Programozási tételek funkcionálisan

Visnovitz Márton

visnovitz.marton@inf.elte.hu
ELTE IK

Absztrakt. A programozás oktatásának egyik alapvető része a programozási tételek megismerése. Ezen általános algoritmusmintákat hagyományosan az imperatív programozási paradigmának megfelelően szekvenciákkal, ciklusokkal és elágazásokkal valósítjuk meg. Ez a cikk bemutatja, hogy ezek a programozási tételek hogyan valósíthatóak meg funkcionális programozás segítségével, bemutatja az egyes tételek esetében felmerülő problémás kérdéseket, illetve rámutat a funkcionális implementáció használatának lehetséges előnyeire.

Kulcsszavak: programozási tételek, funkcionális programozás, programozás alapjai

1. Bevezetés

A programozás oktatásában régóta használatos megközelítés, hogy az alapvető algoritmizálást az úgynevezett programozási tételek alapján oktatjuk. Az emelt szintű érettségin, a vizsgakövetelményeknek [1] megfelelően évről évre olyan feladatok kerülnek kitűzésre, melyek ezen alapvető algoritmusminták köré szerveződnek. A kódolás oktatásának másik jellemzője, hogy elsősorban az imperatív, strukturált programozás van előtérben, habár egyre nagyobb szerepet kapnak az objektum-orientált programozási nyelvek is. Ezt mutatja az is, hogy az érettségin, a diákok több mint 50%-a választott objektum-orientált programozást támogató nyelvet [2].

2. Programozási tételek általánosan

Programozási tételeknek nevezzük azon algoritmusosztályokat, melyek általánosan megoldják azokat a nagy feladatosztályokat, melyekbe a programozási feladatok besorolhatóak. Ezeket a tételeket tovább lehet csoportosítani a bemenet és a kimenet alapján, az alábbi módon [3]:

1. *egy* sorozathoz *egy* értéket rendelő feladatok
2. *egy* sorozathoz *egy* sorozatot rendelő feladatok
3. *egy* sorozathoz *több* sorozatot rendelő feladatok
4. *több* sorozathoz *egy* sorozatot rendelő feladatok

Habár különböző források más-más programozási tételeket különböztetnek meg [3, 4], ez a cikk a Szlávi Péter és Zsakó László által meghatározott 6 + 6 tétel [3] közül az első tizenegyet vizsgálja. A tizenkettediket, az *összefuttatás* tételt kihagyjuk, mivel az az *uniónak* egy speciális esete, melyben az előfeltétel erősebb megkötései miatt létezhet egy hatékonyabb, párhuzamos feldolgozáson alapuló megoldás a feladatra.

A tételek valamely azonos típusú elemek sorozatára, vagy több ilyen sorozatra van definiálva. A továbbiakban ezt a típust H jelöli. A logikai típust L-lel jelöljük. Az algoritmusminták egy része tartalmaz egy olyan $H \to L$ típusú függvényt, mely egy adott sorozat egy elemére vizsgál egy tulajdonságot. Ezt a függvény a továbbiakban T tulajdonságnak nevezzük. A vizsgált tizenegy tétel az alábbi:

- **Sorozatszámítás:** Adott típusú elemek sorozatához rendel hozzá egy értéket, melyet egy, az adott típus két elemére definiált $(H, H) \rightarrow L$ függvény sorozatos kiszámítása állít elő.
- **Eldöntés:** Adott típusú elemek sorozatához rendel egy logikai értéket, mely meghatározza, hogy a sorozatban létezik-e egy T tulajdonságú elem.
- **Kiválasztás:** Adott típusú elemek sorozatához rendeli az első olyan elemet, vagy annak indexét, mely elem T tulajdonságú. A tételben előfeltétel, hogy létezik legalább egy ilyen elem.
- **(Lineáris) keresés:** Lényegét tekintve megegyezik a kiválasztás tétellel, de ebben az esetben nem garantált, hogy létezik T tulajdonságú elem, így ezt is vizsgálni kell.
- **Megszámolás:** Adott típusú elemek sorozatához rendel egy darabszámot, mely megadja, hogy a sorozat hány eleme T tulajdonságú.
- **Maximumkiválasztás:** Adott típusú elemek sorozatából megadja a legnagyobb (több esetén az egyik legnagyobb) T tulajdonságú elemet vagy annak indexét.
- **Másolás:** Adott típusú elemek sorozatából olyan sorozatot állít elő, melynek minden eleme az eredeti sorozat azonos indexű elemének valamely $H \rightarrow H'$ függvény általi átalakításával jön létre.
- **Kiválogatás:** Megadja adott típusú elemek sorozatának minden T tulajdonságú elemét egy sorozatban.
- **Szétválogatás:** Egy sorozat elemeit valamely T tulajdonság szerint két sorozatra válogatja, az egyikben az eredeti sorozat azon elemei szerepelnek, melyekre teljesül T, a másikban azok, melyekre nem. Több T tulajdonság esetén általánosítható K db sorozatra történő szétválogatásra is.
- **Metszet:** Kettő vagy több, azonos típusú elemekből álló sorozatból kiválogatja azokat az elemeket, melyek minden sorozatban szerepelnek.
- **Unió:** Kettő vagy több azonos típusú elemekből álló sorozatból létrehoz egy olyan sorozatot, melyben minden elem szerepel, amely bármelyik eredeti sorozatban szerepelt, de minden elem csak egyszer kerül bele az új sorozatba.

Ezen tizenegy tétel határozza meg azokat a feladatosztályokat, melyekbe az egyszerűbb programozási feladatok tartoznak, és melyek fontos szerepet játszanak az algoritmizálás tanításában, az algoritmikus gondolkodás fejlesztésében [5]. A továbbiakban azt vizsgáljuk, hogy mindezen tételek hogyan valósíthatóak meg funkcionális programozási minták segítségével a hagyományos, imperatív megközelítéssel szemben.

3. Imperatív és funkcionális megvalósítás

A programozási tételeknek több különböző funkcionális programozási implementációját is megadjuk. Ezek a funkcionális programozás két különböző módszerét, a hajtogatást (*folding*) és a magasabb rendű függvényeket (*higher order functions*) használják [6]. A kódok bemutatásához a TypeScript [7] programozási nyelvet választottam, mert ez támogatja az imperatív és a funkcionális programozást is. A TypeScript sablonok segítségével lehetővé teszi az általános, típustól független tételek definiálását, és a szintaxisa is elég kifejező ahhoz, hogy könnyen érthető legyen a programkód mind a két paradigmában. A bemutatott kódrészletekben a tételeket egy tetszőleges H típusra fogalmazzuk meg. A feldolgozó függvények (f) és a tulajdonságfüggvények (T) szignatúrája is mindenhol erre az általános H típusra van definiálva. A TypeScript nyelv a példák bemutatásán túl akár az oktatásban is használható lehet [8]. Minden tétel esetében legalább háromféle megvalósítás szerepel az alábbi sorrendben:

1. *imperatívan*: szekvenciákkal, ciklusokkal és elágazásokkal,
2. *funkcionálisan, hajtogatással*: a bemenő sorozat felbontása első elemre, és a további elemek sorozatára, és ezt felhasználva rekurzív kiszámítás,
3. *funkcionálisan, magasabb rendű függvényekkel*: tömbfüggvények segítségével történő kiszámítás.

A funkcionális megoldások nem tartalmaznak szekvenciális utasításokat, ehelyett egyetlen kifejezésként adják meg az eredményt. A hajtogatásos implementációkban látható, hogy a bemenő `x` paramétert egy tömbfelbontással (`[x0, ...xs]`) [9] átalakítjuk egy `x0` kezdőelemre, és egy `xs` tömbre, ami a további elemeket tartalmazza. A rekurzív számításos megoldásban a rekurzió végét egy feltételes kifejezés formájában (`a ? b : c`) adjuk meg. Ebben az `x0 === undefined` feltétel azt vizsgálja, hogy a felbontás után létezik-e első elem, vagyis elértük-e már a sorozat végét a feldolgozással. Magasabb rendű függvények használata esetén a tételek tömbre alkalmazott tömbfüggvény(ek) segítségével valósíthatóak meg.

3.1 Sorozatszámítás

```
sorSzam = <H> (x: H[], f: (s: H, e: H) => H, f0: H): H
```

1. ábra A sorozatszámítás tétel szignatúrája

A *sorozatszámítás* tételt megvalósító függvény (`sorSzam`) szignatúráját az 1. ábra mutatja be TypeScript nyelven.

Paraméterek

- `x`: a bemeneti *H* típusú elemek sorozata,
- `f`: két *H* típusú elemet egyesítő művelet (pl. +, -, ∨, ∪, stb.),
- `f0`: az adott művelet neutrális eleme (pl. összeadás esetén 0).

Megvalósítás

```
sorSzam = <H> (x: H[], f: (s: H, e: H) => H, f0: H): H => {
  let s: H = f0
  for (let i: number = 0; i < x.length; ++i) {
    s = f(s, x[i])
  }
  return s
}
```

```
sorSzam = <H> ([x0, ...xs]: H[], f: (s: H, e: H) => H, f0: H): H =>
  x0 === undefined ? f0 : f(sorSzam(xs, f, f0), x0)
```

```
sorSzam = <H> (x: H[], f: (s: H, e: H) => H, f0: H): H =>
  x.reduce(f, f0)
```

2. ábra A sorozatszámítás tétel megvalósításai

3.2 Eldöntés

```
eldont = <H> (x: H[], T: (e: H) => boolean): boolean
```

3. ábra Az eldöntés tétel szignatúrája

Az *eldöntés* tételt megvalósító függvény (`eldont`) szignatúráját a 2. ábra mutatja be.

Paraméterek

- `x`: a bemeneti *H* típusú elemek sorozata,
- `T`: tetszőleges tulajdonságfüggvény (logikai értéket állít elő egy *H*-beli elemből).

Megvalósítás

```
eldont = <H> (x: H[], T: (e: H) => boolean): boolean => {
  let i: number = 0
  while (i < x.length && ! T(x[i])) {
    i += 1
  }
  return i < x.length
}
```

```
eldont = <H> ([x0, ...xs]: H[], T: (e: H) => boolean): boolean =>
  x0 === undefined ? false : T(x0) || eldont(xs, T)
```

```
eldont = <H> (x: H[], T: (s: H) => boolean): boolean =>
  x.filter(T).length > 0
```

4. ábra Az eldöntés tétel megvalósításai

Magasabb rendű függvényekkel a megvalósítás (4. ábra) a feladatot a *kiválogatás* tételre vezeti vissza. Ez az implementáció akkor is feldolgoz minden elemet, ha a sorozatban előbb is megállapítható lenne az eldöntés eredménye. Egy másik megoldás lehetne, ha a TypeScript beépített `.some()` metódusát használnánk, ami pontosan az eldöntés tételt valósítja meg (5. ábra).

```
eldont = <H> (x: H[], T: (e: H) => boolean): boolean =>
  x.some(T)
```

5. ábra A sorozatszámítás tétel megvalósítása kiválogatással

3.3 Kiválasztás

```
kivalaszt = <H> (x: H[], T: (e: H) => boolean): H
```

6. ábra A kiválasztás tétel szignatúrája

A *kiválasztás* tételt megvalósító függvény (`kivalaszt`) szignatúráját a 6. ábra mutatja be. A példa a tétel érték-kiválasztásos változatához tartozik.

Paraméterek

- `x`: a bemeneti *H* típusú elemek sorozata,
- `T`: tetszőleges tulajdonságfüggvény (logikai értéket állít elő *H*-beli elemből).

Megvalósítás

```
kivalaszt = <H> (x: H[], T: (e: H) => boolean): H => {
  let i: number = 0
  while (!T(x[i])) {
    i += 1
  }
  return x[i]
}
```

```
kivalaszt = <H> ([x0, ...xs]: H[], T: (e: H) => boolean): H =>
  T(x0) ? x0 : kivalaszt(xs, T)
```

```
kivalaszt = <H> (x: H[], T: (e: H) => boolean): H =>
  x.filter(T).shift()
```

7. ábra A kiválasztás tétel megvalósításai

A magasabb rendű függvényekkel való megoldásban (7. ábra) két tömbfüggvény (szűrés és első elem leválasztása) egymásutáni alkalmazása adja meg az eredményt. Ez a megoldás abból a szempontból kevésbé hatékony a többihez képest, hogy mindenképp feldolgozza a sorozat összes elemét, még akkor is, ha az első elem teljesíti a kiválasztási feltételt. A JavaScript nyelv tartalmazza a `.find()` tömb metódust, mely rövidebben és hatékonyabban oldja meg ezt a problémát, de a TypeScript nyelvnek ez még hivatalosan nem része, ezért használtuk ezt az összetettebb változatot.

3.4 Lineáris keresés

```
keres = <H> (x: H[], T: (e: H) => boolean): H | boolean
```

8. ábra A lineáris keresés szignatúrája

A *lineáris keresés* tételt megvalósító függvény (`keres`) szignatúrája (8. ábra) és megvalósítása (9. ábra) is nagyon hasonlít az előzőkben bemutatott *kiválasztás* tételhez. A különbség az, hogy ebben az esetben nem tudhatjuk biztosan, hogy létezik *T* tulajdonságú elem a listában. A tétel specifikációja szerint [3] a kimenet egy adott elem (vagy annak indexe) és egy logikai érték, ami azt mutatja meg, hogy van-e egyáltalán megfelelő elem a sorozatban. A legtöbb programozási nyelvben egy függvény visszatérési értéke vagy adott típusú vagy tetszőleges típusú. Ez az oka annak, hogy sok nyelvben valamilyen speciális érték (pl. -1) jelzi, hogy a keresés nem talált megfelelő elemet. A TypeScript lehetőséget nyújt arra, hogy egy függvény visszatérési értéke több, különböző típusba is tartozhasson. Ennek megfelelően a `keres` visszatérési típusa *H* vagy logikai. A példa az érték-kiválasztásos változatot mutatja be.

Paraméterek

- `x`: a bemeneti *H* típusú elemek sorozata,
- `T`: tetszőleges tulajdonságfüggvény (logikai értéket állít elő egy *H*-beli elemből).

Megvalósítás

```
keres = <H> (x: H[], T: (e: H) => boolean): H | boolean => {
  let i: number = 0
  while (i < x.length && !T(x[i])) {
    i += 1
  }
  if (i < x.length) {
    return x[i]
  } else {
    return false
  }
}
```

```
keres = <H> ([x0, ...xs]: H[], T: (e: H) => boolean): H | boolean =>
  x0 === undefined ? false : (T(x0) ? x0 : keres(xs, T))
```

```
keres = <H> (x: H[], T: (e: H) => boolean): H | boolean =>
  x.some(T) ? x.filter(T).shift() : false
```

9. ábra A lineáris keresés megvalósításai

A harmadik változat kiszámítása jelentősen egyszerűbb, ha *T* tulajdonságú elemet nem tartalmazó sorozatra nem a `false`, hanem az `undefined` értékkel térünk vissza. Ebben az esetben a magasabb rendű függvényekkel dolgozó megvalósítás megegyezik a *kiválasztás* tételével. A jelenlegi megoldás `.some()` metódushívása kiváltható `.filter()`-rel az eldöntés mintájára.

3.5 Megszámolás

```
megszamol = <H> (x: H[], T: (e: H) => boolean): number
```

10. ábra A megszámolás tétel szignatúrája

A *megszámolás* tételt megvalósító függvény (`megszamol`) szignatúráját a 10. ábra tartalmazza.

Paraméterek

- `x`: a bemeneti *H* típusú elemek sorozata,
- `T`: tetszőleges tulajdonságfüggvény (logikai értéket állít elő egy *H*-beli elemből).

Megvalósítás

```
megszamol = <H> (x: H[], T: (e: H) => boolean): number => {
  let db: number = 0
  for (let i: number = 0; i < x.length; ++i) {
    if (T(x[i])) {
      db += 1
    }
  }
  return db
}
```

```
megszamol = <H> ([x0, ...xs]: H[], T: (e: H) => boolean): number =>
  x0 === undefined ? 0 : (T(x0) ? 1 : 0) + megszamol(xs, T)
```

```
megszamol = <H> (x: H[], T: (e: H) => boolean): number =>
  x.filter(T).length
```

11. ábra A megszámolás tétel megvalósításai

A megszámolás tétel magasabb rendű függvényekkel történő megvalósítására két különféle megoldást adunk. Az első megoldás (11. ábra) egy szűrés (ld. *kiválogatás* tétel) segítségével ad megoldást.

```
megszamol = <H> (x: H[], T: (e: H) => boolean): number =>
  x.reduce((s: number, e: H) => T(e) ? s + 1 : s, 0)
```

12. ábra A megszámolás tétel megvalósítása sorozatszámítással

A 12. **ábra** ábrán látható másik implementáció a sorozatszámítás tételre vezethető vissza. Ebben a `.reduce()` metódus függvényparaméterét lambda-függvényként definiáltuk.

3.6 Maximumkiválasztás

```
maximum = <H> (x: H[], c: (a: H, b: H) => boolean): H
```

13. ábra A maximumkiválasztás tétel szignatúrája

A *maximumkiválasztás* tételt megvalósító függvény (`megszamol`) szignatúrája a 13. ábraán látható módon írható le. Ez a változat a maximális értéket, nem pedig annak indexét adja vissza. A *maximumkiválasztás* tétel különbözik a többi tételtől abban, hogy itt nincs értelmes eredmény üres bemen-

ti sorozat esetén. Ezt a megvalósítások úgy kezelik, hogy üres listára `undefined` értéket adnak eredményül.

Paraméterek

- `x`: a bemeneti *H* típusú elemek sorozata,
- `c`: összehasonlító függvény, mely két *H* típusú elemre megadja, hogy az első nagyobb-e mint a második.

Megvalósítás

```
maximum = <H> (x: H[], c: (a: H, b: H) => boolean): H => {
  let max: H = x[0]
  for (let i: number = 1; i < x.length; ++i) {
    if (c(x[i], max)) {
      max = x[i]
    }
  }
  return max
}
```

```
maximum = <H> ([x0, ...xs]: H[], c: (a: H, b: H) => boolean): H =>
  xs.length == 0 ? x0 : (c(x0, maximum(xs, c)) ? x0 : maximum(xs, c))
```

```
maximum = <H> (x: H[], c: (a: H, b: H) => boolean): H =>
  x.reduce((s: H, e: H) => c(e, s) ? e : s, x[0])
```

14. ábra A maximumkiválasztás tétel megvalósításai

A *maximumkiválasztás* tétel is többféleképpen oldható meg magasabb rendű függvényekkel. Az első példa (14. ábra) a *sorozatszámítás* tételre történő visszavezetéssel ad megoldást.

```
maximum = <H> (x: H[], c2: (a: H, b: H) => number): H =>
  x.sort(c2).pop()
```

15. ábra A maximumkiválasztás tétel megvalósítása rendezéssel

A második változat (15. ábra) egy rendezéssel oldja meg a feladatot. Ehhez módosítani kell az összehasonlító függvényt, mivel a TypeScript beépített `.sort()` eljárása egy olyan függvényt vár paraméterül, ami egy pozitív értékkel tér vissza, ha az első paraméter a nagyobb, negatívval, ha a második, és nullával, ha egyenlőek. Egy ilyen összehasonlító függvény esetén egyszerűbb kódot kapunk. Számok sorba állítására egyszerűen írható ilyen függvény, ha az a két szám különbségét adja eredményül. Szövegek sorba rendezését a `.sort()` metódus paraméter nélküli változata végzi. Ez a megoldás nagy adatmennyiségre lényegesen lassabb (gyorsrendezés esetén n helyett $n \cdot log(n)$ bonyolultságú).

3.7 Másolás

```
masol = <H, M> (x: H[], f: (e: H) => M): M[]
```

16. ábra A másolás tétel szignatúrája

A *másolás* tételt megvalósító függvény (`masol`) szignatúrájában (16. ábra) egy új típus (*M*) jelenik meg, mivel *másolás* tétel leképező függvénye (f) nem feltétlenül ugyanolyan típusú elemeket állít elő, mint amik eredetileg szerepeltek a sorozatban, például amikor szövegek sorozatából előállítjuk az egyes szövegek hosszát.

Paraméterek

- `x`: a bemeneti *H* típusú elemek sorozata,
- `f`: a másolás tétel leképezését végző függvény, mely egy *H* típusú elemből *M* típusút állít elő.

Megvalósítás

```
masol = <H, M> (x: H[], f: (e: H) => M): M[] => {
  let x2: M[] = []
  for (let i: number = 0; i < x.length; ++i) {
    x2[i] = f(x[i])
  }
  return x2
}
```

```
masol = <H, M> ([x0, ...xs]: H[], f: (e: H) => M): M[] =>
  x0 === undefined ? [] : [f(x0), ...masol(xs, f)]
```

```
masol = <H, M> (x: H[], f: (e: H) => M): M[] =>
  x.map(f)
```

17. ábra A másolás tétel megvalósításai

3.8 Kiválogatás

```
kivalogat = <H> (x: H[], T: (e: H) => boolean): H[]
```

18. ábra A kiválogatás tétel szignatúrája

A *kiválogatás* tételt megvalósító függvény (`kivalogat`) szignatúrája (18. ábra) hasonlít a kiválasztás tételéhez és a keresés tételéhez. A különbség az, hogy nem csak egy darab *T* tulajdonságú elemet keresünk, hanem az összeset.

Paraméterek

- `x`: a bemeneti *H* típusú elemek sorozata,
- `T`: tetszőleges tulajdonságfüggvény.

Megvalósítás

```
kivalogat = <H> (x: H[], T: (e: H) => boolean): H[] => {
  let y: H[] = []
  for (let i: number = 0; i < x.length; ++i) {
    if (T(x[i])) {
      y.push(x[i])
    }
  }
  return y
}
```

```
kivalogat = <H> ([x0, ...xs]: H[], T: (e: H) => boolean): H[] =>
  x0 === undefined ? [] : (T(x0)
    ? [x0, ...kivalogat(xs, T)]
    : kivalogat(xs, T))
```

```
kivalogat = <H> (x: H[], T: (e: H) => boolean): H[] =>
  x.filter(T)
```

19. ábra A kiválogatás tétel megvalósításai

3.9 Szétválogatás

```
szetvalogat = <H> (x: H[], T: (e: H) => boolean): [H[], H[]]
```

20. ábra A szétválogatás tétel szignatúrája

A *szétválogatás* tétel a *kiválogatásnak* az az esete, amikor a *nem T* tulajdonságú elemeket is kiválogatjuk egy másik sorozatba. Ez a tétel abban különbözik a többitől, hogy két sorozattal tér vissza. TypeScript nyelven ezt a `tuple` típus segítségével lehet megvalósítani, ami egy olyan tömb, melynek fix számú eleme van, és ezen elemek a típusa külön-külön meghatározott. A megvalósító függvény (`szetvalogat`) szignatúráját a 20. ábra mutatja.

Paraméterek

- `x`: a bemeneti *H* típusú elemek sorozata,
- `T`: tetszőleges tulajdonságfüggvény.

Megvalósítás

```
szetvalogat = <H> (x: H[], T: (e: H) => boolean): [H[], H[]] => {
  let y: H[] = []
  let z: H[] = []
  for (let i: number = 0; i < x.length; ++i) {
    if (T(x[i])) {
      y.push(x[i])
    } else {
      z.push(x[i])
    }
  }
  return [y, z]
}
```

```
szetvalogat = <H> ([x0, ...xs]: H[],
                   T: (e: H) => boolean): [H[], H[]] =>
  x0 === undefined ? [[], []] :
    T(x0)
      ? [[x0, ...szetvalogat (xs, T)[0]], szetvalogat (xs, T)[1]]
      : [szetvalogat (xs, T)[0], [x0, ...szetvalogat (xs, T)[1]]]
```

```
szetvalogat = <H> (x: H[], T: (e: H) => boolean): [H[], H[]] =>
  [x.filter(T), x.filter(e => !T(e))]
```

21. ábra A szétválogatás tétel megvalósításai

A *szétválogatás* hajtogatásos megoldásánál (21. ábra) az eredményt megadó kifejezésben négy különböző helyen is látható rekurzív hívás. Ez TypeScriptben nem hatékony, mivel nem tisztán funkcionális nyelv révén ugyanaz a kifejezés többször is kiértékelésre kerül (a függvény-kiértékelés nem lusta, mint például a Haskell nyelvben [10]), így rengeteg felesleges számítást végez a számítógép. Az is látható, hogy a magasabb rendű függvényekkel történő megoldásnál, két külön tömbfüggvény adja a megoldást. Ez azt is jelenti, hogy itt ténylegesen két kiválogatás tétel egymás utáni alkalmazásáról van szó. Ez szintén kevésbé hatékony kódot eredményez. Ezek a hatékonyságbeli különbségek

azonban egy kezdő programozó szintjén nem meghatározóak, csupán módszertani szempontból lehet fontos ezekre felhívni a figyelmet.

A második kiválogatás feltétele az eredeti ellentettje, ezt a példában lambda kifejezéssel adjuk meg. A tétel megvalósítható a sorozatszámításra visszavezetve is (22. ábra). TypeScript kód esetében ügyelni kell a típusok helyességére, mivel az automatikus típusfelismerő nem helyesen határozza meg a `[H[], H[]]` típusú kifejezések típusát. Ezeket a típusokat külön jelezni kell az `as` kulcsszóval.

```
szetvalogat = <H> (x: H[], T: (e: H) => boolean): [H[], H[]] =>
  x.reduce(([x1, x2]: [H[], H[]], x0: H) =>
    T(x0)
      ? ([[...x1, x0], x2] as [H[], H[]])
      : ([x1, [...x2, x0]] as [H[], H[]]),
    [[], []])
```

22. ábra A szétválogatás tétel megvalósítása sorozatszámításra visszavezetve

3.10 Metszet

```
metszet = <H> (x: H[], y: H[]): H[]
```

23. ábra A metszet tétel szignatúrája

A *metszet* tétel több sorozatot tartalmaz bemenetként, és ezekből állít elő egy sorozatot. A tételnek előfeltétele, hogy mind a két sorozat ismétlődésmentes legyen, mivel a metszet, mint művelet halmazokra értelmes. A megvalósítások szempontjából ez azért lényeges, mert mindhárom implementáció azon az elven alapul, hogy az egyik sorozat elemeiből válogatja ki azokat, melyek a másik sorozatban is szerepelnek. Ha az első sorozatban ismétlődés van, akkor az így az eredményben is megjelenne, de ha a paraméterek sorrendjét megcseréljük, akkor nem. A megvalósító függvény (`metszet`) szignatúráját a 23. ábra mutatja.

Paraméterek

- `x1`, `x2`: a bemeneti *H* típusú elemek sorozatai.

Megvalósítás

```
metszet = <H> (x: H[], y: H[]): H[] => {
  let z: H[] = []
  for (let i: number = 0; i < x.length; ++i) {
    let j: number = 0
    while (j < y.length && y[j] !== x[i]) {
      j += 1
    }
    if (j < y.length) {
      z.push(x[i])
    }
  }
  return z
}
```

```
metszet = <H> ([x0, ...xs]: H[], y: H[]): H[] =>
  x0 === undefined ? [] :
    (y.some((e) => e === x0)
      ? [x0, ...metszet(xs, y)]
      : metszet(xs, y))
```

```
metszet = <H> (x: H[], y: H[]): H[] =>
  x.filter((a) => y.some((b) => b === a))
```

24. ábra A metszet tétel megvalósításai

A *metszet* tétel magasabb rendű függvényekkel történő megvalósításában (24. ábra) két egymásba ágyazott lambda függvényt használtunk. Ez lényegében egy keresés tétel, amiben a *T* tulajdonság egy egyenlőségvizsgálat. Ahogy az már az *eldöntés* tételnél szerepelt, a TypeScript nyelv még nem tartalmazza hivatalosan a `.find()` tömb metódust, mellyel rövidebb megoldást kaphatnánk. Még a `.find()`-nál is elegánsabb lenne a JavaScript `.includes()` tömb metódusát használni, mely azt vizsgálja, hogy egy elem szerepel-e egy másik tömbben. A TypeScript jelenleg még ezt sem támogatja, csak a megfelelő kiegészítésekkel.

3.11 Unió

```
metszet = <H> (x: H[], y: H[]): H[]
```

25. ábra A metszet tétel szignatúrája

A *metszethez* hasonlóan az *unió* tételnek fontos előfeltétele, hogy mind a két bemeneti sorozat ismétlődésmentes legyen. Ha ez a kritérium nem teljesül, akkor a paraméterek sorrendjétől függően változhat az eredmény, mivel a megoldás alapja, hogy az egyik sorozathoz hozzátesszük azokat az elemeket a másikból, amik még nem szerepelnek benne. Ennek megfelelően, ha az egyik sorozatban ismétlődés van, akkor az eredményben is szerepelhet az ismétlődés, ha azt használjuk kiinduló sorozatnak. Ha a másik sorozatból indulunk ki, akkor az ismétlődések kimaradnak. Az *unió* tételt megvalósító függvény (`unio`) szignatúrája (25. ábra) megegyezik a *metszet* tételéével (23. ábra).

Paraméterek

- `x1`, `x2`: a bemeneti *H* típusú elemek sorozatai.

Megvalósítás

```
unio = <H> (x: H[], y: H[]): H[] => {
  let z: H[] = []
  for (let i: number = 0; i < x.length; ++i) {
    let j: number = 0
    while (j < y.length && x[i] !== y[j]) {
      j += 1
    }
    if (j === y.length) {
      z.push(x[i])
    }
  }
  z = [...z, ...y]
  return z
}
```

```
unio = <H> ([x0, ...xs]: H[], y: H[]): H[] =>
  x0 === undefined ? y :
    (y.some((e) => e === x0)) ? unio(xs, y) : [x0, ...unio(xs, y)]
```

```
unio = <H> (x: H[], y: H[]): H[] =>
  [...x.filter((a) => !y.some((b) => b === a)), ...y]
```

26. ábra Az unió tétel megvalósításai

Az unió tétel – hasonlóan a metszethez – az elöntés tételre vezethető vissza. A különbség ebben az, hogy az eldöntés feltétele megfordul, vagyis itt azt akarjuk eldönteni, hogy egy adott elem hiányzik-e a másik sorozatból. Ennek megfelelően az *unió* az első sorozat minden eleméből és a második sorozat azon elemeiből épül fel, melyek az elsőben nem szerepelnek. Mind a három implementáció (26. ábra) erre az alapelvre épül. A magasabb rendű függvényes megvalósítás itt is egyszerűsíthető lenne a JavaScript `.find()` vagy `.includes()` metódusaival, vagy megvalósítható az eldöntés tételnél bemutatott `.filter(T).length > 0` megoldással is.

Tételek csoportosítása funkcionális megvalósítás alapján

A tizenegy vizsgált programozási tétel (feladatosztály) mindegyike megoldható magasabb rendű függvények segítségével. Ezekben a megoldásokban a `.reduce()`, `.filter()` és `.map()` tömbfüggvények valamelyikét használtuk. Ezek a metódusok egy az egyben megfelelnek a *sorozatszámítás*, *kiválogatás*, és *másolás* tételeknek. Ha a programozási tételeket ezen funkcionális megvalósítások alapján csoportosítjuk, akkor három kategória jön létre (1. táblázat).

`.reduce()` **sorozatszámítás**	`.filter()` **kiválogatás**	`.map()` **másolás**
sorozatszámítás	eldöntés	másolás
maximumkiválasztás	kiválasztás	
	megszámolás	
	lineáris keresés	
	kiválogatás	
	metszet	
	unió	

1. táblázat A tételek csoportosítása funkcionális megvalósítás alapján

A táblázatból látható, hogy egyedül a *másolás* tétel tartozik a `.map()` metódussal megoldott tételek közé. Ezt a kategóriát azért tartjuk meg mégis önállóan, mert a másolás/leképezés alkalmazása nagyon gyakori művelet a funkcionális programozás világában.

Mint ahogy azt néhány példában láthattuk is a `.filter()` és a `.map()` metódus is visszavezethető a `.reduce()` használatára (27. ábra). Ez is igazolja, hogy a vizsgált programozási tételek mind visszavezethetőek a sorozatszámításra, vagyis igazából egy általános programozási tétel van [4, 11].

```
filter = <H, M> (x: H[], T: (e: H) => boolean) =>
  x.reduce((s: H[], e: H) => T(e) ? [...s, e] : s, [])
```

```
map = <H, M> (x: H[], f: (e: H) => M) =>
  x.reduce((s: H[], e: H) => [...s, f(e)], [])
```

27. ábra A `.filter()` és `.map()` függvények visszavezetése `.reduce()`-ra

Konklúzió

A hagyományos, imperatív programozásra építő módszertanban nagyon nagy hangsúlyt kap az alapvető algoritmusminták használatának megtanítása. A programozási tételek alkalmazásakor a tételek saját algoritmusa, és azon belül a tényleges feladatot megoldó feldolgozó vagy tulajdonságfüggvény hasonló hangsúllyal szerepelnek. Ezzel szemben a funkcionális megoldások magasabb rendű függvényeket használó változatánál a tétel algoritmusa háttérbe szorul, és így a tényleges feladat kerül előtérbe.

Gyakorlatban ez azt jelentheti, hogy ha ezt a megoldást alkalmazzuk, akkor a tanulóknak több ideje marad a tényleges feladatok megoldására és kevésbé kell magukra a tételekre koncentrálni. Ezáltal lehetségessé válhat több, komplexebb feladat összeépítése, mely pozitívan hathat a diákok érdeklődésére is.

Habár ezek a megvalósítások bizonyos esetekben futási idő és erőforrások szempontjából kevésbé hatékony megoldást adnak, ez egy kezdő programozó esetében nem feltétlenül elsődleges szempont. Ezzel szemben a korai programozási élmények kialakításában fontos lehet az, hogy az egyszerű alkalmazhatóság révén könnyebbé válhat a programok elkészítése.

Irodalom

[1] *Informatika Részletes érettségi vizsgakövetelmények.* Oktatási Hivatal, 2017. https://www.oktatas.hu/pub_bin/dload/kozoktatas/erettsegi/vizsgakovetelmenyek2017/informatika_vk_2017.pdf (utoljára megtekintve: 2017.11.12.)

[2] *Statisztika az emelt szintű érettségin választott programozási környezetekről.* Oktatási Hivatal, 2013-2016.

[3] Szlávi Péter, Zsakó László: *Módszeres programozás: Programozási tételek.* ELTE Informatikai Kar, 2008.

[4] Gregorics Tibor: *Programozás 1. kötet Tervezés.* ELTE Eötvös Kiadó, 2013.

[5] Szlávi Péter, Zsakó László: *Informatika oktatása.* 2011. http://tamop412.elte.hu/tananyagok/infokt (utoljára megtekintve: 2017. 11. 08.)

[6] Horváth Zoltán: *A funkcionális programozás nyelvi elemei.* Nyékyné Gaizler Judit (szerk.): Programozási nyelvek. Kiskapu Kiadó, 2003.

[7] *TypeScript – JavaScript that scales.* Microsoft, 2017. https://www.typescriptlang.org (utoljára megtekintve: 2017. 11. 08.)

[8] Horváth Gy., Menyhárt L.: *Teaching introductory programming with JavaScript in higher education.* Proceedings of the 9th International Conference on Applied Informatics, Eger, 2014. 1. kötet, 339-350. oldal

[9] *Destructuring assignment - JavaScript | MDN.* Mozilla, 2017. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment (utoljára megtekintve: 2017. 11. 12.)

[10] *Haskell language.* Haskell.org, 2017. https://www.haskell.org (utoljára megtekintve: 2017. 11. 13.)

[11] Szlávi Péter, Törley Gábor, Zsakó László: *Programming Theorems Have the Same Origin* Veronika Stoffová, Román Horváth (szerk.): XXX. DIDMATTECH. Travana University, Faculty of Education, 2017. ISBN 978-80-568-0029-4. 52-58. oldal