Funkcionális Programozás elméleti összefoglaló v0.1 Draft verzió

Merth Borbála

August 2024

1 Bevezetés

Ez a dokumentum az ELTE Informatikai Karán oktatott Funkcionális programozás tárgy anyagát foglalja össze. A tárgy a Haskell programozási nyelven keresztül mutatja be a funkcionális programozás alapjait, így ez az anyag tekinthető egy elméleti Haskell alapozásnak is. A dokumentum tartalmaz egymásra épülő definíciókat, így érdemes azt a megadott sorrendben olvasni. A leírásban említett függvényekről részletesebb információt a dokumentációban, a Hoogle weboldalon keresztül lehet kapni.

2 Mit jelent a funkcionális programozás?

A funkcionális programozás egy programozási módszertan, amely általában a deklaratív stílust követi. Az imperatív programozással szemben deklaratív módszer esetén nem az eredményhez vezető lépéseket, hanem az eredmény képletét határozzuk meg. A funkcionális programnyelvek a programozási feladatot egy függvény kiértékelésének tekintik. A két fő eleme az érték és a függvény, nevét is a függvények kitüntetett szerepének köszönheti.

Kicsit másképpen megfogalmazva mondhatjuk úgy is, hogy funkcionális programozás során azt írjuk le, **mit** kell megoldani, **nem** pedig azt, hogy mindez **hogyan** történjen.

A funkcionális programnyelvek alapja általában a λ-kalkulus (Lambda-kalkulus), ezáltal a funkcionálisan definiált függvények nagyban hasonlítanak a matematikából ismert függvényekhez.

3 A Haskell programozási nyelvről röviden

A Haskell programozási nyelv...

- ... egy funkcionális paradigmát követő programozási nyelv, tehát az építőelemei függvények és értékek. Szigorúan véve **minden**, amit a nyelvben definiálunk, valamilyen **kiértékelhető függvény**.
- ... **tisztán funkcionális**, azaz mentes mindenféle mellékhatásoktól, a függvények kiértékelései mindig ugyanazt az eredményt adják, nem állít elő egyéb adatot.
- ... **statikusan és erősen típusozott**, azaz minden kifejezés és részkifejezés típusa már fordítási időben ismert. A típusokat be kell tartani, csak explicit konverzióval lehet egy típust másik típussá alakítani.
- ... rendelkezik a **hivatkozási helyfüggetlenség** tulajdonságával, azaz a kifejezések értéke nem függ attól, hogy a program szövegében hol helyezkednek el, bárhol helyezzük el őket, ugyanazt jelentik.
- ... nem engedi a függvény túlterhelést (angolul: *function overload*), azaz nem lehet felüldefiniálni már létező függvényeket, kifejezéseket.

4 Modulok

Haskellben a modul jelenti a fordítási egységet, a modulnevek mindig nagybetűvel kezdődnek. Egy modul több különböző függvényt, egyedi típusokat tartalmazhat, amelyek általában valamilyen módon kapcsolatban állnak egymással. Modulokat lehet importálni más modulokba, a benne található exportált függvényeket és típusokat fel lehet használni a megoldásokban.

A Haskell rendelkezik egy `Prelude` nevű modullal, amely az alapvető típusokat és függvényeket tartalmazza, és amely minden egyes programba alapértelmezetten importálva van.

További hasznos modulok:

- `Data.Char`: a karakter típushoz (5.4. fejezet) tartalmaz több hasznos függvényt, pl: kisbetűvé alakítás, nagybetűvé alakítás, szám karakter-e, szóköz-e ...
- `Data.List`: a listákhoz (12. fejezet) tartalmaz hasznos műveleteket, pl: permutációk előállítása ...

5 Alapvető típusok

A tárgy elvégzése során megismerkedtek néhány alapvető típussal és azok tulajdonságaival. A következő fejezetben ezek kerülnek kifejtésre.

5.1 `Int` és `Integer`

Az `Int` és `Integer` típusok az **egész számok**nak egy-egy reprezentációja. A kettő közti különbség a felvehető minimális és maximális értékekben rejlik. Az `Int` típus a hagyományos, az architektúrától függően 32 vagy 64 bites egész szám reprezentáció, így ennek van alsó és felső korlátja. Az `Integer` egy tetszőleges pontosságú típus (angolul: *arbitrary precision type*), amelynek meghatározott korlátja nincsen. Egyedül a számítógép memóriájától függ, hogy mekkora maximális és minimális értéket tud felvenni. Általában az `Integer` típust ajánlott használni.

Az `Int` és az `Integer` típusok konstruktorai maguk a szám literálok, tehát például `5` és `16`.

Az `Int` és az `Integer` típusokra értelmezett, gyakran használt `Prelude` műveletek:

- `(+)` : számok összege,
- `(-)` : számok különbsége,
- `(*)` : számok szorzata,
- `div` : egész osztás hányadosa, (egész rész)
- `mod` : egész osztás maradék része, ha a második paraméter negatív, akkor az eredményt is negatív irányban adja meg,
- `(^)` : hatványozás, az első számot (alap) a második számmal (hatványkitevő) egyező hatványra emeli, csak nemnegatív egész szám lehet a hátványkitevő.
- `(<)` és `(<=)` : eldönti, hogy az első érték kisebb-e (vagy egyenlő) a másodiknál (`Bool` értéket ad vissza)
- `(>)` és `(>=)` : eldönti, hogy az első érték nagyobb-e (vagy egyenlő) a másodiknál (`Bool` értéket ad vissza)
- `(==)` : eldönti, hogy két egész szám egyenlő-e (`Bool` értéket ad vissza)
- `(/=)` : eldönti, hogy két egész szám különbözik-e (`Bool` értéket ad vissza)
- `fromIntegral` : egész szám típusból (`Int` és `Integer`) bármilyen szám típusba képez (`Int`, `Integer`, `Float`, `Double`, stb.)

5.2 `Double` és `Float`

A `Double` és `Float` típusok a **törtszámok**nak egy-egy reprezentációja. Széles skálán tudja reprezentálni a nem egész értékeket, bizonyos fokú pontosság mellett. A `Double` egy dupla pontosságú lebegőpontos számtípus, míg a `Float` egy szimpla pontosságú lebegőpontos számtípus. Általában a `Double` típust használjuk.

A `Double` és a `Float` típusok konstruktorai is a számliterálok, illetve az egész részt a tört résztől elválasztó `.` karakterből állnak, tehát például `5.6` és `13.125`. A *"nagy"* vagy *"nagyon kicsi"* értékekkel normalizált formában találkozunk, pl.

- $1.815454233e8$, ez a $1.815454233 * 10^8$ érték, azaz 181545423.3,
- $9.0e-7$, ez a $9.0 * 10^{-7}$ érték, azaz 0.0000009.

A `Double` és a `Float` típusra értelmezett, gyakran használt `Prelude` műveletek:

- `(+)` : számok összege,
- `(-)` : számok különbsége,
- `(*)` : szokok szorzata,
- `(/)` : számok osztása,
- `(<)` és `(<=)` : eldönti, hogy az első érték kisebb-e (vagy egyenlő) a másodiknál (`Bool` értéket ad vissza),
- `(>)` és `(>=)` : eldönti, hogy az első érték nagyobb-e (vagy egyenlő) a másodiknál (`Bool` értéket ad vissza),
- `(==)` : eldönti, hogy a két szám egyenlő-e (`Bool` értéket ad vissza), használata nem javasolt lebegőpontos számokra,
- `(/=)` : eldönti, hogy a két szám különbözik-e (`Bool` értéket ad vissza),
- `round` : kerekítés, az általános kerekítési szabály szerint kerekíti egész számra a törtszámot, egyedül a `.5` végződésű értékeket kerekíti mindig a páros egész szám irányába,
- `truncate` : egészrészre történő kerekítés, a tizedespontot követő részt figyelmen kívül hagyja,
- `floor` : alsó egészrészre történő kerekítés, a kisebb egész érték felé kerekíti a számot,
- `ceiling` : felső egészrészre történő kerekítés, a nagyobb egész érték felé kerekíti a számot,

5.3 `Bool`

A `Bool` típus a logikai értékek halmaza. Két konstruktora a `True` (igaz) és a `False` (hamis).

A `Bool` típusra értelmezett, gyakran használt `Prelude` műveletek:

- `(==)` : eldönti, hogy két logikai érték megegyezik-e,
- `(/=)` : eldönti, hogy két logikai érték különbözik-e,
- `(&&)` : logikai konjunkció (és) művelete,
- `(||)` : logikai diszjunkció (vagy) művelete,
- `not` : logikai tagadás művelete.

5.4 `Char`

A `Char` típus a karaktereket ábrázolja, konstruktorai a karakter literálok két aposztróf (`'`) közt, tehát például `'a'` és `'Z'`. A kis- és nagybetűs karakterek különbözőnek számítanak.

A `Char` típusra értelmezett, gyakran használt `Prelude` műveletek:

- `(==)` : eldönti, hogy két karakter megegyezik-e (`Bool` értéket ad vissza),
- `(/=)` : eldönti, hogy két karakter különbözik-e (`Bool` értéket ad vissza),
- `(<)` és `(<=)` : eldönti, hogy az első karakter kódja kisebb-e a másodiknál vagy egyenlőek (`Bool` értéket ad vissza),
- `(>)` és `(>=)` : eldönti, hogy az első karakter kódja nagyobb-e a másodiknál vagy egyenlőek (`Bool` értéket ad vissza),

6 Típusosztályok

A típusosztályok műveletek gyűjteménye, amely valamilyen tulajdonság köré épülnek. Ilyen például a `Num`, ami a számokra értelmezett alapvető aritmatikai műveleteket, az `Eq`, ami az egyenlőségvizsgálatot adja meg, stb.

A típusztályok segítségével *"túlterhelhetünk"* műveleteket, azaz, ugyanazt a műveletet több különböző típusra értlemezni tudjuk. Ilyen például az `Num` típusosztályhoz tartozó összeadás `(+)`, ami használható egészekre vagy lebegőpontosakra is. Ugyanazt a szimbólumot használjuk, de más a mögötte lévő megvalósítás, hiszen ezen típusok reprezentációja különbözik.

Hogy melyik típusosztálynak mik a követelményei, azt a fordítótól meg lehet kérdezni: `:i {típusosztály neve}`. Ha ezt beírjuk a konzolba, akkor láthatjuk többek közt a típusosztály definícióját a hozzá tartozó függvényekkel, illetve alatta a MINIMAL jelzés után a példányosításhoz kötelezően definiálandó függvényeket. A típusosztály definíciójában szerepelhetnek megkötések. Ezek azt jelzik, hogy az adott típusosztály mely típusosztályktól függ. Például az `Integral` típusosztály a `Real` és az `Enum` típusosztályoknak is részét képezi.

```
ghci> :i Integral
type Integral :: * -> Constraint
class (Real a, Enum a) => Integral a where
  quot :: a -> a -> a
  rem :: a -> a -> a
  div :: a -> a -> a
  mod :: a -> a -> a
  quotRem :: a -> a -> (a, a)
  divMod :: a -> a -> (a, a)
  toInteger :: a -> Integer
  {-# MINIMAL quotRem, toInteger #-}
        -- Defined in `GHC.Real'
instance Integral Int -- Defined in `GHC.Real'
instance Integral Integer -- Defined in `GHC.Real'
instance Integral Word -- Defined in `GHC.Real'
```

Az `Integral` típusosztály adatai

Fontosabb típuosztályok:

- `Num`: Alapvető aritmatikai műveletek (összeadés, különbségképzés, szorzás, negálás, stb.), amelyek minden számtípusra értelemzhetőek.
- `Fractional`: Tört számok, amely a nem egész osztás műveletével `(/)` bővíti a `Num` típusosztály műveleteit.
- `Integral`: Az egész számok osztását vezeti be `div` és `mod`.

- `Eq`: Az egyenlőség vizsgálatának műveletét adja meg `(==)`.
- `Ord`: Az elemek összehasonlíthatóságát adja, `(<)`, `(<=)`, `(>)`, stb.
- `Enum`:
- `Show`:

Természetesen léteznek más típusozstályok is, ezek megismerését az olvasóra bízzuk. Az `Eq` típusosztály például az egyenlőség vizsgálatának lehetőségét garantálja, míg az `Ord` típusosztály sorrendiséget határoz meg egy típus értékei közt. A `Num` típusosztály az alapvető matematikai műveleteket garantálja. A `Show` típusosztály felel azért, hogy egy típus értékei kiírhatóak legyenek a konzolra.

7 Rendezett n-esek (`Tuple`)

Haskell programozási nyelvben lehetőség van arra, hogy több értéket "egyszerre" adjunk át. Különböző értékeket egy rendezett n-esbe csomagolhatunk, ahol 'n' a tárolt értékek száma. Egy rendezett n-esben szerepelhet több **különböző típusú** érték is, tehát ez egy **heterogén** adatszerkezet. (Adatszerkezetekről bővebben a 10.3. fejezetben.)

A rendezett párok konstruktora a `(,)` függvény. Általánosan a rendezett n-esek konstruktora hasonlóképpen (n-1) vessző zárójelben. Mikor egy rendezett n-est szeretnénk megadni, csak fel kell soroljuk a kívánt értékeket zárójelek közt és vesszővel elválasztva.

A típus nevéből is adódóan a benne tárolt típusok és értékek **sorrendje lényeges**. Ha egy `(Int, Double)` típusú értéket szeretnék megadni, akkor a rendezett n-es első helyére csak `Int` típusú értéket, míg a második helyére csak `Double` típusú értéket írhatunk. Ebből kifolyólag a `(Int, Double)` típus nem egyezik meg a `(Double, Int)` típussal.

Szintén a sorrendiségből kifolyólag az `(1, 2)` sem egyezik meg a `(2, 1)` értékkel még akkor sem, ha a típusuk megegyezik.

A rendezett pároknak létezik két **destruktora** is, amelyek segítségével kinyerhetjük a rendezett pár egyes tagjait. Az első tagot az `fst` függvény segítségével, a második tagot az `snd` függvény segítségével tudjuk elérni. Tehát például az `fst (1, 2)` eredménye 1, míg az `snd (1, 2)` eredménye 2 lesz.

8 Egyszerű függvények, kifejezések

A Haskell programnyelv legfontosabb elemei a függvények, hiszen alapvetően minden függvény ebben a nyelvben. Éppen ezért nagyon fontos tisztában lenni a használatukkal, felépítésükkel, definiálásukkal.

8.1 Szintakszis és alaptulajdonságok

Minden függvénynek van típusszignatúrája és függvénytörzse. A típusszignatúra tartalmazza az esetleges paraméterek sorrendjét és típusát, valamint a függvény visszatérési értékének típusát is. Ezt nem kötelező megadni, mivel a legtöbb esetben ki tudja következtetni a fordító, azonban segítség a definíció típushelyességének ellenőrzésében, így erősen ajánlott megadni. Ennek szintakszisa az alábbi (a sötétszürkével jelölt rész opcionális):

```
{név} :: {1.par. típusa} -> {2.par. típusa} -> ... -> {eredmény típusa}

Például:
f :: Int -> Integer -> Double
g :: Bool -> Bool
h :: Char
```

A függvények neve mindig kisbetűvel kell kezdődjön, ezt követheti újabb kisbetű, nagybetű, szám és aposztróf.

A típusszignatúrával ellentétben a függvénytörzs a függvények elengedhetetlen része, ugyanis ez tartalmazza a visszatérési érték képletét, a függvény definícióját. Amennyiben paraméteres függvényt definiálunk, úgy a paramétereknek is itt kell nevet adnunk. A paraméterek neve kisbetűvel kell kezdődjön. Amilyen nevet adunk egy paraméternek az egyenlőség bal oldalán, olyan néven tudunk hivatkozni rá a jobb oldalon. Ennek szintakszisa az alábbi (a szürkével jelölt rész opcionális):

```
{név} {1.param.} {2.param.} ... = {eredmény képlete/definíciója}

Például:
f n m = 1.0
g b = not b
h = 'h'
```

Mint azt a példák is mutatják, függvényt lehet paraméterek nélkül is definiálni (*lásd 3. példa*), ám ezesetben az adott függvény biztosan konstans lesz. A definícióban nem kötelező felhasználni a paramétereket, akár egy paraméteres függvény is lehet konstans, vagy lehet fel nem használt paramétere (*lásd 1. példa*).

A hivatkozási helyfüggetlenségnek köszönhetően nem kell, hogy a típusszignatúrát és a függvénytörzset közvetlen egymás után adjuk meg, bármilyen sorrendben szerepelhetnek a program szövegében, és akár egyéb definíciók is kerülhetnek a típusszignatúra és a teljes függvénytörzs közé.

FONTOS azonban, hogyha a függvénytörzs több sorból áll, annak sorai közé nem írhatunk más definíciókat. Illetve típustól függetlenül nem létezhet két ugyanolyan nevű függvény, mivel nem megengedett a függvény túlterhelés (*function overload*).

8.2 Függvényhívás, függvényalkalmazás

Haskell nyelvben a függvényhívás eltér az imperatív nyelvekben megszokott módtól. A paramétereket nem a függvény neve után zárójelben kell felsorolni, hanem egyszerűen a függvény neve után szóközökkel elválasztva kell megadni őket.

```
Adott az alábbi függvény:
f :: Int -> Int -> Int
f x y = x + y
A függvény egy helyes meghívása:
f 10 20
```

Amennyiben egy másik függvényhívás eredményét szeretnénk átadni egy függvénynek, úgy azt zárójelek közt kell átadnunk, mert egyéb esetben nem egyetlen paraméterként lesz értelmezve.

```
Példa függvényhívásra egy másik függvényhívás eredményével:
f (10 + 20) 9
f 9 (10 + 20)
```

Mivel a `(->)` operátor egy jobbra kötő művelet (bővebben 8.3. fejezetben), ezért megtehetjük azt, hogy egy függvénynek nem adjuk át az összes paraméterét. Ilyenkor egy újabb függvényt kapunk, amely a hátramaradt paramétereket várja még, hogy konkrét eredményt adjon. Mikor egy függvénynek nem adjuk át az összes paraméterét (de legalább egyet igen), akkor **parciális függvényalkalmazást** végzünk.

8.3 Fixitás

A bináris - azaz két paraméteres - függvények esetében különösen fontos téma a fixitásuk, ugyanis ez határozza meg, hogy milyen irányba kötnek. A legtöbb függvényt alapvetően csak **prefixen** lehet alkalmazni, azaz először meg kell adni a függvény nevét, majd utána a paramétereit egymás után. Néhány függvény esetében, mint például a bináris függvények, azonban van lehetőség **infix** alkalmazásra, amikor a függvény neve két paraméter között helyezkedik el. Az infix használat esetén fontos megkötni, hogy van-e kötési iránya, ha van, akkor balra vagy jobbra köt a művelet, illetve mekkora a kötési erőssége (a többi függvényhez képest).

FONTOS, hogy mindig a **prefixen** alkalmazott függvénynek lesz **nagyobb** a kötési erőssége.

Egy függvény balra köt, ha többszörös infix alkalmazás során balról jobbra értékeli ki a kifejezést. Egy függvény jobbra köt, amennyiben jobbról balra értékeli ki a kifejezést. Tulajdonképpen a fixitás a zárójelezésről és a függvény asszociativitásáról

szól. Ha nem határozunk meg kötési irányt, úgy zárójelek nélkül nem fogja tudni értelmezni a fordító a többszörös függvényalkalmazást.

```
Adott az alábbi kifejezés:
1 + 2 + 3
Jelentése, ha a (+) balra köt:
(1 + 2) + 3
Jelentése, ha a (+) jobbra köt:
1 + (2 + 3)
```

A kötési erősség 0 és 9 közti érték kell legyen, ahol a 0 a leggyengébb, a 9 a legerősebb kötést jelöli. Egy függvény fixitásának megadásához háromféle kulcsszó áll rendelkezésre: `infixl` a balra kötést, `infixr` a jobbra kötést, `infix` a kötési irány nélküli infixitást jelöli. Mindhárom kulcsszó a kötési erősséget és a függvény infix nevét várja paraméterül.

Azokat a függvényeket, amelyeket nem latin betűkkel nevezünk el, hanem valamilyen "speciális" karakterrel (pl: `(+)`, `(-)`), azokat **operátor**oknak szokás hívni. A bináris operátorok alapvetően infixen alkalmazandóak. Operátor prefix alkalmazásához a zárójeles nevét kell használni. Ha latin betűkkel elnevezett bináris függvényt szeretnénk infixen alkalmazni, akkor a nevét backtick (‘) karakterek közé kell írni. Példa fixitás megadására:

```
infixl 6 +
infixr 7 ‘f‘
infix 2 ‘f‘
f :: Int -> Int -> Int
f x y = x * y
```

8.4 Lusta kiértékelés

Haskell nyelvben a kifejezések kiértékelése lusta, minden kifejezés és részkifejezés csak akkor értékelődik ki, ha szükség van az eredményére. Ez a későbbiekben például a listák (12. fejezet) esetében fog nagy előnyt jelenteni.

```
Adott az alábbi függvény:
f :: Int -> Int
f x = 1
Az alábbi függvényhíváskor a ‘(10 + 20)‘ paraméterül átadott
kifejezés nem szükséges az eredményhez, így nem lesz kiértékelve:
f (10 + 20)
```

8.5 Curry-zés és η-redukció (éta-redukció)

8.5.1 Curry és uncurry

Mivel már megismerkedtünk a függvény típussal, és a rendezett n-esekkel, érdemes megismerkednünk kettő paradigmával, amit a több paraméteres függvények esetén alkalmazunk. Ugyanis n paramétert ádahatunk rendezett n-es formájában (`(a , b, c) -> d`), vagy átadhatjuk "egyesével" a paramétereket, és egy másik függvénnyel térünk vissza (`a -> b -> c -> d`).

Az első módszert inkább az imperatív nyelvek, míg a másodikat inkább a deklaratív nyelvek szokták alkalmazni. Jó hír azonban, hogy a kettő kifejezés ekvivalens, az átalakításra adott két függvény, a `curry` és az `uncurry`, a következő típusokkal:

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

Ezek alapján a rendezett n-es megoldást "uncurry-zett formának", a másikat "curry-zett formának" szokás nevezni.

A különbség a két megoldás között, hogy a paramétereket egyszerre adjuk át, vagy megengedjük a felhasználónak, hogy a parmétereket egymás után tudja meg adni, ezzel egyben esetleg parciálisan is tudják használni a függvényt.

8.5.2 η-redukció

Az éta-redukciót megérteni legegyszerűbben egy gyakorlati példán keresztül lehet. Írjunk meg egy függvényt, amely egy számot kap, és eredményül ennek háromszorosával tér vissza:

```
triple :: Int -> Int
triple = _
```

Amint megpróbáljuk ezt lefordítani, hibát kapunk, mely szerint a `_` helyére egy `Int -> Int` típusú dolgot kell írnunk. Jegyezzük ezt meg, majd vegyük fel a paramétert!

```
triple :: Int -> Int
triple x = _
```

Ekkor a hibaüzenet szerint a `_` helyére már `Int` típusú dolgot kell írnunk.

```
triple :: Int -> Int
triple x = (*) 3 x -- (*) 3 x == 3 * x
```

Alkalmazzuk rá az η-redukciót! Látható, hogy az utoljára felvett paraméterünk, vagyis az `x`, az egenlőség jobboldalán, a kifejezés "legvégén" helyezkedik el. Emiatt az egyenlőség mind a két oldalán "eltüntethetjük" a paramétert. Így a függvény definíciója az alábbira módusul:

```
triple :: Int -> Int
triple = (*) 3
```

Ilyenkor valójában a függvényünket egy másik, megfelelő típusú függvénnyel definiáljuk. Ez természetesen több paraméterre is alkalmazható.

A matematika nyelvén megfogalmazva: legyen f és g kettő függvény, amelyeknek egyetlen paramétere van. Abban az esetben, ha $\forall x : f(x) = g(x)$ akkor $f = g$. Hiszen minden bemeneti paraméterre pontosan ugyanazt a kimenetet adják, így a két függvény egyenlő.

Megjegyzés: A `(*) 3` típusa `Int -> Int` lesz, hiszen az egy parciálisan alkalmazott függvény. Feljebb láthattuk is, hogy amikor nem vesszük fel a paramétert, akkor egy függvényt kell megadnunk az egyenlőség jobboldalán, amire a szorzás kifejezése pont illeszkedik.

8.6 Esetszétválasztás (parciális és totális függvény)

Függvények definiálásakor van olyan, hogy bizonyos feltételek mentén el szeretnénk ágazni. Erre Haskell nyelvben is van lehetőség az őrfeltételek (angolul: *guard*) segítségével. Lehet egyszerre több őrfeltételt is megadni, azonban fontos, hogy ezek a feltételek fentről lefelé kerülnek ellenőrzésre, és az első igaz feltételnél leáll az ellenőrzés, és kiértékelődik a függvény az adott ágon.

Az őrfeltételek megadásának szintakszisa:

```
{függvénynév} {1.param} ... {utolsó param}
    | {feltétel} = {eredmény, ha teljesül}
    | {feltétel} = {eredmény, ha teljesül}
    ...
```

FONTOS, hogy a paraméterek után **nincs** egyenlőség, csak minden egyes feltétel után. Illetve oda kell figyelni arra, hogy lehetőleg minden esetet lefedjünk a feltételekkel, hogy a függvény **totális** legyen, azaz minden lehetséges bemenetre legyen kimenete. Amennyiben van olyan bemeneti érték, amelyet nem fed le egyetleg ág sem, úgy a függvény **parciális**.

Általánosan bevett szokás a konstans igaz feltétellel zárni a feltételek sorát, hogy garantáltan minden esetet lekezeljünk. Ehhez a feltétel helyére írhatjuk a `Bool` típus `True` konstruktorát, de van rá egy konstans függvény is, az `otherwise`, ami szintén

a `True` értéket adja vissza. Természetesen ezt nem szükséges használni, amennyiben szándékosan parciális a függvényünk, vagy enélkül is biztosra tudjuk, hogy teljes a lefedettség.

Pl: ha `x` egy szám, akkor `x < 0` és `x >= 0` feltételek után felesleges egy `otherwise` ág is.

8.7 Mintaillesztés

A függvények definiálásakor sokszor az egyes paraméterek típusa mentén is szeretnénk küklönböző eseteket meghatározni. Erre egy egyszerű példa a `not` függvény, amely igazra hamisat, hamisra pedig igazat kell visszaadjon. Akkor lenne a legkönnyebb definiálni, ha minden ágon tudhatnánk a paraméter igazságértékét.

Mintaillesztésnél háromféle mintát használhatunk. Az első, amelyet már eddig is használtunk, hogy adunk az értéknek egy nevet. Ilyenkor a típusán belül bármilyen értéket felvehet. Amennyiben nem akarjuk felhasználni az adott paramétert egy ágon, úgy használhatjuk rá a joker mintát, azaz az `'_'` (alávonás) karaktert. Ekkor szintén bármilyen értéket felvehet a típusán belül, azonban nem tudunk hivatkozni rá a definíció jobboldalán.

A harmadik lehetőség, hogy a paraméter típusának konstruktorára illesztjük a mintánkat. Ez lesz segítségünkre például a `not` definiálásakor is. Ha a `Bool` típus konstruktoraira illesztünk mintát, akkor a `not` függvény az alábbi módon néz ki:

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

Ahogy a példa is mutatja, mintaillesztés esetén legtöbbször több sorban, több ágon definiáljuk a függvényünket. Ahogy az esetszétválasztás esetén, úgy a függvénytörzs esetén is fentről lefele halad a kiértékelés, és az első illeszkedő mintánál megáll. Éppen ezért nagyon fontos odafigyelni arra, hogy a **legszűkebb mintát helyezzük legfelülre** és a legtágabbat legalulra.

8.8 Lokális hatókör, segédfüggvények

Mikor nehezebb feladatokat oldunk meg, néha nagyon hosszú definíciókat kell megadnunk sok-sok részkifejezéssel, vagy segédfüggvényekre is szükségünk lehet. Haskell nyelvben minden függvénynek, mi több a függvénytörzsének minden sorának lehet saját lokális hatóköre a `where` kulcsszó segítségével. Ebben a lokális hatókörben definiálhatunk konstans függvényeket a definíció részkifejezéseinek, illetve segédfüggvényeket is egyes részfeladatok ellátására. Azokat a függvényeket, amelyeket a lokális

hatókörben definiálunk, csak a hozzátartozó függvénytörzs sorban tudjuk kívülről is elérni.

Egy egyszerű példa a lokális hatókör bevezetésére:

```
example :: Int -> Int -> Int
example n m = z where
    z = helper n m
    helper :: Int -> Int -> Int
    helper x y = x * y
```

FONTOS, hogy minél méllyebben kívánunk újabb lokális hatókört bevezetni, annál nagyobb kell legyen a behúzás a sorok elején. Azok a függvények, amelyek nincsenek beljebb húzva, azok globálisak.

8.9 Rekurzió

Az imperatív nyelvekből ismert ciklusok, mint a `for` vagy a `while` helyett Haskell nyelvben a rekurziót szokás használni. A rekurzió a leggyakrabban használt nyelvi elem bonyolultabb függvények definiálásakor. **Egy függvény rekurzív, ha definíciójában meghívja önmagát.**

Létezik **véges** és **végtelen** rekurzió. Amennyiben a definíciónak van olyan ága, amely nem hívja vissza önmagát, és ez az ág előbb-utóbb elérhető lesz, úgy véges a rekurzió, egyéb esetben végtelenségig hívja újra önmagát a függvény, ezáltal végtelen a rekurzió.

Ezen felül létezik még a **végrekurzió**, amikor a függvényünk egy rekurzív függvénynek az eredményével tér vissza. Ez a későbbiekben a listák (12. fejezet) esetén még jelentős lesz. Egy példa a végrekurzióra:

```
fact :: Int -> Int
fact n = helper n 1 where
    helper :: Int -> Int -> Int
    helper 1 acc = acc
    helper n acc = helper (n - 1) (n * acc)
```

Látható, hogy a segédfüggvényünkben bevezettünk egy akkumuláló elemet, amely elvégzi menet közben a szorzásokat, ez egy egész hatékony megoldás.

8.10 Nyelvi elemek: `let-in` és `case-of`

8.10.1 `let-in` kifejezés

Már megismerkedhettünk a `where` kulcsszóval, amely létrehoz egy lokális hatókört egy függvényhez. Ennek párja a `let ... in ...` kifejezés, amely a függvényen belül hoz létre egy lokális hatókört. Ezt bárhol használhatjuk a függvény definícióján belül, akár egymásba is ágyazhatjuk őket. Példák a használatára:

```
apple :: Int
apple = let
    pear :: Int
    pear = 32
  in pear + 10

nested :: Int -> Int
nested x = let
    inc = (+) 1
  in let
    doubleinc input = inc (input * 2)
  in doubleinc (inc x)
```

A fordító a `let-in` kifejezéseket sokkal jobban tudja kezelni, mint a `where` blokkot, ugyanis `let-in` esetén előre deklaráljuk a szükséges függvényeket, a lokális hatókört. Ha szeretnénk nagyon effektív Haskell programot írni, akkor ajánlott a `where` blokk helyett `let-in` kifejezést használni.

8.10.2 `case-of` kifejezés

A feladatok során többször előfordulhat, hogy szeretnénk egy általunk meghívott függvény eredményére mintát illeszteni. Ebben az esetben definiálhatunk lokális függvényt, vagy használhatjuk a `case-of` kifejezést is, amivel a függvény definícióján belül tudunk egy értékre mintát illeszteni. `Case-of` kifejezések esetén is fontos odafigyelni a mintaillesztések megfelelő sorrendjére.

A `case-of` kifejezés szintakszisa az alábbi:

```
case {kifejezés} of
    {1. minta} -> {hozzá tartozó érték}
    {2. minta} -> {hozzá tartozó érték}
    ...
    {utolsó minta} -> {hozzá tartozó érték}
```

Vegyünk elő egy egyszerű példát, amelyben eldöntjük egy franciakártyáról, hogy fekete-e az alapszíne. Ehhez egy rendezett párban átvesszük a lap típusát jelző karaktert ('s' - spades/pikk, 'c' - clubs/treff, 'd' - diamonds/káró, 'h' - hearts/kőr), illetve a lap számát. Ha a lap típusa 's' vagy 'c', akkor igazat kell adjunk, egyébként hamisat. Persze ezt a feladatot meg lehetne oldani egyszerűbben is mintaillesztéssel, de ettől most tekintsünk el. Nézzük meg, hogyan oldanánk meg ezt a feladatot `where` kulcsszó segítségével, illetve hogyan oldanánk meg `case-of` segítségével.

```
where:
cardColour :: (Char, Int) -> Bool
cardColour input = help (fst input) where
    help 's' = True
    help 'c' = True
    help  _  = False

case-of:
cardColour' :: (Char, Int) -> Bool
cardColour' input = case fst input of
    's' -> True
    'c' -> True
    _   -> False
```

9 Polimorfizmus

Ugyan a Haskell nem engedi a létező függvények felüldefiniálását, mégis van lehetőségünk általánosabban, nem konkrét típusok segítségével megfogalmazni egy függvény típusszignatúráját és értékét. Erre lesznek segítségünkre a típusparaméterek.

9.1 Parametrikus polimorfizmus

Amikor pusztán típusparaméterekkel határozzuk meg egy vagy több paraméter típusát, tehát nem teszünk semmilyen típusbeli megkötést a paraméterekre, akkor parametrikus polimorfizmust használunk. Egy típusparaméter, amely egy kisbetűvel kezdődő szó/karakter, bármilyen típusra illeszkedhet, éppen ezért a típusspecifikus műveleteket nem is alkalmazhatjuk az értékére. Amint egy típusparaméter illeszkedett egy típusra, onnantól minden további előfordulásában is ugyanazt a típust jelenti. Nézzük például az alábbi függvényt:

```
first :: a -> b -> a
first x _ = x
```

Amennyiben a `first` függvénynek első paraméterül egész számot adunk át, úgy az eredmény is egész szám kell legyen. Amennyiben első paraméterül karaktert adunk át, úgy karakterrel kell visszatérjen, és így tovább.

Ha egy kicsit módosítunk a `first` függvény típusszignatúráján, a helyzet is megváltozik:

```
first :: a -> a -> a
first x _ = x
```

Ebben az esetben már szigorúbb a típusszignatúra, ugyanis elvárja azt is, hogy a két paraméter típusa megegyezzen. Ha mégis két különböző típusú paraméterrel hívjuk meg, akkor hibát fogunk kapni. (Hibaüzenetekről bővebben a 14. fejezetben.)

Két különböző típusparaméter illeszkedhet ugyanarra a típusra is, nem kötelező különbözőnek lenniük. Viszont mivel nem feltétlen jelentik ugyanazt, ezért a függvények definiálásakor különbözőként kell kezelni őket. Vegyük például az alábbi `f` függvényt:

```
f :: a -> b -> b
f x y = x
```

Ezt a definíciót nem fogja elfogadni a fordító, mivel az `a` típus nem feltétlen egyezik a `b` típussal. Fordítási idejű hibát fogunk kapni. (Hibaüzenetekről bővebben a 14. fejezetben.)

9.2 Ad-hoc polimorfizmus

A polimorfizmus egy szigorúbb fajtája az ad-hoc polimorfizmus. Ilyenkor valamilyen megszorítás mentén lesz polimorf a függvény. Haskell nyelvben az ad-hoc polimorfizmus alatt azt értjük, hogy a típusparaméterekre kikötjük, mely típusosztály(ok)ból kell származniuk. Ezzel egyben garantáljuk, hogy a paraméter értékére tudjuk alkalmazni a típusosztály(ok) kötelező függvényeit. Tehát a megkötés(ek)ért cserébe nyerünk is valamit.

Például a hatványozást így lehetne a legáltalánosabban definiálni:

```
power :: (Num a, Num b, Eq b) => a -> b -> a
power x 0 = 1
power x y = x * (power x (y - 1))
```

Ebben a definícióban az `a` típusparaméter csak olyan típusokra tud illeszkedni, amelyek a `Num` és az `Eq` típusosztályból is származnak.

10 Saját típus bevezetése

Ahogy a legtöbb programozási nyelvben, Haskell-ben is van lehetőség saját típusokat definiálni. Háromféle mód adott erre, bevezethetünk típusszinonimát, csinálhatunk új típust egy már létező típusból, és létrehozhatunk saját algebrai adatszerkezeteket is.

10.1 Típusszinonimák (`type`)

Típusszinonimának nevezzük azt, amikor egy meglévő típushoz egy új nevet is társítunk. Ilyenkor nem hozunk létre ténylegesen új típust, a háttérben az új név alatt is az eredeti típus lesz. Ez abban tud segíteni, hogy könnyebben olvasható legyen a program szövege, a paraméterek típusainak adhatunk olyan nevet, amely árulkodik a jelentéséről. Például ha évszámokkal dolgozik a függvényünk, adhatunk az `Int` típusnak egy szinonimát 'Évszám' néven. Ennek szintakszisa az alábbi:

```
type {szinonima neve} = {létezõ típus}

Például:
type Year = Int
type Average = Double
```

10.2 Új típus bevezetése (`newtype`)

Ha nem csupán új nevet szeretnénk adni egy létező típusnak, hanem ténylegesen új típusként szeretnénk bevezetni, akkor a `newtype` kulcsszó lesz a segítségünkre. Amikor a `newtype` segítségével vezetünk be típust, akkor meg kell nevezzünk egy konstruktort, és a konstruktornak paramléterül meg kell adnunk egy létező típust. A konstruktorok **mindig** nagybetűvel kell kezdődjenek.

Példa új típus bevezetésére:

```
newtype Year = Y Int
               ↑
           konstruktor
```

FONTOS, hogy a `newtype` korlátolt, az új típusnak pontosan egy konstruktora, és annak pontosan egy paramétere kell legyen.

10.3 Algebrai adatszerkezetek (`data`)

Amennyiben bonyolultabb adatszerkezetet szeretnénk létrehozni, amelynek több konstruktora is lehet, egy-egy konstruktorának pedig akár több paramétere, úgy a `data` kulcsszó lesz segítségünkre. A `data` kulcsszóval algebrai adatszerkezetet tudunk létrehozni, így a `newtype` korlátai itt nem lesznek érvényesek. Itt is fontos, hogy a konstruktorok **mindig** nagybetűvel kell kezdődjenek.

Példák algebrai adatszerkezetek bevezetésére:

```
data WorkDays = Monday | Tuesday | Wednesday | Thursday | Friday
                  ↑         ↑          ↑          ↑         ↑
                                  konstruktorok
```

Ennél a példánál a munkanapok egy-egy konstruktorként vannak felvéve, és nem tárolnak semmilyen egyéb adatot, nincsen paraméterük. A konstruktorokat mindig a pipeline (`|`) karakterrel választjuk el egymástól.

```
type Height = Int
type Age = Int
data Tree = Tree Age Height
```

Ebben a példában a fák egy általános leképezését adtuk meg. Egy fának van életkora és magassága, ezekre bevezettünk típusszinonimákat, hogy könnyebben értelmezhető legyen.

```
type Age = Int
data Person = Female Age | Male Age
```

Ez a példa az embereket reprezentálja, külön konstruktor van a női nemű és a férfi nemű személyekre, azonban mindkettő esetén tudjuk az illető életkorát.

Egy algebrai adatszerkezet rendelkezhet típusparaméterekkel is. Ezek a típusparaméterek lényegében úgy viselkednek, mint ahogy a parametrikus polimorfizmus (9.1. fejezet) esetében is teszik. Erre egy példa:

```
data Registry a = Registered a | Unregistered
```

Ha egy `Registry` típusú értéket szeretnénk felvenni pl típusszignatúrába, akkor meg kell adnunk a típusparaméter típusát is. Tehát a `Registered 'a'` típusa `Registry Char` lesz.

10.4 Példányosítás típusosztályra (`instance`)

10.4.1 Alapértelmezett példányosítás

A kényelmes és egyszerű módja a saját típusunk valamely típusosztályra való példányosításának a `deriving` kulcsszó használata. A típusszinonimák **nem** számítanak új típusnak. Amikor a `deriving` kulcsszóval példányosítunk, akkor a típusosztály műveleteit egy általános séma mentén definiálja nekünk a fordító. Például az Eq típusosztály esetében a konstruktorok és paramétereik egyenlősége alapján dönti el, hogy két érték egyezik-e.

A `deriving` kulcsszót a típusunk definíciója alá kell írni eggyel beljebb húzva. Amennyiben csak egy típusosztályra szeretnénk példányosítani, úgy nem szükséges zárójelezni, egyébként a típusosztályokat egymástól vesszővel elválasztva egy zárójelben kell átadni. Példa:

```
data WorkDays = Monday | Tuesday | Wednesday | Thursday | Friday
    deriving (Eq, Show)
```

10.4.2 Manuális példányosítás (instance írás)

Vannak helyzetek, amikor nem a megszokott módon szeretnénk példányosítani a saját típusunkat egy típusosztályra, például másképpen szeretnénk kiírni a típusunk értékeit, vagy az egyenlőség vizsgálatánál egy paramétert ki szeretnénk hagyni az ellenőrzésből.

Ahhoz, hogy saját instance-t (példányt) írjunk, fontos tudnunk, hogy a kívánt típusosztálynak mik a minimum követelményei, mely függvényeket kell definiálnunk a saját típusunkra (bővebben a 6. fejezetben). A példányosítás szintakszisa az alábbi:

```
instance {típusosztály neve} {típus neve} where
    {szükséges függvények definíciója egymás alatt}
```

A munkanapokat ábrázoló adatszerkezetünkre például az alábbi módon definiálhatunk saját `Eq` példányt:

```
instance Eq Workdays where
    Monday    == Monday    = True
    Tuesday   == Tuesday   = True
    Wednesday == Wednesday = True
    Thursday  == Thursday  = True
    Friday    == Friday    = True
    _         == _         = False
```

Nem mindig ilyen egyszerű azonban a helyzet. Vegyük például a 10.3. fejezetben bemutatott `Registry` típust. Ha erre a típusra szeretnénk megfelelően működő `Eq` példányt írni, akkor ki kell kötnünk azt is, hogy a típusparaméter csak az `Eq` típusosztályból származhat.

```
instance Eq a => Eq (Registry a) where
    (Registered x) == (Registered y) = x == y
    Unregistered   == Unregistered   = True
    _              == _              = False
```

11 `Maybe` és `Either` típusok

11.1 `Maybe` típus

A `Maybe` egy olyan algebrai adatszerkezet, amelynek van típusparamétere. Két konstruktora a `Just` és a `Nothing`. A `Just` konstruktor tárol egy, a típusparaméternek megfelelő értéket, míg a `Nothing` önmagában áll. A típus definíciója az alábbi:

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

A `Maybe` típust előszeretettel szokás használni parciális függvények totálissá tételére, hiszen a le nem kezelt esetekben elég egy `Nothing` értékkel visszatérni, ahol pedig eddig is volt eredménye a függvénynek, ott az eredményt `Just`-ba csomagolhatjuk.

11.2 `Either` típus

Az `Either` típus a `Maybe` típushoz hasonló, két típusparaméteres algebrai adatszerkezet. Ezt a típust tulajdonképpen úgy kell elképzelni, hogy a két típusparamétere közül **vagy** az egyiket **vagy** a másikat tárolja. Két konstruktora a `Left`, amely az első típusparaméternek megfelelő, és a `Right`, amely a második típusparaméternek megfelelő értéket tárol. A típus definíciója az alábbi:

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

Az `Either` típust akkor szokás használni, hogyha egy függvény definíciójában különböző esetekhez különböző típusú értékeket szeretnénk hozzárendelni.

12 Listák

A legtöbb nyelvben létezik valamilyen konténer típus, mint például a tömbök vagy a sorok. Haskell nyelvben egyirányú láncolt listák vannak alapvetően definiálva. A lista egy **homogén** adatszerkezet, tehát csak ugyanazon típus elemeit tudja tárolni. Amennyiben mégis megpróbálnánk különböző típusú értékeket ugyanabba a listába tenni, úgy fordítási idejű hibába futnánk (hibákról bővebben a 14). Mivel láncolt lista, így számít az elemek sorrendje és multiplicitása is, nem csak egy egyszerű gyűjteményként kell elképzelni. Bármilyen típus értékeit listába lehet foglalni.

A karakterek listája egy speciális eset, van is rá `String` néven egy típusszinonima bevezetve. Karakterek listáját, azaz szöveget, megadhatunk két idézőjel között is, hogy ne kelljen minden egyes karaktert idézőjelek közé tenni, majd egy listáb foglalni.

A listáknak két konstruktora van, az üres lista konstruktora a `[]`, illetve az elem hozzáfűzés konstruktora a `(:)`, amelynek első paramétere egy érték, amely a lista első eleme lesz, második paramétere pedig egy lista, amely az értékkel azonos típusú elemeket tárol (lehet üres is!). Ez egy **rekurzív adatszerkezet**, mivel az elem hozzáfűzésnek második paramétere szintén egy lista, önmagára hivatkozik. A definíciója az alábbi:

```
A Prelude modul definíciója:
data [] a = [] | a : [a]
infixr 5 :

Egy ekvivalens, talán könnyebben érthető változata:
data List a = Nil | Cons a (List a)
```

FONTOS, hogy egy lista lehet végtelen is, nem minden esetben kell valamikor véget érjen. Például a természetes számok halmaza is végtelen. A lusta kiértékelésnek köszönhetően a végtelen listákkal is tudunk dolgozni, hiszen nem kell kivárnunk, míg a teljes lista kiértékelődik, mindig csak azok az elemek számítódnak ki, amelyekre szükségünk van.

FONTOS azonban, hogy a lusta kiértékelés is csak akkor van segítségünkre, hogyha a végtelen listánkat folyamatosan állítjuk elő, tehát például végrekurzió esetén nem lesz a segítségünkre.

12.1 Pont-pont kifejezések (halmaz felírás)

Sűrűn előfordul, hogy egy intervallumon, az abc karakterein vagy hasonló sorrendbe rendezhető halmazokon szeretnénk végighaladni. Ilyenkor nem szükséges az összes elemet felsorolni egy listában, felírhatjuk őket a kezdőértékük, utoló értékük, illetve a lépésszám segítségével.

Az 1-től 100-ig terjedő egész számokat például a `[1..100]` kifejezés adja meg. Nem kötelező megadni utolsó értéket, azonban ennek hiányában végtelen listát állítunk elő. Ilyen például a természetes számok halmaza, amit a `[1..]` kifejezéssel tudunk megadni.

Hogyha csak a páros számokat szeretnénk megadni, akkor már meg kell adnunk a lista második elemét is, hogy megfelelő léptetéssel kerüljenek be az elemek a listába. A páros természetes számokat tehát a `[2,4..]` kifejezés adja meg. Ugyanígy meg kell adnunk a második elemet is, hogyha visszafele szeretnénk számolni. A 10-től 1-ig terjedő egész számokat a `[10,9..1]` kifejezés adja meg.

Természetesen ezek a szabályok érvényesek más típusok esetén is, nem csak a számoknál. Karakterek esetén például karakterkódok sorrendje mentén kell ezeket a szabályokat alkalmazni.

12.2 Listagenerátor/Halmazkifejezés

Van lehetőségünk a matematikai halmazkifejezések mintájára is meghatározni listákat. A listagenerátorok segítségével bonyolultabb halmazokat is meg tudunk adni, mint például a kettő hatványait vagy a pitagoraszi számhármasokat.

Egy matematikai halmazkifejezés a páros természetes számok négyzeteire:

$$\{n^2 | n \in \mathbb{N}, 2|n\}$$

Ugyanez a kifejezés listagenerátorral:

```
[ n ^ 2 | n<-[1..], even n]
```

Látható, hogy még a szintakszis is nagy mértékben egyezik. A listagenerátor bal oldalára (a pipeline (`|`) karakter előtti szakasz) kell megadni a lista egyes számított elemeit. A jobb oldalon tudjuk megadni, milyen listán szeretnénk végighaladni, és az egyes elemekre milyen néven hivatkozhatunk. Emellett a jobb oldalon meghatározhatunk feltételeket is.

FONTOS, hogy egy listaelemre csak akkor írhatunk fel feltételt, ha előtte már felvettük jobb oldalon. Tehát például az alábbi listagenerátor hibát fog eredményezni.

```
[ n ^ 2 | even n, n<-[1..]]
```

A bevett definiálási mód az, hogy először felvesszük az összes listát, amin végig szeretnénk haladni (tehát a generátorokat rájuk), egymástól vesszővel elválasztva. Csak ezután kezdünk el feltételeket meghatározni, a feltételeket szintén fel lehet sorolni, de a logikai `(&&)` (konjunkció) függvénnyel is össze lehet fűzni őket.

```
[ {mi megy a listába} | {generátorok}, {feltételek} ]
```

12.3 Rekurzió listákon

A listákon rekurzívan is végig tudunk haladni, gyakorlaton az egyik legtöbbet gyakorolt feladat a listákon definiált rekurzió. FONTOS azonban odafigyelni arra, hogy amennyiben a feladat nem jelenti ki, hogy csak véges listákra kell működnie, akkor a rekurziót úgy definiáljuk, hogy végtelen listákra is működjön, ne kelljen a végtelenségig várni, folyamatosan generáljuk az eredményt.

Gyakori buktató például a `length` vagy a `genericLength` függvény használata, amely végtelen lista esetén a végtelenségig számol.

12.4 Mintaillesztés listákra

Mivel a listák sokféle típusból tárolhatnak értékeket, illetve ürestől a végtelen hosszúságig terjedhetnek, így a mintaillesztésükre nagyon oda kell figyelni. Alapvetően a két konstruktor mentén tudunk mintát illeszteni az általános mintán (egyszerű elnevezés, pl xs) és a joker karakteren (`_`) felül. Mivel azonban a lista egy rekurzív adatszerkezet, így még a tárolt értékeire való illesztés nélkül is nagyobb mélységig el lehet vinni a mintaillesztést.

FONTOS, hogy a `(:)` első értéke mindig a lista első eleme és második értéke mindig a maradék lista, bárhogy nevezzük el őket. Tehát az `(x:xs)` és az `(xs:xs)` minták közt nincs lényegi különbség.

Néhány példa listákra való mintaillesztésre, és hogy ezek pontosan milyen listákra illeszkednek:

```
[] - üres lista
(x:xs) - legalább egy elemű lista
(x:y:xs) - legalább két elemű lista
((x:xs):xxs) - legalább egy elemű, listákat tartalmazó lista,
               amelynek első eleme legalább egy elemű
[x] - pontosan egy elemű lista
[x,y:ys] - pontosan két elemű, listákat tartalmazó lista,
           amelynek második eleme legalább egy elemű
```

Ezen felül még a tárolt értékekre is lehet tovább illeszteni, amivel tovább komplikálódik a feladat. Fontos megtanulni megkülönböztetni, hogy melyek azok a mintaillesztések, amelyek a listákra vannak, és melyek vonatkoznak a tárolt értékeikre.

13 Magasabbrendű függvények

Sűrűn előfordul, hogy egy problémacsoportra tudunk általános képletet megfogalmazni, amelynek az egyes problémák speciális esetei lesznek. Például a programozási alaptételek használata is előnyünkre lehetne.

Haskell programozási nyelvben függvényt átvehetünk egy másik függvény paramétereként. Tekintsünk erre a lehetőségre úgy, hogy az általános definíciónkat megadhatjuk egy függvényparaméter segítségével. Később a speciális esetek definiálásához pedig átadhatunk különböző függvényeket paraméterül.

Minden olyan függvény, amely függvényt vár paraméterül vagy függvénnyel tér vissza, magasabbrendű függvény. Ezek közül Haskell nyelvben inkább csak azokat nevezzük magasabbrendűeknek, amelyek függvényt várnak paraméterül, hiszen Haskell-ben lényegében minden függvény egy másik függvénnyel tér vissza.

13.1 Szeletek

Magasabbrendű függvények paramétereként sokszor olyan függvényeket adunk át, amelyeket parciális függvényalkalmazás során kapunk, azaz olyan esetben, amikor egy függvényt kevesebb paraméterrel hívunk meg, mint amennyit vár.

A szeletek ezeknek a függvényeknek azon speciális esetei, amikor operátorokat alkalmazunk parciálisan. Ilyen például a `(<4)`, amely igazat fog adni, amennyiben a paramétere kisebb 4-nél. További példák: `(+2)`, `(=='a')`, `(2*)`, `(`mod` 2)`

FONTOS, hogy a `(-4)` nem a 4-gyel való csökkentést jelenti, a 4-gyel való csökkentést a `(+(-4))` szelet adja meg.

13.2 Lambda függvények

Amikor egy egyszerűbb függvényt szeretnénk átadni egy magasabbrendű függvénynek, sokszor úgy érezhetjük, hogy feleslegesen sokat kell írni ahhoz, hogy ezt az egyszerű függvényt egy `where` vagy egy `let-in` blokkban definiáljuk. Sokkal egyszerűbb lenne, ha át tudnánk adni valamilyen "egyszer használatos" függvényt, amit a magasabbrendű függvényünk is tudna értelmezni.

Itt jönnek képbe a lambda függvények, amik valójában a névtelen függvények. Mivel a lambda függvények is függvények, így ugyanazok a szabályok vonatkoznak

rájuk. Ugyanúgy minden paraméter neve kisbetűvel kell kezdődjön, és ugyanúgy figyelni kell a típushelyességre. A lambda függvények szintakszisa az alábbi:

```
(\ {1. param.} {2. param.} ... {utolsó param.} -> {eredmény} )
```

Lambda függvényekben is van lehetőség mintaillesztésre, azonban **csak egy mintára** illeszthetünk, így csak akkor érdemes és ajánlott mintát illeszteni, hogyha az adott típusnak egyetlen konstruktora van, mint például a rendezett n-eseknek.

A lambda függvényeket persze nem csak a magasabbrendű függvényeknek adhatjuk át paraméterül, önmagukban is használhatóak, értelmezhetőek.

13.3 Hajtogatás

Észrevehetjük, hogy bizonyos rekurzív listafeldolgozó függvényeink nagy mértékben hasonló logikával dolgoznak. Ezt a közös logikát egyszerűbben meg tudjuk foglamazni hajtogatás segítségéve. A hajtogatás lényege, hogy egy listából egy bináris (két paraméteres) függvény segítségével valamilyen eredményt "hajtogatunk". Ekkor lépésenként összekombináljuk a lista elemeit a függvény által kiszámított eredménnyel. A végeredményünk nem kell, hogy lista legyen, hajtogatással leírható például az összegzés is.

A hajtogatásnak két iránya van attól függően, hogy a lista melyik végéhez helyezzük a neutrális elemet. A neutrális elem az az érték, amellyel a hajtogatás az üres listát kezeli, tehát amennyiben üres listán szeretnénk hajtogatni, akkor az eredményünk biztosan a neutrális elem lesz. Tudunk balról hajtogatni a `foldl` függvénnyel, illetve jobbról hajtogatni a `foldr` függvénnyel.

A `foldl` és a `foldr` közti egyik legnagyobb különbség, hogy a `foldr` **végtelen listákra is működik**, míg a `foldl` **erre nem képes**, hiszen elsőnek a neutrális elemre és a lista **utolsó elemére** próbálja elvégezni az adott függvényt.

14 Hibaüzenetek

14.1 Fordítási idejű hiba

Fordítási idejű hibának számít minden olyan hiba, amelyet már fordításkor észrevesz a fordítóprogram. A legfőbb ilyen hiba a típushiba, amikor a kapott érték és a várt érték típusa nem egyezik.

Vegyük elő ismét a 9.1. fejezetben bemutatott példát.

```
first :: a -> a -> a
first x _ = x
```

Ebben az esetben a típusszignatúra elvárja, hogy a két paraméter típusa megegyezzen. Ha mégis két különböző típusú paraméterrel hívjuk meg a program szövegében, akkor fordítási idejű hibát fogunk kapni, `error` üzenetet.

```
ghci> :{
ghci| first :: a -> a -> a
ghci| first x _ = x
ghci|
ghci| err :: a -> b -> a
ghci| err x y = first x y
ghci| :}

<interactive>:6:19: error:
    * Couldn't match expected type `a' with actual type `b'
      `b' is a rigid type variable bound by
        the type signature for:
          err :: forall a b. a -> b -> a
        at <interactive>:5:1-18
      `a' is a rigid type variable bound by
        the type signature for:
          err :: forall a b. a -> b -> a
        at <interactive>:5:1-18
    * In the second argument of `first', namely `y'
      In the expression: first x y
      In an equation for `err': err x y = first x y
    * Relevant bindings include
        y :: b (bound at <interactive>:6:7)
        x :: a (bound at <interactive>:6:5)
        err :: a -> b -> a (bound at <interactive>:6:1)
```

Fordítási idejű hiba

14.2 Futási idejű hiba

Futási idejű hibának számít minden olyan hiba, amely csak futási időben üti fel a fejét. Ilyen hiba lehet például, ha egy parciálisan definiált függvényt olyan érték(ek)kel hívunk meg, amelyekre nincsen definiálva.

Vegyük például az alábbi függvényt:

```
example :: Bool -> Int
example True = 1
```

Amennyiben ezt a függvényt a `False` értékkel hívjuk meg, úgy a hívásunk típushelyes, azonban láthatóan nem definiált a függvényünk a `False` értékre, így futási idejű hibát, `exception`-t kapunk.

```
ghci> :{
ghci| example :: Bool -> Int
ghci| example True = 1
ghci|
ghci| err :: Int -> Int
ghci| err n = n + example False
ghci| :}
ghci> err 10
*** Exception: <interactive>:84:1-16: Non-exhaustive patterns in function example
```

Futási idejű hiba

Tartalomjegyzék