False, "ferfi"),
                      ("Nemeth", "Balazs", True, "ferfi"),
                      ("Eper", "Alma", True, "no")]),
           ("tarsastanc",
                     [("Nagy", "Salamon", False, "ferfi"),
                      ("Janosi", "Janos", False, "ferfi"),
                      ("Elek", "Tibor", False, "ferfi"),
                      ("Nemeth", "Csilla", True, "no"),
                      ("Nagy", "Xenia", True, "no"),
                      ("PO", "C3", True, "01100110 01100101
01110010 01100110 01101001"),
                      ("Solymos", "Klara", True, "no"),
                      ("Nemeth", "Balazs", False, "ferfi")]),
           ("west-coast swing",
                     [("Kis", "Sandor", True, "ferfi"),
                      ("Moricz", "Zsigmond", True, "ferfi"),
```

```
                        ("Elek", "Tibor", True, "ferfi"),
                        ("Nemeth", "Csilla", True, "no"),
                        ("Nagy", "Sandor", True, "ferfi"),
                        ("PO", "C3", True, "01100110 01100101
01110010 01100110 01101001"),
                        ("Solymos", "Klara", True, "no"),
                        ("Nemeth", "Balazs", True, "ferfi")])
          ]
```

1. feladat (⭐): Add meg azokat a nőket, akik társastánchoz keresnek partnert!

```
ballroomWomen :: [(String, [(String, String, Bool, String)])]
-> [String]
```

Tesztek:

```
ballroomWomen dancers == ["Nemeth Csilla","Nagy
Xenia","Solymos Klara"]
```

2. feladat (⭐): Add meg azokat a férfiakat, akik társastáncolnak!

```
ballroomMen :: [(String, [(String, String, Bool, String)])] -
> [String]
```

Tesztek:

```
ballroomMen dancers == ["Nagy Salamon","Janosi Janos","Elek
Tibor","Nemeth Balazs"]
```

3. feladat (⭐⭐): Kik azok, akik legalább 2 táncstílusban táncolnak?

```
atLeast2Styles :: [(String, [(String, String, Bool,
String)])] -> [String]
```

Tesztek:

```
atLeast2Styles dancers == ["Nemeth Csilla","Solymos
Klara","Nemeth Balazs"]
```

4. feladat (⭐⭐⭐): Add meg hány nő táncol legalább három stílust!

```
atLeast3Styles :: [(String, [(String, String, Bool,
String)])] -> Int
```

Tesztek:

```
atLeast3Styles dancers == 0
atLeast3Styles (dancers ++ [("bachata", [("Solymos", "Klara",
True, "no")])]) == 1
```

5. feladat (🔥): Add meg azokat a táncosokat, akiknek ugyanaz a vezetékneve!

```
sameName :: [(String, [(String, String, Bool, String)])] ->
[String]
```

Tesztek:

```
sameName dancers == ["Kovacs Bela","Kovacs Agoston","Nemeth
Csilla","Nemeth Balazs","Nagy Salamon","Nagy Xenia","Nagy
Sandor"]
```

6. feladat (⭐ ⭐): Van-e olyan táncstílus, amelyben mindenki partnert keres?

```
allLookingForPartner :: [(String, [(String, String, Bool,
String)])] -> Bool
```

Tesztek:

```
allLookingForPartner dancers
not (allLookingForPartner (init dancers))
```

7. feladat (🔥): Készítsd el egy táncos társkereső alkalmazás alapját! Minden táncpartnert kereső férfi nevéhez rendeld hozzá azon partnert kereső nőket (a nevüket), akik legalább egy közös stílust táncolnak az adott férfival!

```
findPartners :: [(String, [(String, String, Bool, String)])]
-> [(String, [String])]
```

Tesztek:

```
findPartners dancers == [("Kiss Samuel",["Kiralyi
Evelin","Nemeth Csilla","Solymos Klara","Eper Alma"]),
("Nemeth Balazs",["Kiralyi Evelin","Nemeth Csilla","Solymos
Klara","Eper Alma"]),("Kis Sandor",["Nemeth Csilla","Solymos
Klara"]),("Moricz Zsigmond",["Nemeth Csilla","Solymos
Klara"]),("Elek Tibor",["Nemeth Csilla","Solymos Klara"]),
("Nagy Sandor",["Nemeth Csilla","Solymos Klara"])]
```

6. gyakorlat - Rekurzió

Ezen a gyakorlaton a rekurziót vezetjük be, számtípusokkal kapcsolatos, jól ismert függvények megadásával. Megbeszéljük, hogy a rekurzív függvényalkalmazás is "csak" egyszerűsítés (behelyettesítések/beta-redukciók sorozata), illetve néhány függvényt megadunk végrekurzív módon is.

Alapvető feladatok

1. feladat (⭐): Definiáld egy függvényt, amely kiszámítja egy szám faktoriálisát!

```
fact :: Integer -> Integer
```

Tesztek:

```
fact 0 == 1
fact 1 == 1
fact 2 == 2
fact 3 == 6
all (\x -> fact x == product [1..x]) [1..30]
```

2. feladat (⭐): Definiálj egy függvényt, amely megadja az n-edik Fibonacci számot!

```
fib :: Integer -> Integer
```

Tesztek:

```
fib 0 == 0
fib 1 == 1
fib 2 == 1
fib 10 == 55
fib 20 == 6765
```

3. feladat (⭐): Definiáld a hatványozás függvényt szorzásra alapozva!

```
pow :: (Num b, Integral a) => b -> a -> b
```

Tesztek:

```
let xs = [ (x, y) | x <- [1..10], y <- [1..10]] in map
(uncurry (^)) xs == map (uncurry pow) xs
```

4. feladat (⭐): Definiálj egy függvényt, amely megadja két szám legnagyobb közös osztóját!

```
gcdiv :: Integral a => a -> a -> a
```

Tesztek:

```
all (\(x, y) -> gcdiv x y == gcd x y) [(x, y)| x <- [3..150],
y <- [40..230]]
```

Gyakorló feladatok

1. feladat (⭐): Definiáld egész számok szorzását az összeadásra műveletre alapozva! Negatív számokra is működjön a megoldás!

```
mul :: Integral a => a -> a -> a
```

Tesztek:

```
let xs = [ (x, y) | x <- [-10..10], y <- [-10..10]] in map
(uncurry (*)) xs == map (uncurry mul) xs
```

2. feladat (⭐⭐): Definiáld a `<=` relációt természetes számokon (`Int`-et használva) kizárólag `==`-re és kivonásra alapozva!

```
le :: Int -> Int -> Bool
```

Tesztek:

```
le 3 3
le 0 0
le 3 5
not (le 5 1)
not (le 10 1)
```

3. feladat (⭐⭐): Add meg a Fibonacci számsort! Használj végtelen listát!

```
fibs :: [Integer]
```

Tesztek:

```
take 10 fibs == [0,1,1,2,3,5,8,13,21,34]
```

4. feladat (⭐⭐⭐): Add meg a faktoriálist és az n-edik Fibonacci számot kiszámító függvényeket végrekurzívan! Hasonlítsd össze a gyakorlaton megadott (nem végrekurzív) függvénnyel a hatékonyságát!

7. gyakorlat - Listák bevezető, rekurzió

Ezen a gyakorlaton visszatérünk a listák tárgyalásához, megmutatjuk, hogy listákat kétféleképpen lehet létrehozni (egy lista vagy üres, vagy van legalább egy eleme és egy maradéklistája). Ezután listákat feldolgozó függvényeket adunk meg rekurzívan. Fontos megjegyezni, hogy az itt felsorolt függvények közül több a szabványkönyvtárban már létezik, ezeknek a neve `List` szuffixxel lett ellátva.

Alapvető feladatok

1. feladat (⭐): Definiálj egy függvényt, amely megadja egy lista fejelemét!

```
headList :: [a] -> a
```

Tesztek:

```
headList [1..100] == 1
headList [-10] == -10
headList [1..] == 1
```

2. feladat (⭐): Definiálj egy függvényt, amely eldobja egy lista fejelemét (azaz, megadja egy lista "maradékát")!

```
tailList :: [a] -> [a]
```

Tesztek:

```
tailList [1..100] == [2..100]
tailList [-10] == []
```

3. feladat (⭐): Döntsd el egy listáról, hogy üres-e!

```
nullList :: [a] -> Bool
```

Tesztek:

```
nullList []
not (nullList [1..10])
not (nullList [10])
not (nullList [1..])
```

4. feladat (⭐): Döntsd el egy listáról, hogy kételemű-e!

```
contains2Elems :: [a] -> Bool
```

Tesztek:

```
contains2Elems [1,2]
contains2Elems "as"
not (contains2Elems ['a'..'z'])
not (contains2Elems [1..])
```

5. feladat (⭐⭐): Döntsd el egy listáról, hogy 139 elemű-e! Tipp: itt már érdemes rekurziót használni esetszétválasztás (mintaillesztés) helyett!

```
contains139Elems :: [a] -> Bool
```

Tesztek:

```
not (contains139Elems [])
contains139Elems [1..139]
not (contains139Elems [1..])
```

6. feladat (⭐): Definiálj egy függvényt, amely megadja egy lista első `n` elemét!

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

Tesztek:

```
takeList 10 [1..10] == [1..10]
takeList 100 [1..1000] == [1..100]
takeList 10 [1..5] == [1..5]
takeList 10 [1..] == [1..10]
```

7. feladat (⭐): Fűzz össze két listát elemenként! Az eredmény i-edik eleme a két paraméterlista i-edik elemeinek párja legyen.

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

Tesztek:

```
zipList [] [] == []
zipList [1..5] ['a'..'e'] == [(1,'a'),(2,'b'),(3,'c'),
(4,'d'),(5,'e')]
zipList [1..] [1..5] == [(1,1),(2,2),(3,3),(4,4),(5,5)]
zipList [1..5] [1..] == [(1,1),(2,2),(3,3),(4,4),(5,5)]
```

8. feladat (⭐): Add meg egy lista minden 3. elemét!

```
everyThird :: [a] -> [a]
```

Tesztek:

```
everyThird [1..10] == [3,6,9]
everyThird [1,2] == []
everyThird [1] == []
everyThird [] == []
```

9. feladat (⭐): Add össze egy számokat tartalmazó lista elemeit!

```
sumList :: Num a => [a] -> a
```

Tesztek:

```
sumList [1..10] == 55
sumList [] == 0
sumList [1.1,1.2] == 2.3
```

10. feladat (⭐⭐): Válogass szét egy listát az elemek paritása szerint! A páros elemeket az eredmény pár első, a páratlanokat a második listájában gyűjtsd!

```
oddEven :: [Int] -> ([Int], [Int])
```

Tesztek:

```
oddEven [1..10] == ([2,4..10],[1,3..9])
oddEven [2,4..100] == ([2,4..100], [])
oddEven [1,3..100] == ([], [1,3..100])
```

11. feladat (⭐⭐): Fűzz össze két listát rendezett módon! Mindkét listában az elemek egymáshoz viszonyított sorrendje ne változzon az eredményben!

```
merge :: Ord a => [a] -> [a] -> [a]
```

Tesztek:

```
merge [1,2,3] [4,5,6] == [1..6]
merge [6,2,8,1] [3,7,1] == [3,6,2,7,1,8,1]
merge [] [] == []
```

Gyakorló feladatok

1. feladat (⭐): Fűzz össze három listát elemenként! Az eredmény i-edik eleme a három paraméterlista i-edik elemeinek hármasa legyen.

```
zip3List :: [a] -> [b] -> [c] -> [(a, b, c)]
```

Tesztek:

```
zip3List [] [] [] == []
zip3List [1..5] ['a'..'e'] [10..15] == [(1,'a',10),
(2,'b',11),(3,'c',12),(4,'d',13),(5,'e',14)]
zip3List [1..] [1..5] [1..5] == [(1,1,1),(2,2,2),(3,3,3),
(4,4,4),(5,5,5)]
zip3List [1..5] [1..] [1..] == [(1,1,1),(2,2,2),(3,3,3),
(4,4,4),(5,5,5)]
```

2. feladat (⭐): Definiálj egy olyan függvényt, amely előállít egy listát, amely a második paraméterét annyiszor tartalmazza, amennyi az első paramétere volt!

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

Tesztek:

```
replicateList 5 'a' == "aaaaa"
replicateList 10 'p' == "pppppppppp"
replicateList 2 2 == [2,2]
```

3. feladat (⭐): Add meg egy `Bool` értékeket tartalmazó lista össze-"és"-elésének eredményét!

```
andList :: [Bool] -> Bool
```

Tesztek:

```
andList [True, True, True, True]
andList []
not (andList [True, False, True, True])
not (andList [False, False, False])
```

4. feladat (⭐⭐): Add meg azt a függvényt, amely eldönti, hogy egy érték eleme-e egy listának!

```
elemList :: Eq a => a -> [a] -> Bool
```

Tesztek:

```
elemList 4 [1,2,3,4]
elemList 'c' "cica"
not (elemList 'c' "almafa")
not (elemList 3 [])
```

5. feladat (⭐⭐): Adj meg egy függvényt, amely kiválasztja egy lista legnagyobb elemét! Üres lista esetén adj futásidejű hibát (pl. `undefined`, `error` függvényekkel)! Egyelemű lista esetén az egyetlen elem legyen a maximum! Használd a `max` függvényt két érték közül a nagyobbik kiválasztására.

```
maximumList :: Ord a => [a] -> a
```

Tesztek:

```
maximumList [1..10] == 10
maximumList [10,9..1] == 10
maximumList [1,2,7,1,4,2,6] == 7
```

6. feladat (⭐⭐): Adj meg egy függvényt, amely egy kulcs-érték párokat tartalmazó listából megadja az adott kulcshoz tartozó értéket! Adott a függvény paramétereként egy alapértelmezett érték is, ha nem szerepel a kulcs a listában, ezt az értéket add vissza! Hogyan módosulna a függvényszignatúra, illetve a megoldás, ha feltehető, hogy a lista kulcs szerint (növekvően) rendezett?

```
getValue :: Eq a => a -> [(a, b)] -> b -> b
```

Tesztek:

```
getValue 2 [(1, "asd"),(2, "vbn"),(3, "f")] "default" ==
"vbn"
getValue 4 [(1, "asd"),(2, "vbn"),(3, "f")] "default" ==
"default"
```

7. feladat (⭐⭐): Adj meg egy függvényt, amely egy rendezett listába beszúr egy értéket a rendezettséget megtartva!

```
insert :: Ord t => t -> [t] -> [t]
```

Tesztek:

```
insert 2 [4, 5, 9] == [2, 4, 5, 9]
insert 6 [4, 5, 9] == [4, 5, 6, 9]
insert 11 [4, 5, 9] == [4, 5, 9, 11]
insert 5 [4, 5, 9] == [4, 5, 5, 9]
insert 2 [] == [2]
```

8. feladat (⭐⭐): Add meg azt a függvényt, amely a beszúró rendezés algoritmusa alapján visszaadja a paraméter lista rendezett változatát! Tipp: egy üres listába szúrj be egyesével az elemeket a rendezettség szerint megfelelő helyre. Használhatod az előbbi insert függvényt is akár.

```
sortList :: Ord t => [t] -> [t]
```

Tesztek:

```
sortList [5, -2, 3] == [-2, 3, 5]
sortList "almafa" == "aaaflm"
sortList [] == []
```

9. feladat (⭐⭐): Add meg azt a függvényt, amely az összefésülő rendezés algoritmusa alapján visszaadja a paraméter lista rendezett változatát! Tipp: használd a gyakorlaton definiált `merge` függvényt!

```
mergeSort :: Ord t => [t] -> [t]
```

Tesztek:

```
mergeSort [5, 2, -4, 9, 3] == [-4, 2, 3, 5, 9]
mergeSort "almafa" == "aaaflm"
mergeSort [] == []
```

10. feladat (⭐⭐): Definiálj egy függvényt, mely egy listában egy megadott érték előtti és utáni darabot visszaadja (a második rész tartalmazza a "vágás" karakterét)!

```
breakOn :: Eq a => a -> [a] -> ([a], [a])
```

Tesztek:

```
breakOn ' ' "haskell is cool" == ("haskell", " is cool")
breakOn ' ' "is cool" == ("is", " cool")
breakOn '/' "haskell/gyak/gyak.hs" == ("haskell",
"/gyak/gyak.hs")
```

11. feladat (⭐⭐⭐): Definiálj egy függvényt, mely adott érték mentén darabol egy listát! Tipp: használd a `breakOn` függvényt az előző feladatból!

```
splitOn :: Eq a => a -> [a] -> [[a]]
```

Tesztek:

```
splitOn ' ' "haskell is cool" == ["haskell", "is", "cool"]
splitOn '/' "haskell/gyak/gyak.hs" == ["haskell", "gyak",
"gyak.hs"]
splitOn '/' "haskell" == ["haskell"]
splitOn '/' "" == []
```

8. gyakorlat - Listák, map, filter

Ezen a gyakorlaton két híres magasabbrendű függvénnyel foglalkozunk: `map` (listák transzformálására) és `filter` (listaelemek szűrésére). Gyakorlásként javasoljuk ezen feladatok megoldását rekurzióval is, illetve az 5. gyakorlat feladatainak megoldását `map`-pal és `filter`-rel.

Alapvető feladatok

1. feladat (⭐): Add meg egy lista páratlan elemeinek számát!

```
oddCount :: [Int] -> Int
```

Tesztek:

```
oddCount [1..10] == 5
oddCount [2,4..100] == 0
oddCount [1,3..100] == 50
```

2. feladat (⭐): Add meg egy szöveg azon szavait, amelyek számmal kezdődnek!

```
startsWithNumber :: String -> [String]
```

Tesztek:

```
startsWithNumber "1word 2 3 4 5" == ["1word","2","3","4","5"]
startsWithNumber "Thi0s sente23nce do4es n2ot cont1ain
numbe3rs" == []
startsWithNumber "wor1d 1word wo4rd" == ["1word"]
```

3. feladat (⭐⭐): (Újra) Add meg egy lista minden 3. elemét!
4. feladat (⭐): Duplázz meg minden páros elemet egy listában!

```
doubleEven :: [Int] -> [Int]
```

Tesztek:

```
doubleEven [1..10] == [1,4,3,8,5,12,7,16,9,20]
doubleEven [1,3..100] == [1,3..100]
doubleEven [2,4..100] == [4,8..200]
```

5. feladat (⭐): Add meg, hogy egy karakterkódokat tartalmazó lista milyen szöveget reprezentál!

```
toString :: [Int] -> String
```

Tesztek:

```
toString [97..122] == ['a'..'z']
toString [] == []
```

6. feladat (⭐): (Újra) Add meg egy asszociatív listában az 1-es kulcshoz tartozó értékeket!

Gyakorló feladatok

1. feladat (⭐): Definiálj egy függvényt, amely minden nagybetűt kisbetűvé alakít, a többit eldobja! Tipp: használd a `Data.Char` modul függvényeit!

```
upperToLower :: String -> String
```

Tesztek:

```
upperToLower "" == ""
upperToLower "Hello World!" == "hw"
upperToLower "haSKell" == "sk"
```

2. feladat (⭐ ⭐): Definiálj egy függvényt, amely eldönti, hogy egy predikátum teljesül-e egy lista minden elemére!

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

Tesztek:

```
allList (\n -> n > 0) []
allList (\n -> n > 0) [1,2,9,6]
not (allList (\n -> n > 0) [1,2,-4,9,6])
not (allList even [4,6,8,2,3,0])
```

3. feladat (⭐ ⭐): Definiálj egy függvényt, amely eldönti, hogy egy predikátum teljesül-e egy lista legalább egy elemére!

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

Tesztek:

```
not (anyList (\n -> n == 2) [])
not (anyList (\n -> n == 2) [1,5,9,0,3])
anyList (\n -> n == 2) [1,5,9,2,0,3,2]
anyList even [1,2,5,9,2,0,3,2]
```

9. gyakorlat - Listák, fold, egyéb magasabbrendű függvények

A listák témakörét a hajtogató függvények (elsősorban `foldr`) gyakorlásával zárjuk le. Ezen a gyakorlaton a hajtogatások tárgyalása mellett további, magasabbrendű függvényeket is definiálunk. Javasoljuk a korábbi feladatok közül az 5. és 8 gyakorlatok feladatainak megoldását valamilyen hajtogatási módszert használva.

Alapvető feladatok

1. feladat (⭐): (Újra) Összegezd egy lista elemeit!
2. feladat (⭐): Összegezd egy lista páros elemeit!

```
sumEvens :: Integral a => [a] -> a
```

Tesztek:

```
sumEvens [1..10] == 30
sumEvens [] == 0
```

3. feladat (⭐): Add meg egy lista hosszát!

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

Tesztek:

```
lengthList [1..100] == 100
lengthList [] == 0
```

4. feladat (⭐⭐): (Újra) Válogasd szét egy listának az elemeit paritásuk alapján!
5. feladat (⭐⭐): Definiálj egy függvényt, amely megfordít egy listát! Próbáld meg a feladatot hajtogatással megoldani!

```
reverseList :: [a] -> [a]
```

Tesztek:

```
reverseList [1..10] == [10,9..1]
reverseList "gezakekazeg" == "gezakekazeg"
reverseList [] == []
```

Gyakorló feladatok

1. feladat (⭐): Definiáld a függvényt, amely előállít egy érték és egy függvény alapján egy végtelen listát a következő módon: a lista első eleme legyen a megadott érték, és minden további elemet úgy határozunk meg, hogy az előző elemre alkalmazzuk a paraméter függvényt.

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

Tesztek:

```
take 5 (iterateList (\n -> n * 2) 1) == [1, 2, 4, 8, 16]
take 5 (iterateList (\n -> n * 4) 1) == [1, 4, 16, 64, 256]
```

2. feladat (⭐): Definiáld azt a függvényt, amely két listát egy művelet segítségével aggregál! Tipp: ha a művelet pont a párképzés (`\x y -> (x, y)` vagy rövidebben `(,)`), akkor ez a függvény ekvivalens a `zip` függvénnyel.

```
zipWithList :: (a -> b -> c) -> [a] -> [b] -> [c]
```

Tesztek:

```
zipWithList min [1,9,2,5] [5,0,3,8] == [1,0,2,5]
zipWithList min [1,0,3] [5,2,10,1] == [1,0,3]
zipWithList (*) [2,0,6] [1,5,4,9] == [2,0,24]
```

3. feladat (⭐): Definiáld az órán tanult `map` függvényt `foldr` segítségével!

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

Tesztek:

```
mapFoldr (\n -> n + 2) [] == []
mapFoldr (\n -> n + 2) [2,3,4] == [4,5,6]
mapFoldr even [2,3,4] == [True, False, True]
```

4. feladat (⭐): Definiáld az órán tanult `filter` függvényt `foldr` segítségével!

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

Tesztek:

```
filterFoldr (\n -> n > 5) [] == []
filterFoldr (\n -> n > 5) [1,2,5,6,0] == [6]
filterFoldr even [1,2,5,6,0] == [2,6,0]
filterFoldr (elem 0) [[5,6],[4,1,2,0],[0,5]] == [[4,1,2,0],
[0,5]]
```

5. feladat (⭐⭐): Definiálj egy függvényt, amely visszaadja egy listának azon prefixét, amely minden elemére teljesül a paraméter predikátum!

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

Tesztek:

```
takeWhileList (\n -> n > 5) [] == []
takeWhileList (\n -> n > 5) [6,7,9,5,2,1] == [6,7,9]
takeWhileList odd [6,7,9,5,2,1] == []
takeWhileList odd [7,9,5,2,1] == [7,9,5]
```

6. feladat (⭐⭐): Definiálj egy függvényt, amely visszaadja egy listának azon szuffixét, amely előtti listarészben minden elemre teljesül a paraméter predikátum!

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

Tesztek:

```
dropWhileList (\n -> n > 5) [] == []
dropWhileList (\n -> n > 5) [6,7,9,5,2,1] == [5,2,1]
dropWhileList odd [6,7,9,5,2,1] == [6,7,9,5,2,1]
dropWhileList odd [7,9,5,2,1] == [2,1]
```

7. feladat (⭐ ⭐): Definiálj egy függvényt, amely egy lista elemeit átalakítva számokká egy függvény segítségével összeadja azokat. A listáról feltehető, hogy véges!

```
sumWith :: Num b => (a -> b) -> [a] -> b
```

```
sumWith (\x -> length x) ["apple", "pear"] == 9
sumWith (\x -> x + 1) [1,2,4] == 10
sumWith (\x -> x) [1,2,3] == 6
sumWith (\_ -> 1) [1,2,3] == 3
```

8. feladat (⭐ ⭐): Számítsd ki egy együtthatóival megadott polinom helyettesítési értékét a megadott helyen! Pl. `[3,1,2,7]` az $3 + x + 2x^2 + 7x^7$ polinomot reprezentálja. Tipp: Használj Horner-sémát!

```
polinom :: Num a => [a] -> a -> a
```

Tesztek:

```
polinom [3,1,2,7] 4 == 487
polinom [] 10 == 0
polinom [1,1,1,1] 1 == 4
polinom [-10,2,-1,2] 3 == 41
```

9. feladat (⭐⭐): Definiálj egy függvényt, amely megadja egy nemüres lista maximumát a paraméterként megadott rendezési reláció szerint!

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

Tesztek:

```
maximumByList (<) [1..10] == 1
maximumByList (>) [1..10] == 10
maximumByList (\x y -> abs x > abs y) [2,4,-10,3,4] == -10
```

> **Megjegyzés**
>
> A `maximumBy` egy standard függvény, viszont a szignatúrája minimálisan eltér az itt megadott függvényétől.

10. gyakorlat - Egyszerű ADT-k

Ezen a gyakorlaton egyszerű algebrai (felsorolási és konstruktorparaméterrel rendelkező) adattípusokat definiálunk, továbbá ezen a gyakorlaton részletesen tárgyaljuk a mintaillesztés fogalmát. Végül két típusosztályról (`Eq`, `Show`) is röviden szó esik.

Oldd meg az adatbázisokkal kapcsolatos feladatokat `String`-ek helyett a következő adattípusok használatával:

```
data Dance = Social | Salsa | WestCoast | Bachata deriving (Eq,
Show)
data Sex = Male | Female deriving (Show)
data Fuel = PlugIn | Gasoline | Diesel | Electric | MildHybrid |
Hybrid deriving (Eq, Show)
```

Alapvető feladatok

1. feladat (⭐): Döntsd el egy meghajtás típusról (`Fuel`), hogy tartalmaz-e hibrid jegyeket (azaz elektromos és belsőégésű meghajtást is)!

   ```
   isHybrid :: Fuel -> Bool
   ```

   Tesztek:

   ```
   all isHybrid [PlugIn, Hybrid, MildHybrid]
   all (not . isHybrid) [Gasoline, Diesel, Electric]
   ```

2. feladat (⭐): Definiálj egy típust (`Time`), amely időt reprezentál óra-perc formátumban! A típus egy konstruktorral rendelkezzen (`T`), és az órákat, illetve a perceket is `Int`-tel reprezentálja! Add meg a típushoz a `Show` és `Eq` típusosztályok példányait a lenti tesztek alapján! Két időpont akkor tekinthető egyenlőnek, ha az időpontok órái 24-gyel, a percei pedig 60-nal maradékosan osztva megegyezik!

   Tesztek:

```
show (T 10 20) == "10:20"
show (T 33 88) == "33:88"
T 10 10 == T 10 10
T 24 60 == T 0 0
T 12 24 /= T 24 12
T 12 30 /= T 44 30
T 10 30 == T 10 30
T 10 30 /= T 10 40
T 10 30 /= T 11 30
T 10 30 /= T 9 30
```

3. feladat (⭐): Mutasd meg, hogy a `Time` típus izomorf az `(Int,Int)` típussal! Azaz, definiálj két függvényt, amelyek egymásra képezik ezt a két típust úgy, hogy a függvények kompozíciója identitás!

```
fTime :: Time -> (Int, Int)
gTime :: (Int, Int) -> Time
```

Tesztek:

```
zip [0..23] [0..59] == map (fTime . gTime) (zip [0..23]
[0..59])
zipWith T [0..23] [0..59] == map (gTime . fTime) (zipWith T
[0..23] [0..59])
```

Gyakorló feladatok

1. feladat (⭐): Definiáld azt a függvényt, amely eldönti, hogy egy időpont korábban van-e, mint egy másik!

```
isEarlier :: Time -> Time -> Bool
```

Tesztek:

```
isEarlier (T 10 15) (T 10 30)
isEarlier (T 10 35) (T 12 30)
not (isEarlier (T 12 30) (T 12 30))
not (isEarlier (T 10 35) (T 10 30))
not (isEarlier (T 13 30) (T 12 30))
```

2. feladat (⭐ ⭐): Adj meg egy "smart" konstruktort a `Time` típushoz, amely csak értelmezhető formátumú időértékek létrehozására képes! Tipp: órák 0-23 és percek 0-59 intervallumokban vesznek fel valid értékeket. A `Maybe` típust részletesen a következő gyakorlaton tárgyaljuk.

```
time :: Int -> Int -> Maybe Time
```

Tesztek:

```
time 12 60 == Nothing
time 24 15 == Nothing
time (-5) 2 == Nothing
time 15 (-10) == Nothing
(time 12 30) == Just (T 12 30)
(time 23 59) == Just (T 23 59)
(time 00 00) == Just (T 00 00)
not ((time 12 30) == Just (T 20 00))
```

3. feladat (⭐): Definiáld elsőként a `Month` felsorolási algebrai adattípust a tesztekben látható konstruktorokkal! Ezek után írd meg azt a függvényt, amely igazat ad vissza, ha őszi hónap a paramétere, egyébként pedig hamisat!

```
data Month = ... deriving (Eq, Show)
iLikeAutumn :: Month -> Bool
```

Tesztek:

```
iLikeAutumn Sep
iLikeAutumn Oct
iLikeAutumn Nov
all (not . iLikeAutumn) [Jan, Feb, Mar, Apr, May, Jun, Jul,
Aug, Dec]
```

4. feladat (⭐ ⭐): Definiáld azt a függvényt, amely egy valid, `Time` típusú értéket tud szövegből előállítani! Ha nem értelmezhető a szöveg formátuma, `"Not valid time format!"`, ha nem valid a két számból alkotott idő, akkor `"Not valid time!"`, egyébként a szövegben reprezentált, `Time` típusú érték legyen a végeredmény (lásd tesztek)! Tipp: `readMaybe` függvény használata (és a `::` típusspecializáló operátor) segíthet, nézz utána!

```
parseTime :: String -> Either Time String
```

Tesztek:

```
parseTime "12:14" == Left (T 12 14)
parseTime "26:32" == Right "Not valid time!"
parseTime "alma" == Right "Not valid time format!"
```

5. feladat (⭐⭐): Adott a következő dátum típus:

```
data Date = D Int Month Int Time deriving (Eq, Show)
```

Az első paraméter az évet, a második a hónapot, a harmadik a napot, és a negyedik egy időpontot reprezentál.
Példányosítsd az `Ord` típusosztályt a `Date` típusra!

```
instance Ord Date where
  (<=) :: Date -> Date -> Bool
```

Tesztek:

```
D 24 Dec 2012 (T 23 00) <= D 24 Dec 2013 (T 23 00)
D 24 Dec 2012 (T 23 00) <= D 24 Dec 2012 (T 23 00)
D 24 Dec 2012 (T 23 00) <= D 24 Dec 2012 (T 23 59)
D 24 Nov 2012 (T 23 00) <= D 24 Dec 2012 (T 23 00)
D 23 Dec 2012 (T 23 00) <= D 24 Dec 2012 (T 23 00)
not (D 24 Dec 2014 (T 23 00) <= D 24 Dec 2013 (T 23 00))
not (D 24 Nov 2012 (T 23 00) <= D 24 Sep 2012 (T 23 00))
not (D 23 Dec 2012 (T 23 00) <= D 22 Dec 2012 (T 23 00))
not (D 23 Dec 2012 (T 23 00) <= D 23 Dec 2012 (T 22 00))
```

6. feladat (⭐⭐): Készíts egy függvényt, amely egy megadott határidő és egy leadási időpontokat tartalmazó lista alapján meghatározza, hogy kik azok a hallgatók, akik a megoldásukat a határidő előtt adták le! A leadási időpontokhoz rendelt hallgatónevek határozzák meg, hogy ki mikor adta be a megoldását.

```
validAssignments :: Date -> [(Date, String)] -> [String]
```

Teszt:

```
validAssignments (D 14 Nov 2021 (T 23 59)) [(D 14 Oct 2021 (T
23 59), "Moricka"),(D 13 Nov 2021 (T 23 59), "Pistike"),(D 15
Nov 2021 (T 23 59), "Jozsika"),(D 14 Nov 2021 (T 23 34),
"Xavier")] == ["Moricka","Pistike","Xavier"]
```

11. gyakorlat - Parametrikus ADT-k

Ezen a gyakorlaton típusparaméterrel rendelkező algebrai adattípusokat tárgyalunk, különös figyelmet szánva a `Maybe` és `Either` szabványos típuskonstruktorokra. Ezenfelül a gyakorlaton szó esik a típusizomorfizmusról, és konkrét példákat is mutatunk izomorf típusokra.

Alapvető feladatok

1. feladat (⭐): Definiáld a `safeHead` függvényt, amely megadja egy lista első elemét, ha az létezik (különben `Nothing` legyen az eredmény)!

   ```
   safeHead :: [a] -> Maybe a
   ```

   Tesztek:

   ```
   safeHead [1..10] == Just 1
   safeHead [] == Nothing
   ```

2. feladat (⭐): Definiáld a `safeDiv` függvényt, amely eloszt két számot egymással, kivételek kiváltása nélkül! Ha a hányados 0, `Nothing` legyen az eredmény!

   ```
   safeDiv :: (Fractional a, Eq a) => a -> a -> Maybe a
   ```

   Tesztek:

   ```
   safeDiv 5 0 == Nothing
   safeDiv 4 4 == Just 1
   ```

3. feladat (⭐⭐): Adott az alábbi típusszinonima, amely asszociatív listákat reprezentál (`Int` kulcsokkal és `a` értékekkel), továbbá megadtuk az asszociatív listákba való beszúrás műveletet.

```
type Map a = [(Int, a)]

put :: Int -> a -> Map a -> Map a
put k v []     = [(k, v)]
put k v (x:xs) = if fst x /= k then x:put k v xs else (k,
v):xs
```

Definiáld a fenti típushoz a `get` műveletet, amely megadja, hogy az adott kulcshoz milyen érték tartozik!

```
get :: Int -> Map a -> Maybe a
```

Tesztek:

```
get 0 [] == Nothing
get 1 [(1,"apple")] == Just "apple"
get 10 [(1, "apple"),(2,"pear"),(0,"")] == Nothing
get 0 [(1, "apple"),(2,"pear"),(0,"")] == Just ""
```

4. feladat (⭐): Szorozz meg egy `Maybe Int` típusú értéket kettővel!

```
mult2Maybe :: Num a => Maybe a -> Maybe a
```

Tesztek:

```
mult2Maybe Nothing == Nothing
map (mult2Maybe . Just) [1..100] == map Just [2,4..200]
```

5. feladat (⭐): Adj össze két `Maybe Int` típusú értéket!

```
addMaybe :: Num a => Maybe a -> Maybe a -> Maybe a
```

Tesztek:

```
addMaybe Nothing Nothing == Nothing
addMaybe (Just 5) Nothing == Nothing
addMaybe Nothing (Just 5) == Nothing
addMaybe (Just 10) (Just 20) == Just 30
```

6. feladat (⭐): Definiáld a `Functor` példányt a `Maybe` típusra! Tipp: Ehhez a feladathoz érdemes a `Maybe` típust elrejteni: `import Prelude hiding (Maybe, Just, Nothing)`, és újradefiniálni `data Maybe a = Nothing | Just a`!
7. feladat (⭐): Adj meg egy olyan (nemüres) asszociatív listát (a fenti `Map` típussal), amely értékként karaktereket és logikai értékeket is tartalmaz! Milyen típussal lehet ezt kifejezni?
8. feladat (⭐⭐): Oldd meg az 1., 2. és 3. feladatokat `Either a String` típus használatával, ahol a `Right` konstruktor egy (feladatspecifikus) hibaüzenetet tartalmazzon!

Gyakorló feladatok

1. feladat (⭐): Definiáld a `safeLast` függvényt, amely megadja egy lista utolsó elemét, ha az létezik (különben `Nothing` legyen az eredmény)!

```
safeLast :: [a] -> Maybe a
```

Tesztek:

```
safeLast [1..10] == Just 10
safeLast [] == Nothing
```

2. feladat (⭐): Definiáld a `findIndex` függvényt, megadja a paraméterként kapott érték indexét a paraméter listában, amennyiben az érték a lista eleme (az első előfordulást vedd alapul)! Egyébként az eredmény legyen `Nothing`! Tipp: jól jöhet a `Maybe` típus `Functor` példánya (`fmap`).

```
findIndex :: Eq a => a -> [a] -> Maybe Int
```

Tesztek:

```
findIndex 3 [1..10] == Just 2
findIndex 110 [1..10] == Nothing
findIndex 6 [1,5,6,6,6,6] == Just 2
findIndex 'a' "almafa" == Just 0
```

3. feladat (⭐ ⭐): Definiáld a `lefts` (és `rights`) függvény, amely kiválogatja egy unió típusú értékeket tartalmazó listából a `Left` (és `Right`) konstruktorral létrehozottakat!

```
lefts :: [Either a b] -> [a]
rights :: [Either a b] -> [b]
```

Tesztek:

```
lefts [Left "cica", Right 4, Left "alma"] == ["cica", "alma"]
lefts ([Left (chr x) | x <- [97..110]] ++ [Right x | x <-
[97..110]]) == "abcdefghijklmn"
rights ([Left (chr x) | x <- [97..110]] ++ [Right x | x <-
[97..110]]) == [97..110]
rights [Left "cica", Right 4, Left "alma"] == [4]
```

4. feladat (⭐⭐): Definiáld a `partition` függvényt, amely szétválogat egy listát a `Left` és `Right` konstruktorok mentén!

```
partition :: [Either a b] -> ([a], [b])
```

Tesztek:

```
partition [Left "cica", Right 4, Left "alma"] == (["cica",
"alma"], [4])
partition ([Left (chr x) | x <- [97..110]] ++ [Right x | x <-
[97..110]]) == ("abcdefghijklmn", [97..110])
```

5. feladat (⭐⭐): Mutasd meg az alábbi izomorfizmusokat! (Azaz definiálj egy-egy függvényt, amely a két izomorf típus értékeit megfelelteti egymásnak úgy, hogy a kompozíciójuk identitás!)

- `Either () ()` és `Bool`
- `Maybe (Either a b)` és `Either a (Maybe b)`
- `(Bool, a)` és `(Either a a)`
- `(a,a)` és `(Bool -> a)`

6. feladat (⭐⭐): Definiáld az fenti asszociatív lista (`Map`) adattípus `put` (beszúrás) és `get` (lekérdezés) műveleteit úgy, hogy az asszociatív listát kulcs szerint növekvően rendezetten reprezentáljuk!

12. gyakorlat - Rekurzív ADT-k

Az utolsó gyakorlaton részletesen tárgyaljuk a rekurzív algebrai adattípusokat (pl. listák és bináris fák). A rekurzív adattípusok kapcsán két fontos típusosztályt is látnak a hallgatók: `Functor` és `Foldable`. Különös figyelmet szánunk a `Foldable` típusosztály példányosítására bináris fákhoz, ahol több különböző módon is definiálható a hajtogatás.

Alapvető feladatok

1. feladat (⭐): Definiálj egy típust, amely a nemüres listákat reprezentálja! A típus neve legyen `ConsList`!
2. feladat (⭐): Definiáld a `Foldable` és `Functor` típusosztály példányait a `ConsList` típushoz!
3. feladat (⭐): Definiálj egy függvényt, amely egy listát nemüres listára képez! Használj `Maybe` típust!

```
toConsList :: [a] -> Maybe (ConsList a)
```

Tesztek:

```
toConsList [] == Nothing
Data.Foldable.toList (case toConsList [1..10] of Just xs ->
xs) == [1..10]
```

4. feladat (⭐⭐): Definiálj egy típust, amely bináris fákat reprezentál! A típus neve legyen `Tree`!
5. feladat (⭐⭐): Add meg egy bináris fa magasságát!

```
height :: Tree a -> Int
```

6. feladat (⭐⭐): Add meg egy bináris fában szereplő számok összegét!

```
sumTree :: Num a => Tree a -> a
```

7. feladat (⭐⭐): Duplázz meg minden számot egy bináris fában!

```
doubleTree :: Num a => Tree a -> Tree a
```

8. feladat (⭐⭐⭐): Definiáld a `Foldable` és `Functor` típusosztály példányait a bináris fa típushoz!

Gyakorló feladatok

Definiálunk két különböző fa típust, az első csak a leveleiben, amíg a második csak a csúcsaiban tárol értékeket:

```
data LeafTree a = LLeaf a | LNode (LeafTree a) (LeafTree a)
deriving (Eq,Show)
data NodeTree a = NLeaf | NNode (NodeTree a) a (NodeTree a)
deriving (Eq,Show)
```

1. feladat (⭐ ⭐ ⭐): Definiáld ezekhez a fa típusokból a `Functor` és `Foldable` típusosztályok példányait!

   Tesztek :

```
fmap (+1) (LNode (LNode (LLeaf 1) (LLeaf 2)) (LNode (LLeaf 3)
(LNode (LLeaf 4) (LLeaf 5)))) == (LNode (LNode (LLeaf 2)
(LLeaf 3)) (LNode (LLeaf 4) (LNode (LLeaf 5) (LLeaf 6))))
fmap (*2) (LNode (LNode (LLeaf 1) (LLeaf 2)) (LNode (LLeaf 3)
(LNode (LLeaf 4) (LLeaf 5)))) == (LNode (LNode (LLeaf 2)
(LLeaf 4)) (LNode (LLeaf 6) (LNode (LLeaf 8) (LLeaf 10))))
sum (LNode (LNode (LLeaf 1) (LLeaf 2)) (LNode (LLeaf 3)
(LNode (LLeaf 4) (LLeaf 5)))) == 15
product (LNode (LNode (LLeaf 1) (LLeaf 2)) (LNode (LLeaf 3)
(LNode (LLeaf 4) (LLeaf 5)))) == 120
fmap (+1) (NNode (NNode (NNode NLeaf 1 NLeaf) 2 NLeaf) 3
(NNode NLeaf 4 (NNode NLeaf 5 NLeaf))) == (NNode (NNode
(NNode NLeaf 2 NLeaf) 3 NLeaf) 4 (NNode NLeaf 5 (NNode NLeaf
6 NLeaf)))
fmap (*2) (NNode (NNode (NNode NLeaf 1 NLeaf) 2 NLeaf) 3
(NNode NLeaf 4 (NNode NLeaf 5 NLeaf))) == (NNode (NNode
(NNode NLeaf 2 NLeaf) 4 NLeaf) 6 (NNode NLeaf 8 (NNode NLeaf
10 NLeaf)))
sum (NNode (NNode (NNode NLeaf 1 NLeaf) 2 NLeaf) 3 (NNode
NLeaf 4 (NNode NLeaf 5 NLeaf))) == 15
product (NNode (NNode (NNode NLeaf 1 NLeaf) 2 NLeaf) 3 (NNode
NLeaf 4 (NNode NLeaf 5 NLeaf))) == 120
```

2. feladat (⭐⭐⭐): Adott egy inorder és egy preorder bejárása egy bináris fának (egy-egy listában leírva). Építsd fel a fát a bejárások alapján! Feltételezhető, hogy a két paraméterlista egyforma hosszú.

```
buildTree :: [a] -> [a] -> NodeTree a
```

Tesztek:

```
buildTree [0, 1, 3, 4, 2, 5] [3, 1, 4, 0, 5, 2] == NNode
(NNode (NNode NLeaf 3 NLeaf) 1 (NNode NLeaf 4 NLeaf)) 0
(NNode (NNode NLeaf 5 NLeaf) 2 NLeaf)
```

3. feladat (⭐⭐⭐): A következő típus olyan fákat ábrázol, ahol minden csúcsnak valahány gyereke lehet:

```
data RoseTree a = RNode a [RoseTree a] deriving (Eq, Show)
```

Add meg egy ilyen fa magasságát!

```
rHeight :: RoseTree a -> Int
```

Tesztek:

```
rHeight (RNode 1 []) == 1
rHeight (RNode 1 [RNode 2 [],RNode 3 [],RNode 4 []]) == 2
rHeight (RNode 1 [RNode 2 [RNode 2 [RNode 2
[]]]],RNode 3 [],RNode 4 []]) == 5
```

4. feladat (🔥): Add meg egy `RoseTree` típus `Functor` és `Foldable` példányait! Próbáld meg a `Foldable` példányt többféle módon is megadni (postorder és preorder bejárással)!

Tesztek:

```
fmap (+1) (RNode 1 []) == RNode 2 []
fmap (+1) (RNode 1 [RNode 2 [RNode 4 []],RNode 3 []]) ==
RNode 2 [RNode 3 [RNode 5 []],RNode 4 []]
sum (RNode 1 [RNode 2 [RNode 4 []],RNode 3 []])
 -- postorder, balról-jobbra bejárás esetén:
Data.Foldable.toList (RNode 1 [RNode 2 [RNode 4 []],RNode 3
[]]) == [4,2,3,1]
 -- preorder, balról-jobbra bejárás esetén:
Data.Foldable.toList (RNode 1 [RNode 2 [RNode 4 []],RNode 3
[]]) == [1,2,4,3]
 -- preorder, balról-jobbra bejárás esetén:
foldr (++) [] (RNode "a" [RNode "p" [RNode "p" [], RNode "l"
[]], RNode "e" []]) == "apple"
```