1. táblás gyakorlat – specifikáció

Specifikáció és visszavezetés

Újdonságok:

A specifikáció[1] a **feladat** megadásának *egy lehetséges* eszköze, aminek egy formájával már tavaly, progalapból megismerkedhettünk.

A félév első felében ezt fogjuk kissé általánosabb módon, aztán a félév második felében még általánosabb módon használni.

Lássuk a különbségeket az összegzés tétellel illusztrálva:

Régi

Be:

$$N \in \mathbb{N}, X \in Tömb[1..N, \mathbb{Z}]$$

Ki:

$$S \in \mathbb{Z}$$

Ef:

$$-$$

Uf:

$$S = \sum_{i=1}^{N} X[i]$$

Új

Feltevések:

Adott egy $\mathbb{H}$ halmaz, amin értelmezett egy $+$ jelű asszociatív, baloldali semleges elemes művelet
Adott egy $f: [m..n] \to \mathbb{H}$ függvény

Állapottér:

$$m: \mathbb{Z}, n: \mathbb{Z}, s: \mathbb{H}$$

Ef:

$$m = m' \wedge n = n'$$

Uf:

$$Ef \wedge s = \sum_{i=m}^{n} f(i)$$

Általánosítás:

Először is, vannak bizonyos jelölésbeli különbségek (kisbetűs változónevek, tömb típus jelölése, stb…)

Másrészt, eddig, ha nagyon pontosak akartunk lenni (de nem igazán akartunk), akkor elvileg mindig csak tömbön tudtuk a tételeinket „futtatni”, mostantól pedig mindig egész számok egy $[m..n]$ intervallumát járjuk be, amit alkalmasan $[1..n]$-nek megválasztva egy lehetséges tömb-indextartományt kapunk. Tehát az új modell általánosabb.

Ezen kívül most fontos kiemelni, hogy az összegzés bármilyen olyan halmazra értelmes, ami rendelkezik valamilyen „aggregálható” függvénnyel. Így a $+$ csak egy *függvényszimbólum*, ami lehet az összeadás is, de egyáltalán nem kötelező, hogy az legyen. Lehet, hogy ez a szorzás, lehet, hogy a $\mathbb{H}$ az a szavak halmaza, ekkor a $+$ pedig a szövegösszefűzés, stb.

Az az adott f függvény, ami az intervallumot a $\mathbb{H}$-ra képzi pedig a létező legáltalánosabb. Ez lehet például a progalap X tömbje, hiszen ha adott egy tömb és adott az indextartománya, akkor miért ne lehetne az f függvény olyan, hogy az adott tömb adott indexű elemét rendeli

[1] A való életben is „specifikációnak” szokták hívni azt a dokumentumot, amelyet többé-kevésbé a megrendelővel egyetértésben készítünk és rögzítjük benne az elkészítendő termékre vonatkozó főbb feltételeket. Ebben a legkülönbözőbb szempontok szerepelhetnek, és nem feltétlenül a lehető legformálisabban. Kritikusabb részfeladatoknál esetleg előfordulhat az itt tanultakhoz hasonló – vagy akár még ennél is részletesebb specifikáció (pl. ki kell térni az adatbekérés módjára, párhuzamosítási lehetőségekre, a programmal szemben támasztott hatékonysági-gyorsasági feltételekre, stb.

minden indexhez. Viszont ez az f függvény teljesen másmilyen is lehet, ezért általánosabb, mint a progalapos verzió.

Az $[m..n]$ intervallum tetszőleges *egész* intervallum lehet – akár üres is, ha előfeltételben külön nem kötöttük ki, akkor ennek is értelemszerűen működnie kell.

Visszavezetés:

Amikor majd erre a nagyon absztrakt, nagyon általános sémára megpróbáljuk ráhúzni a konkrétabb feladatunkat, azt a folyamatot fogjuk tételre való *visszavezetés*nek hívni. Feltesszük, hogy a tétel specifikációjában megfogalmazott absztrakt feladatnak megoldása a tétel struktogramjában megadott absztrakt program. Ekkor, ha a visszavezetés során minden előforduló elemet képesek vagyunk következetesen megfeleltetni a konkrét specifikációnkban és struktogramunkban, akkor biztosak lehetünk benne, hogy a konkrét struktogram a konkrét feladat megoldását adja meg.

Ezáltal, amennyiben a tételeket tudjuk, a feladat szabatos megfogalmazása után egy mechanikus, kevesebb kreativitást igénylő módszerrel elő tudunk állítani garantáltan helyes programokat. Ehhez a tételek szószerinti ismerete szükséges, hiszen különben nem tudhatnánk, konkrétan melyik változóknak kell megfeleltetni az aktuális megoldandó feladat változóit.

A specifikáció részei:

Mostantól minden „külső körülményt” formálisan meg kell majd adni. Segédfüggvényeket, adattípusokat, halmazokat, konstansokat.

Ezt követően a majdani megoldó program a *feladat* szempontjából *lényeges* állapotait értékükkel meghatározó változók listáját adjuk meg. Ez a lista tartalmazza a változók nevét és típusát. Ezt hívjuk *állapottér*nek.

Lényegében az állapottérbe azok a változók kerülnek, amelyek kezdeti értékétől függ a feladat és/vagy amelyek majdani értéke jelenti a feladat megoldását. Tehát minden olyan változót fel kell itt sorolni, amit eddig a „ki”, vagy „be” listákban soroltunk fel. Ezekre a változókra tehetünk majd megszorításokat az előfeltételben, ezekre a változókra fogalmazhatunk meg elvárásokat az utófeltételben, valamint ezeket a változókat (is) használhatjuk a feladatot megoldó programban.

Az állapottér elemei ellentétben a tavalyi megközelítéssel, most hagyományos, állapottal, konkrét aktuális értékkel rendelkező *változó*k. Az előfeltétel minden esetben ezen változók feletti elsőrendű logikai állítás, és azt fejezi ki, hogy az algoritmus végrehajtása előtt mi minden legyen igaz a változók aktuális értékére, állapotára. Ha nem várunk el semmit, akkor azt írjuk, „igaz”, ekkor bármi is a kezdeti értéke a változóinknak, a program értelmezve lesz. Ha ennél szigorúbbat írunk, akkor minden olyan állapotra, ami kívül esik az előfeltétel igazsághalmazán a program nem köteles helyesen működni. Mi arra vállalunk garanciát, hogy olyan programot írunk, mely az előfeltételnek megfelelő kezdeti állapotok mindegyikére garantáltan eljuttat az utófeltételnek megfelelő végállapotok egyikébe.

Az előfeltétel tehát az állapottér *minden* komponensétől függő logikai állítás, mely a feladat értelmezési tartományát adja meg. Ha úgy gondoljuk, hogy valamelyik változónk nem „bemenő szemantikájú”, hogy ettől nem függ a feladat (azaz progalapból ezt a változót a „ki” rovatba írtuk volna), akkor egyszerűen erről a változóról nem fogalmazunk meg semmilyen elvárást. Ennek a változónak ekkor is lesz valamilyen *kezdeti értéke*! Csak éppen minket ez

nem érdekel, mert mondjuk tudjuk, hogy a programunk úgy is azzal fogja kezdeni a végrehajtását, hogy kinullázza ezt a változót.

A struktogramba sose kerül beolvasásról és kiírásról szóló rész. Oda csak a változókat lekérdező, vagy azokat módosító utasítások kerülnek. Mégis, a specifikáció alapján tudunk következtetni arra, hogy majd a C++ megvalósításban pontosan hogy fogalmazzuk meg az adatbekérést és eredménykiírást. Minden „bemenő" (azaz előfeltételben szereplő) változót az algoritmus futtatása előtt bekérünk. A bekérés során biztosítanunk kell az adott változó típusát (ezt az állapottérből tudjuk kiolvasni) és az adott változóra vonatkozó előfeltételeket. Ha a felhasználó olyan adatokat adott meg, amelyekre nem teljesül, akkor vagy széttárjuk a kezünket és közöljük, hogy ilyen bemenetre a programnak nem kell futnia, vagy esetleg egy do-while ciklussal addig idegesítjük szerencsétlen felhasználót, amíg vagy nem ír be megfelelő adatot, vagy nem dobja ki a számítógépét az ablakon.

$$A = (a{:}\,\mathbb{N}, b{:}\,\mathbb{N})$$

$$ef = (a = a' \wedge 2|a)$$

$$uf = (ef \wedge b = a/2)$$

Eme példát pl. így kódolhatnánk le (main függvény):

```
int a,b; //az int az ugyan nem természetes szám, de még mindig ez van
hozzá a legközelebb
cout<<”ird be a-t”<<endl;
cin>>a;    //csak a-t kérjük be, mert róla vannak megkötések az
előfeltételben, b-ben nyugodtan lehet memóriaszemét is
if(cin.fail() || a<0)
{
    cerr<<”nem tipushelyes a bemenet”<<endl;;
    exit(1);     //valami hibajelzéssel kiléphetünk, mert innen nem
}                //garantálnánk, hogy a program értelmesen működne, hisz
if(a mod 2 == 1)//sérülnek a feltételek
{
  cerr<<”nem teljesul az elofeltetel”<<endl;;
  exit(2);       //természetesen nem muszáj más hibajelzést adni, a két
}                //ellenőrzést össze is vonhattam volna
b = a/2;   //ez maga az algoritmus, ami reményeink szerint előállítja az
utófeltételben elvárt állapotokat
cout<<b<<endl;  //mert b a kimenő érték az utófeltétel alapján
return 0;       //ez meg a ”minden rendben volt” típusú visszatérés
```

(tehát ha a egy páros természetes szám, akkor b-be számoljuk ki annak felét, ami szintén egy természetes szám lenne, ha nem, akkor nem kell csinálnunk semmit, azon kívül persze, hogy tájékoztatjuk a felhasználót… de ez már nem algoritmikus kérdés, ezért a struktogramban nem fog sosem szerepelni.)

Láthatunk az előfeltételben még egy újdonságot. Ez az a=a' rész. Ez ha kiolvassuk, mindössze annyit jelent, hogy „az előfeltétel akkor igaz (egyebek mellett), ha az a értéke kezdetben éppen a'". Igen, de mi az az a'?

Mivel az előfeltételben más szó nem esik az a'-ről, ezért igazából bármi lehet. A lényeg az, hogy egyenlő a-val. Akkor mire jó ez az egész? Arra jó, hogy ezt az a'-t mint a specifikáció szintjén bevezetett segéd-konstansértéket az utófeltételben is használhatom. Ne feledjük, a

egy változó, annak állapota van. Az előfeltételben (a beolvasás után) és az utófeltételben (a programvégrehajtás után) értéke eltérő lehet! Mi van akkor ha én azt szeretném kifejezni, hogy `a` egy tisztán bemenő változó és emiatt értéke ne változzon, vagy mi van akkor, ha én azt szeretném kifejezni, hogy `b` értéke legyen `a` eredeti értékének a fele? Progalapból ezzel nem volt gond, mert egyszerűen ami bemenő érték volt, az a végrehajtás után is ugyanaz az érték volt. De most ez egyáltalán nem biztos.

$$A = (a{:}\,\mathbb{N}, b{:}\,\mathbb{N})$$

$$ef = (igaz)$$

$$uf = (b = a)$$

Ezt a programot az `a,b:=0,0` szimultán értékadás megoldja. Ha én azt akarom, hogy `b` értéke `a` eredeti értékével legyen egyenlő, akkor ezt kell írnom:

$$A = (a{:}\,\mathbb{N}, b{:}\,\mathbb{N})$$

$$ef = (a = a')$$

$$uf = (b = a')$$

Ha pedig azt szeretném, hogy `a` értéke se változzon[2], akkor pedig ezt:

$$A = (a{:}\,\mathbb{N}, b{:}\,\mathbb{N})$$

$$ef = (a = a')$$

$$uf = (a = a' \wedge b = a')$$

Vegyük észre: amik eddig tisztán bemenő változók voltak, azok mostantól úgy jelennek meg, hogy az előfeltételben „szó van róluk”, de mindenképpen „elmentjük” egy specifikációs segédkonstansba a pillanatnyi értéküket, és az utófeltételben ehhez a konstanshoz viszonyítunk.

A kimenő változókról pedig az előfeltételben nem szólunk, az utófeltételben pedig valamilyen a bemenő változók kezdeti értéke alapján előálló formulába helyettesítjük be őket.

Ha egy változó tisztán bemenő akkor tehát „megtartjuk az előfeltételt” az utófeltételben, és ezt általában úgy jelöljük (mivel így a legrövidebb), ahogy az összegzés példájában is láttuk, hogy az utófeltételt így kezdjük: $ef \wedge \ldots$, tehát minden olyan kikötés, ami kezdetben a bemenőkre igaz volt, az a bemenőkre a végén is legyen igaz. Arról nem szól a fáma, hogy mi van az eleje és a vége között, de nyilván nem fogunk olyan programot írni, ami beszorozza kettővel a bemenő változót, hogy azután elossza kettővel. Tehát ha egy változót bemenőnek látunk, akkor azt vehetjük úgy, hogy a kódban a beolvasáson kívül „békén hagyjuk”, ezzel garantálva azt, hogy $a = a'$ akkor is fennálljon a végrehajtás után, hogy a'-t egyálalán nem is írhatunk a kódba vagy a struktogramba (hiszen az egy specifikációs segédkonstans).

---

[2] Általában azt szoktuk szeretni, hogy a bemenő változók értéke ne változzon. De nem is olyan magától értetődő, hogy általános esetben ez miért is jó. Nem komoly megszorítás ez? Nem. Ugyanis ha majd megoldó programot írunk, úgyis a bemenő változók eredeti értékétől fog minden függni. Ekkor pedig kénytelenek leszünk ezeket elraktározni valahova. Mennyivel egyszerűbb azt mondani, hogy egyszerűen békén hagyom a bemenőket, mint hogy gondoskodjak valami segédváltozóról, amiben eltárolom a’-t. És mivel a legtöbb feladat megoldható így is úgy is, ezért egyáltalán nem jelent komoly megszorítást ez az implementáció szempontjából komoly segítség.

A félév első felében tanult tételeknél egyébként végig ez lesz: meg fogjuk tartani az előfeltételt.

Ám! Ez az egész hűhó arra volt jó, hogy ne legyen *feltétlenül* szükséges megtartani az előfeltételt. Képzeljük el azt a feladatot, ahol a bemenő adatot a végigolvasása után ki kell törölni, módosítani kell, netán pl. egy tömböt helyben rendezni. Ekkor szigorúan bemenő adatokkal (progalapos modell) csak a fejünket vakarhatjuk, de nem jutunk előbbre. Most viszont csak annyit kell tennünk, hogy nem tartjuk meg az utófeltételben az előfeltételt (kizárólag azt a részét, ami a valóban tisztán bemenő változókra vonatkozik, ha vannak ilyenek), hanem a kimenő változók mellett a módosítandó bemenő változókról is valamilyen logikai állítást teszünk, ami persze az összes bemenő változó eredeti értékétől függ majd. Ezek valamivel bonyolultabb feladatok, a félév második felében az általánosítás miatt végig ilyeneket fogunk nézni.

Még egyszer a visszavezetésről:

A feladatok között felírhatunk egy érdekes kapcsolatot. *Szigorúbb* az a feladat, amelynek az utófeltétele szigorúbb:

$$A = (a{:}\,\mathbb{N}, b{:}\,\mathbb{N}) \qquad A = (a{:}\,\mathbb{N}, b{:}\,\mathbb{N})$$
$$ef = (a = a') \qquad ef = (a = a')$$
$$uf = (ef \wedge b = a) \qquad uf = (ef \wedge b|a)$$

Itt a bal oldali nyilván szigorúbb, mert **szigorúbb** az utófeltétele. Azaz minden olyan program, ami ezt megoldja, az megoldja a lazább jobb oldalit is, hiszen a az nyilván a osztója is egyben.

Elmondható, hogy azonos állapottér és előfeltétel mellett, ha $uf_1 \to uf_2$, akkor az 1. feladat szigorúbb, mint a 2.

Hasonlóan, ha azonos állapottér és utófeltétel mellett egy **gyengébb** előfeltétellel rendelkező feladat is *szigorúbb*, mint a másik:

$$A = (a{:}\,\mathbb{N}, b{:}\,\mathbb{N}) \qquad A = (a{:}\,\mathbb{N}, b{:}\,\mathbb{N})$$
$$ef = (a = a') \qquad ef = (a = a' \wedge a > 0)$$
$$uf = (ef \wedge b|a) \qquad uf = (ef \wedge b|a)$$

Itt a jobb oldali feladat előfeltétele többet állít, ez egy szigorúbb feltétel. Tehát a jobb oldali feladat programjának elég kevesebb lehetséges állapotra is jól futnia, míg a baloldali jóval általánosabb… Elmondható tehát az is, hogy azonos állapottér és utófeltétel mellett, ha $ef_2 \to ef_1$, akkor az 1. feladat **szigorúbb**, mint a 2.

A második példához nagyon hasonló viszont így is fel tudunk írni:

$$A = (a{:}\,\mathbb{N}, b{:}\,\mathbb{N}) \qquad A = (a{:}\,\mathbb{N}, b{:}\,\mathbb{N}^+)$$
$$ef = (a = a') \qquad ef = (a = a')$$
$$uf = (ef \wedge b|a) \qquad uf = (ef \wedge b|a)$$

Ez alapján elmondható, hogy azonos előfeltétel és utófeltétel mellett ha $A_2 \subseteq A_1$, akkor az 1. feladat *szigorúbb*, mint a 2.

Miért érdekes ez?

Tegyük fel, hogy adott egy futóverseny eredménye. Kíváncsiak vagyunk arra, ért-e valaki el pontosan 1 perces eredményt. Nyilván egy „lineáris keresés” tételt alkalmazunk, végigmegyünk az adatsoron, megnézzük az időpontokat, ha lett, lett, ha nem, nem. De mi van akkor, ha minket csak az olimpiai 100 m-es síkfutás hoz lázba? Tegyük fel, hogy a versenyről azt is tudjuk, hogy hol és milyen távon futották. Ekkor szigoríthatjuk az

előfeltételt, hogy nekünk csak az olimpiai versenyek jók. Minden más, pl. a megoldó algoritmus is marad. És mivel az előfeltételt szigorúbbá tettük, ezáltal a feladatot gyengébbé tettük. És mivel az eredeti erősebb feladatra adott megoldásunk helyes, ezért nyilván egy gyengített feladatra is helyes ugyanaz a megoldás.

Most azt képzeljük el, hogy esetleg valakik holtversenyben futottak kereken 1 perces eredményt. A lineáris keresést, ha megnézzük, az azt mondja, hogy megkeresi az *első* előfordulását annak az adatnak, amire igaz a keresett feltétel. Azaz ha az adatsorban X futó 1 perces ideje egy fentebbi sorban szerepel, mint Y futóé, akkor garantáltan X-et hozza ki megoldásnak, míg nekünk ez aztán édes mindegy, mert egyszerre értek be… Miért is kell kikötnünk azt, hogy a tömbbeli első egy perces eredményt adjuk vissza? Jó nekem a második is. Vagy bármelyik. Hát hagyjuk el az utófeltételből azt a részt, hogy „minden korábbi eredmény nem egy perces időadatot tartalmaz”. Az algoritmust meg minden mást hagyjuk meg ugyanígy[3]. Ezzel gyengítettük az utófeltételt, azaz gyengítettük magát a feladatot, de az eredeti megoldásunk nyilván továbbra is megoldja az új feladatot is.

És pontosan így működik a visszavezetés. Adott egy absztrakt általános tételünk, ami mondjuk tetszőleges H halmazra képes összegezni. De nekünk mondjuk elég „csak” természetes számokra, tehát szűkítettük az állapotteret, ezáltal gyengébb feladatot kaptunk. Mivel az absztrakt tételt bebizonyították, ezért a gyengített tétel is helyes lesz.

Miket lehet használni a specifikációban?

Az állapottérben változók listáját kell megadni típussal. Nevük bármi lehet, persze mindnek egyedi. A típusuk pedig az alábbiakból kerülhet ki:

- $\mathbb{Q}$, vagy annak részhalmazai ($\mathbb{Z}, \mathbb{N}, \mathbb{N}^+, [a..b]$[4] ahol $a, b \in \mathbb{Z}$ rögzített konstans, stb.)
- $String$
- $\mathbb{L}$ (azaz igaz-hamis)
- Saját típusok, amennyiben definiáljuk mi az (pl. az első oldalon látható $\mathbb{H}$, vagy a rekord[5]-ok)
- a fentiek *gyűjtemény*ei: halmaz, tömb, sorozat, szekvenciális inputfájl, mátrix, …

Az előfeltétel és az utófeltétel egy az állapottért a logikai értékekre képző *elsőrendű logikai formulá*val megadható logikai függvény. Az állapottér változóin kívül *kvantált* formában szerepelhetnek még mindenféle szimbólumok, illetve szerepelhetnek bevezetett konstansok, függvények, „megvesszőzött” bemenő változónevek…

Példa a helyes…

$$A = (a{:}\,\mathbb{N})\ ef = (a = a' \wedge \forall x \in [1..a]{:}\,\exists y \in [1..x]\,\ldots)$$

…és a helytelen kvantált formulára

$$A = (a{:}\,\mathbb{N})\ ef = (a = a' \wedge b = b' \wedge \forall x \in [1..c]{:}\,\exists y \in [1..z]\,\ldots)$$

Mert se `b`, se `c`, se `z` nincs az állapottéren, a bevezetett konstansok között és kvantor mögött se.

---

[3] ezáltal persze nyilván ugyanaz az algoritmus továbbra is ugyanazt az első előfordulást fogja megadni, csak ezt mostantól mi nem várjuk el

[4] az `[a..b]`-t mindig úgy kell érteni, hogy `a` és `b` két akár negatív egész szám, `b` lehet kisebb, mint `a` (üres intervallum) és ilyenkor az `a` és `b` közötti (`a`-t és `b`-t is beleértve!) *egész* számok halmazára gondolunk. pl.: `[-3..2] = {-3,-2,-1,0,1,2}`

[5] C++-ban `struct`