Gregorics Tibor

PROGRAMOZÁS VISSZAVEZETÉSSEL

egyetemi jegyzet[1]

2011

[1] A jegyzet tananyagának kialakítása az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósult meg (a támogatás száma TÁMOP 4.2.1./B-09/1/KMR-2010-0003)

TARTALOM

BEVEZETÉS

„*A programozás az a mérnöki tevékenység, amikor egy feladat megoldására programot készítünk, amelyet azután számítógépen hajtunk végre.*” E látszólag egyszerű meghatározásnak a hátterében több figyelemre méltó gondolat rejtőzik.

Az egyik az, hogy a programkészítés célja egy feladat megoldása, ezért a programozás során mindig a kitűzött feladatot kell szem előtt tartanunk; abból kell kiindulnunk, azt kell elemeznünk, részleteiben megismernünk.

A másik gondolat az, hogy a program fogalma nem kötődik olyan szorosan a számítógéphez, mint azt hinnénk. A programozás valójában egymástól jól elkülöníthető – bár a gyakorlatban összefonódó – résztevékenységekből áll: először kitaláljuk, megtervezzük a programot (ez az igazán nehéz lépés), aztán átfogalmazzuk, azaz kódoljuk azt egy számítógép számára érthető programozási nyelvre, végül kipróbáljuk, teszteljük. Nyilvánvaló, hogy az első a lépésben, a programtervezésben szinte kizárólag a megoldandó feladatra kell összpontosítanunk, és nem kell, nem is szabad törődnünk azzal, hogy a program milyen programozási környezetbe ágyazódik majd be. Nem veszhetünk el az olyan kérdések részletezésében, hogy mi is az a számítógép, hogyan működik, mi az operációs rendszer, mi egy programozási nyelv, hogyan lehet egy programot valamely számítógép számára érthetővé, végrehajthatóvá tenni. A program lényegi része ugyanis – nevezetesen, hogy a feladatot megoldja – nem függhet a programozási környezettől.

A programtervezést egy olyan folyamatnak tekintjük, amely során a feladatot újabb és újabb, egyre részletesebb formában fogalmazzuk meg, miáltal egyre közelebb kerülünk ahhoz a számítógépes környezethez, amelyben majd a feladatot megoldó program működik. Ehhez azt vizsgáljuk, hogyan kell egy feladatot felbontani részekre, majd a részeket továbbfinomítani egészen addig, amíg olyan egyszerű részfeladatokhoz jutunk, amelyekre már könnyen adhatók azokat megoldó programok. Ennek a könyvnek a gyakorlati jelentősége ezeknek a finomító (más szóval részletező, részekre bontó, idegen szóval dekomponáló) technikáknak a bemutatásában rejlik.

A programtervezésnek és a programozási környezetnek a szétválasztása nemcsak a programozás tanulása szempontjából hasznos, hanem a programozói mesterségnek is elengedhetetlen feltétele. Aki a programozást egy konkrét programozási környezettel összefüggésben sajátítja el, az a megszerzett ismereteit mind térben, mind időben erősen korlátozottan tudja csak alkalmazni. Az állandóan változó technikai környezetben ugyanis pontosan kell látni, melyik ismeret az, amit változtatás nélkül tovább használhatunk, és melyik az, amelyet időről időre „le kell cserélnünk”. A programtervezésnek az alapelvei, módszerei, egyszóval a gondolkodásmód nem változik olyan gyorsan, mint a programozási környezet, amelyben dolgoznunk kell.

A bevezető mondathoz kapcsolódó harmadik gondolat a mérnöki jelzőben rejlik. E szerint a programkészítést egy világosan rögzített technológia mentén, szabványok alapján kell végezni. A legösszetettebb feladat is megfelelő elemzés során felbontható olyan részekre, amelyek megoldására már rendelkezünk korábbi mintákkal. Ezek olyan részmegoldások, amelyek helyessége, hatékonysága már bizonyított, használatuk ezért biztonságos. Kézenfekvő, hogy egy új feladat megoldásához ezeket a korábban már bevált mintákat felhasználjuk, ahelyett, hogy mindent teljesen elejéről kezdjünk el kidolgozni. Ha megfelelő technológiát alkalmazunk a minták újrafelhasználásánál, akkor a biztonságos mintákból garantáltan biztonságos megoldást tudunk építeni.

Ebben a kötetben a programtervezést állítottam középpontba: a feladatok specifikációjából kiindulva azokat megoldó absztrakt programokat készítünk. Célom egy olyan programozási módszertan bemutatása, amelyik feladat orientált, elválasztja a program tervezését a terv megvalósításától, és a tervezés során programozási mintákat követ. A programozási mintáknak egy speciális családjával foglalkozom csak, mégpedig algoritmus mintákkal vagy más néven a programozási tételekkel.

I. RÉSZ
PROGRAMOZÁSI FOGALMAK

Ebben a részben egy programozási modellt mutatunk be, amelynek keretében a programozással kapcsolatos fogalmainkat vezetjük be. Ez valójában egy matematikai modell, de itt kerülni fogjuk a pontos matematikai definíciókat, ugyanis ennek a könyvnek nem célja a programozási modell matematikai aspektusainak részletes ismertetése. Ehelyett sokkal fontosabb meglátni azt, hogy ez a modell egy sajátos és nagyon hasznos gondolati hátteret szolgáltat a kötetben tárgyalt programtervezési módszereknek.

Fontos például azt megérteni – és ebben ez a matematikai modell segít –, hogy mit tekintünk megoldandó feladatnak, mire kell egy feladat megfogalmazásában figyelni, hogyan érdemes az észrevételeinket rögzíteni. Tisztázni kell továbbá, hogy mi egy program, hogyan lehet azt általános eszközökkel leírni úgy, hogy ez a leírás ne kötődjön konkrét programozási környezetekhez (tehát absztrakt programot írjunk), de könnyedén lehessen belőle konkrét programokat készíteni. Választ kell kapnunk arra is, mikor lesz helyes egy program, azaz mikor oldja meg a kitűzött feladatot. E kötet későbbi részeiben alkalmazott különféle programtervezési módszereknek ugyanis ebben az értelemben állítanak elő garantáltan helyes programokat. Végül, de nem utolsó sorban, a programozási modell jelölés rendszere alkalmas eszközt nyújt a tervezés dokumentálására.

Az *1. fejezet* a programozási modell alapfogalmait vezeti be. A *2. fejezet* egy a feladatok leírására szolgáló formát, a feladat specifikációját mutatja be. A *3. fejezet* a strukturált program fogalmát definiálja, és vázolja, hogy hogyan lehet matematikai korrektséggel igazolni azt, hogy egy strukturált program megold egy feladatot. A fejezet végén fogalmazzuk meg azt az igényünket, hogy egy program ne csak helyes legyen, hanem megengedett is, amely azt garantálja, hogy egy absztrakt programot egyszerűen kódolhassunk egy konkrét programozási környezetben.

1. Alapfogalmak

Induljunk ki a bevezetés azon megállapításából, amely szerint a *programozás egy olyan tevékenység, amikor egy feladat megoldására programot készítünk*. Ennek értelmében a programozás három alapfogalomra épül: a feladat, a program és a megoldás fogalmaira. Mi az a feladat, mi az a program, mit jelent az, hogy egy program megold egy feladatot? Ebben a fejezetben erre adjuk meg a választ.

1.1. Az adatok típusa

Egy feladat megoldása azzal kezdődik, hogy megismerkedünk azzal az információval, amelyet a probléma megfogalmazása tartalmaz. Nem újdonság, nemcsak a programozási feladatokra érvényes, hogy a megoldás kulcsa a feladat kitűzése során megjelenő információ értelmezésében, rendszerezésében és leírásában rejlik.

Feladatokkal mindenki találkozott már. A továbbiakban felvetődő gondolatok szemléltetése kedvéért most egy „iskolai" feladatot tűzünk ki, amely a következőképpen hangzik: „*Egy nyolc kilogrammos, zöld szemű sziámi macska sétálgat egy belvárosi bérház negyedik emeleti ablakpárkányán. Az eső zuhog, így az ablakpárkány síkos. A cica megcsúszik, majd a mintegy 12.8 méteres magasságból lezuhant. Szerencsére a talpára esik! Mekkora földet éréskor a (becsapódási) sebessége?*"

A feladat által közvetített információ első pillantásra igen összetett. Vannak lényeges és lényegtelen elemei; szerepelnek közöttük konkrét értékeket hordozó adatok és az adatok közötti kapcsolatokat leíró összefüggések; van, ami közvetlenül a feladat szövegéből olvasható ki, de van, ami rejtett, és csak a szövegkörnyezetből következtethetünk rá.

A „macskás" feladatban lényegtelen adat például a macska szemének színe, hiszen a kérdés megválaszolása, azaz a becsapódási sebesség kiszámolása szempontjából ez nem fontos. Lényeges adat viszont a magasság, ahonnan a cica leesett. A kinematikát jól ismerők számára nyilvánvaló, hogy egy zuhanó test becsapódási sebességének kiszámolásához nincs szükség a test tömegére sem. Feltételezve ugyanis azt, hogy a macskának pontszerű teste a Földön csapódik be, a becsapódás sebességét – a feladatban közvetlenül nem említett, tehát rejtett – $v = \sqrt{2 * g * h}$ képlet (h a magasság, g a nehézségi gyorsulás) segítségével számolhatjuk ki. Itt a nehézségi gyorsulás is egy rejtett adat, amelynek az értékét állandóként *10 m/s*^{2}-nek vehetjük, ennek megfelelően a fenti képlet $v = \sqrt{20 * h}$ -ra módosul.

Abban van némi szabadságunk, hogy a feladat szövegében nem pontosan leírt elemeket hogyan értelmezzük. Például a nehézségi gyorsulást is kezelhetnénk bemeneti adatként ugyanúgy, mint a magasságot. Sőt magát a becsapódási sebesség kiszámításának képletét is be lehetne adatként kérni annak minden paraméterével (pl. nehézségi gyorsulással, súrlódási

együtthatóval, stb.) együtt, ha a problémát nemcsak a fent leírt egyszerűsítés (pontszerű test zuhanása légüres térben) feltételezése mellett akarnánk megoldani Ezek és az ehhez hasonló eldöntendő kérdések teszik a feladat megértését már egy ilyen viszonylag egyszerű példánál is bonyolulttá.

A feladat elemzését a benne szereplő lényeges adatok összegyűjtésével kezdjük, és az adatokat a tulajdonságaikkal jellemezzük. Egy adatnak igen fontos tulajdonsága az értéke. A példában szereplő magasságnak a kezdeti értéke *12.8* méter. Ez egy úgynevezett ***bemenő (input) adat***, amelynek értékét feltétlenül ismerni kell ahhoz, hogy a feltett kérdésre válaszolhassunk. Kézenfekvő a kitűzött feladatnak egy olyan általánosítása, amikor a bemeneti adathoz nem egy konkrét értéket rendelünk, hanem egy tetszőlegesen rögzített kezdeti értéket. Ehhez fontos tudni azt, hogy milyen értékek jöhetnek egyáltalán szóba; melyek lehetnek a bemeneti adatnak a kezdőértékei. A példánkban szereplő magasság adat egy nem-negatív valós szám.

Furcsának tűnhet, de adatnak tekintjük az eredményt is, azaz a feladatban feltett kérdésre adható választ. Ennek konkrét értékét nem ismerjük előre, hiszen éppen ennek az értéknek a meghatározása a célunk. A példában a becsapódási sebesség egy ilyen – az eredményt megjelenítő – úgynevezett ***eredmény vagy kimeneti (output) adat***, amelynek értéke kezdetben definiálatlan (tetszőleges), ezért csak a lehetséges értékeinek halmazát adhatjuk meg. A becsapódási sebesség lehetséges értékei is (nem-negatív) valós számok lehetnek.

Az adatok másik jellemző tulajdonsága az, hogy az értékeikkel műveleteket lehet végezni. A példánkban a valós számokhoz kapcsolódó műveletek a szokásos aritmetikai műveletek, a feladat megoldásánál ezek közül a szorzásra és a négyzetgyökvonásra lesz szükségünk. Az adatokról további jellemzőket is össze lehetne gyűjteni (mértékegység, irány stb.), de a programozás szempontjából ezek nem lényegesek.

Egy adat típusát az adat által felvehető lehetséges értékek halmaza, az úgynevezett típusérték-halmaz, és az ezen értelmezett műveletek, az úgynevezett típusműveletek együttesen határozzák meg, más szóval specifikálják[2].

Tekintsünk most néhány nevezetes adattípust, amelyeket a továbbiakban gyakran fogunk használni. A típus és annak típusérték-halmazát többnyire ugyanazzal a szimbólummal jelöljük. Ez nem okoz később zavart, hiszen a szövegkörnyezetből mindig kiderül, hogy pontosan mikor melyik jelentésre is gondolunk.

- ***Természetes szám típus***. Típusérték-halmaza a természetes számokat (pozitív egész számokat és a nullát) tartalmazza, jele: $\mathbb{N}$. ($\mathbb{N}^+$ a nullát nem tartalmazó természetes számok halmazát jelöli.) Típusműveletei az összeadás (+), kivonás (–), szorzás (*), az egész osztás (*div*), az

[2] A típus korszerű definíciója ennél jóval árnyaltabb, de egyelőre ez a „szűkített” változat is elegendő lesz az alapfogalmak bevezetésénél. A részletesebb meghatározással a *7. fejezetben* találkozunk majd.

osztási maradékot előállító művelet (*mod*), esetleg a hatványozás, ezeken kívül megengedett még két természetes szám összehasonlítása (=, ≠, <, ≤, >, ≥).

- ***Egész szám típus***. Típusérték-halmaza (jele: $\mathbb{Z}$) az egész számokat tartalmazza ($\mathbb{Z}^+$ a pozitív, $\mathbb{Z}^-$ a negatív egész számok halmaza), típusműveletei a természetes számoknál bevezetett műveletek.
- ***Valós szám típus***. Típusérték-halmaza (jele: $\mathbb{R}$) a valós számokat tartalmazza ($\mathbb{R}^+$ a pozitív, $\mathbb{R}^-$ a negatív valós számok halmaza, $\mathbb{R}_0^+$ a pozitív valós számokat és a nullát tartalmazó halmaz), típusműveletei az összeadás (+), kivonás (–), szorzás (*), osztás (/), a hatványozás, ezeken kívül megengedett két valós szám összehasonlítása (=, ≠, <, ≤, >, ≥) is.
- ***Logikai típus***. Típusérték-halmaza (jele: $\mathbb{L}$) a logikai értékeket tartalmazza, típusműveletei a logikai „és” (∧), logikai „vagy” (∨), a logikai „tagadás” (¬) és logikai „egyenlőség” (=) illetve „nem-egyenlőség” (≠).
- ***Karakter típus***. Típusérték-halmaza (jele: $\mathbb{K}$) a karaktereket (a nagy és kisbetűk, számjegyek, írásjelek, stb.) tartalmazza, típusműveletei a két karakter összehasonlításai (=, ≠, <, ≤, >, ≥).
- ***Halmaz típus***. Típusértékei olyan véges elemszámú halmazok, amelyek egy meghatározott alaphalmaz részei. *E* alaphalmaz esetén a jele: $\boldsymbol{2}^E$. Típusműveletei a szokásos elemi halmazműveletek: halmaz ürességének vizsgálata, elem kiválasztása halmazból, elem kivétele halmazból, elem berakása a halmazba.
- ***Sorozat típus***. Típusértékei olyan véges hosszúságú sorozatok, amelyek egy meghatározott alaphalmaz elemeiből állnak. *E* alaphalmaz esetén a jele: E^*. Típusműveletei: egy sorozat hosszának (elemszámának) lekérdezése ($|s|$); sorozat adott sorszámú elemére történő hivatkozás (s_i), azaz egy elem kiolvasása illetve megváltozatása; sorozatba új elem beillesztése; adott elem kitörlése.
- ***Vektor típus***. (Egydimenziós tömb) Típusértékei olyan véges, azonos hosszúságú sorozatok, más néven vektorok, amelyek egy meghatározott alaphalmaz elemeiből állnak. A sorozatokat *m*-től *n*-ig terjedő egész számokkal számozzuk meg, más néven indexeljük. Az ilyen vektorok halmazát *E* alaphalmaz esetén az $E^{m..n}$ jelöli, $m=1$ esetén egyszerűen E^n. Típusművelete egy adott indexű elemre történő hivatkozás, azaz az elem kiolvasása illetve megváltoztatása. A vektor első és utolsó eleme indexének (az *m* és az *n* értékének) lekérdezése is lényegében egy-egy típusművelet.
- ***Mátrix típus***. (Kétdimenziós tömb) azaz Vektorokat tartalmazó vektor, amely speciális esetben azonos módon indexelt $E^{l..m}$-beli vektoroknak (úgynevezett soroknak) a vektora $(E^{l..m})^{k..n}$ vagy másképp jelölve $E^{l..m \times k..n}$, ahol *E* az elemi értékek halmazát jelöli), és amelyet $k=l=1$ esetén egyszerűen $E^{m \times n}$-ként is írhatunk. Típusművelete egy adott sor adott elemére (oszlopra) történő hivatkozás, azaz az elem kiolvasása, illetve megváltoztatása. Típusművelet

még a mátrix egy teljes sorára történő hivatkozás, valamint a sorokat tartalmazó vektor indextartományának, illetve az egyes sorok indextartományának lekérdezése.

1.2. Állapottér

Amikor egy feladat minden lényeges adata felvesz egy-egy értéket, akkor ezt az érték-együttest a feladat egy állapotának nevezzük.

A „macskás” feladat egy állapota például az, amikor a magasság értéke *12.8*, a sebességé *16*. Vezessünk be két címkét: a *h*-t a magasság, a *v*-t a sebesség jelölésére, és ekkor az előző állapotot a rövidített (*h*:*12.8, v*:*16*) formában is írhatjuk. Az itt bevezetett címkékre azért van szükség, hogy meg tudjuk különböztetni egy állapot azonos típusú értékeit. A (*12.8, 16*) jelölés ugyanis nem lenne egyértelmű: nem tudnánk, hogy melyik érték a magasság, melyik a sebesség. Ugyanennek a feladatnak állapota még például a (*h*:*5, v*:*10*) vagy a (*h*:*0, v*:*10*). Sőt – mivel az adatok között kezdetben semmiféle összefüggést nem tételezünk fel, csak a típusérték-halmazaikat ismerjük – a (*h*:*0, v*:*10.68*)-t és a (*h*:*25, v*:*0*)-t is állapotoknak tekintjük, noha ez utóbbiak a feladat szempontjából értelmetlenek, fizikai értelemben nem megvalósuló állapotok.

Az összes lehetséges állapot alkotja az állapotteret. Az állapotteret megadhatjuk úgy, hogy felsoroljuk a komponenseit, azaz az állapottér lényeges adatainak típusérték-halmazait egyedi címkékkel ellátva. Az állapottér címkéit a továbbiakban az állapottér változóinak hívjuk. Mivel a változó egyértelműen azonosítja az állapottér egy komponensét, ezért alkalmas arra is, hogy egy konkrét állapotban megmutassa az állapot adott komponensének értékét.

A „macskás” példában az állapottér a (*h*:$\mathbb{R}_0^+$, *v*:$\mathbb{R}_0^+$). Tekintsük ennek a (*h*:*12.8, v*:*16*) állapotát. Ha megkérdezzük, hogy mi itt a *h* értéke, akkor világos, hogy ez a *12.8*. Ha előírjuk, hogy változzon meg az állapot *v* komponense *31*-re, akkor a (*h*:*12.8, v*:*31*) állapotot kapjuk. Egy állapot komponenseinek értékét tehát a megfelelő változó nevének segítségével kérdezhetjük le vagy változtathatjuk meg. Ez utóbbi esetben egy másik állapotot kapunk.

Állapottere nemcsak egy feladatnak lehet. Minden olyan esetben bevezethetjük ezt a fogalmat, ahol adatokkal, pontosabban adattípusokkal van dolgunk, így például egy matematikai kifejezés vagy egy program esetében is.

1.3. Feladat fogalma

Amikor a „macskás” feladatot meg akarjuk oldani, akkor a feladatnak egy olyan állapotából indulunk ki, ahol a magasság ismert, például *12.8*, a sebesség pedig ismeretlen. Ilyen ***kezdőállapot*** több is van az állapottérben, ezeket (*h*:*12.8, v*:$*$)-gal jelölhetjük, ami azt fejezi ki, hogy a második komponens definiálatlan, tehát tetszőleges.

A feladatot akkor oldottuk meg, ha megtaláltuk azt a ***célállapotot***, amelyiknek sebességkomponense a *16*. Egy fizikus számára a logikus célállapot a (*h*:*0, v*:*16*) lenne, de a feladat megoldásához nem kell a valódi fizikai célállapot megtalálnunk. Az informatikus

számára a magasság-komponens egy bemenő adat, amelyre előírhatjuk például, hogy őrizze meg a kezdőértéket a célállapotban. Ebben az esetben a (*h*:*12.8*, *v*:*16*) tekinthető célállapotnak. Ha ilyen előírás nincs, akkor minden (*h*: $*$, *v*:*16*) alakú állapot célállapotnak tekinthető. Állapodjunk meg ennél a példánál abban, hogy megőrizzük a magasság-komponens értékét, tehát csak egy célállapotot fogadunk el: (*h*:*12.8*, *v*:*16*).

Más magasság-értékű kezdőállapotokhoz természetesen más célállapot tartozik. (Megjegyezzük, hogy ha az állapottér felírásánál nem zártuk volna ki a negatív valós számokat, akkor most ki kellene kötni, hogy a feladatnak nincs értelme olyan kezdőállapotokon, amelyek magasság-komponense negatív, például (*h*:*-1*, *v*: $*$), hiszen ekkor a becsapódási sebességet kiszámoló képletben negatív számnak kellene valós négyzetgyökét kiszámolni.). Általánosan leírva az értelmes kezdő- és célállapot kapcsolatokat, a következőt mondhatjuk. A feladat egy (*h*:h_0, *v*: $*$) kezdőállapothoz, ahol h_0 egy tetszőlegesen rögzített (nem-negatív valós) szám, a $*$ pedig egy tetszőleges nem-definiált érték, a (*h*:h_0, *v*:$\sqrt{20*h_0}$) célállapotot rendeli.

Tekintsünk most egy másik feladatot! „*Adjuk meg egy összetett szám egyik valódi osztóját!*” Ennek a feladatnak két lényeges adata van: az adott természetes szám (címkéje legyen *n*) és a válaszként megadandó osztó (címkéje: *d*). Mindkét adat természetes szám típusú. A feladat állapottere tehát: (*n*:ℕ, *d*:ℕ). Rögzítsük az állapottér komponenseinek sorrendjét: megállapodás szerint az állapotok jelölésénél elsőként mindig az *n*, másodikként a *d* változó értékét adjuk meg, így a továbbiakban használhatjuk az (*n*:*6*, *d*:*1*) állapot-jelölés helyett a rövidebb (*6*,*1*) alakot. A feladat kezdőállapotai között találjuk például a (*6*,$*$) alakúakat. Ezekhez több célállapot is tartozik: (*6*,*2*) és (*6*,*3*), hiszen a *6*-nak több valódi osztója is van. Nincs értelme a feladatnak viszont a (*3*,$*$) alakú kezdőállapotokban, hiszen a *3* nem összetett szám, azaz nincs valódi osztója. A feladat általában az (*x*,$*$) kezdőállapothoz, ahol *x* egy összetett természetes szám, azokat az (*x*,*k*) állapotokat rendeli, ahol *k* az *x* egy valódi osztója.

A feladat egy olyan kapcsolat (leképezés), amelyik bizonyos állapotokhoz (kezdőállapotokhoz) állapotokat (célállapotokat) rendel. Ez a leképezés általában nem egyértelmű, más szóval *nem-determinisztikus*, hiszen egy kezdőállapothoz több célállapot is tartozhat. (Az ilyen leképezést a matematikában relációnak hívják.)

Érdekes tanulságokkal jár a következő feladat: „*Adjuk meg egy nem-nulla természetes szám legnagyobb valódi osztóját!*” Itt bemeneti adat az a természetes szám, aminek a legnagyobb valódi osztóját keressük. Tudjuk, hogy a kérdésre adott válasz is adatnak számít, de azt már nehéz észrevenni, hogy a válasz itt két részből áll. A feladat megfogalmazásából következik, hogy nemcsak egy összetett szám esetén várunk választ, hanem egy prímszám vagy az *1* esetén is. De ilyenkor nincs értelme legnagyobb valódi osztót keresni! A számszerű válaszon kívül tehát szükségünk van egy logikai értékű adatra, amely megmutatja, hogy van-e egyáltalán számszerű megoldása a problémának. A feladat állapottere ezért: (*n*:ℕ, *l*:𝕃, *d*:ℕ). A feladat ebben a megközelítésben bármelyik olyan állapotra értelmezhető, amelyiknek az *n* komponenshez

tartozó értéke nem nulla. Az olyan (x,$*$,$*$) állapotokhoz (itt is az *n,l,d* sorrendre épülő rövid alakot használjuk), ahol x nem összetett szám, a feladat az (x,*hamis*,$*$) alakú célállapotokat rendeli; ha x összetett, akkor az (x,*igaz*,k) célállapotot, ahol k a legnagyobb valódi osztója az x–nek.

A feladatok változóit megkülönböztethetjük aszerint, hogy azok a feladat megoldásához felhasználható kezdő értékeket tartalmazzák-e (bemenő- vagy input változók), vagy a feladat megoldásának eredményei (kimenő-, eredmény- vagy output változók). A bemenő változók sokszor megőrzik a kezdőértéküket a célállapotban is, de ez nem alapkövetelmény. A kimenő változók kezdőállapotbeli értéke közömbös, tetszőleges, célállapotbeli értékük megfogalmazása viszont a feladat lényegi része. Számos olyan feladat is van, ahol egy változó egyszerre lehet bemenő és kimenő is, sőt lehetnek olyan (segéd) változói is, amelyek csak a feltételek könnyebb megfogalmazásához nyújtanak segítséget.

1.4. Program fogalma

A szakkönyvek többsége a programot utasítások sorozataként definiálja. Nyilvánvaló azonban, hogy egy utasítássorozat csak a program leírására képes, és önmagában semmit sem árul el a program természetéről. Ehhez ugyanis egyfelől ismerni kellene azt a számítógépet, amely az utasításokat értelmezi, másfelől meg kellene adni, hogy az utasítások végrehajtásakor mi történik, azaz lényegében egy programozási nyelvet kellene definiálnunk. A program fogalmának ilyen bevezetése még akkor is hosszadalmas, ha programjainkat egy absztrakt számítógép számára egy absztrakt programozási nyelven írjuk le. A program fogalmának bevezetésére ezért mi nem ezt az utat követjük, hiszen könyvünk bevezetőjében éppen azt emeltük ki, hogy milyen fontos elválasztani a programtervezési ismereteket a számítógépeknek és a programozási nyelveknek a megismerésétől.

A fenti érvek ellenére időzzünk el egy pillanatra annál a meghatározásnál, hogy a program utasítások sorozata, és vegyünk szemügyre egy így megadott programnak a működését! (Ez a példa egy absztrakt gép és absztrakt nyelv ismeretét feltételezi, ám az utasítások megértése itt remélhetőleg nem okoz majd senkinek sem gondot.)

1. Legyen *d* értéke vagy az *n-1* vagy a *2*.
2. Ha *d* értéke megegyezik *2*-vel
   akkor
amíg *d* nem osztja *n*-t addig növeljük *d* értékét *1*-gyel,
   különben
amíg *d* nem osztja *n*-t addig csökkentsük *d* értékét *1*-gyel.

A program két természetes szám típusú adattal dolgozik: az elsőnek címkéje *n*, a másodiké *d*, azaz a program (adatainak) állapottere az (n:ℕ, d:ℕ). Vizsgáljuk meg, mi történik az

állapottérben, ha ezt a programot végrehajtjuk! Mindenekelőtt vegyük észre, hogy ***a program bármelyik állapotból elindulhat***.

Például a (*6,8*) *kiinduló állapot* esetén (itt is alkalmazzuk a rövidebb, az *n,d* sorrendjét kihasználó alakot az állapot jelölésére) a program első utasítása vagy a (*6,2*), vagy a (*6,5*) állapotba vezet el. Az első utasítás ugyanis nem egyértelmű, ettől a program működése nem-determinisztikus lesz. Természetesen egyszerre csak az egyik végrehajtás következhet be, de hogy melyik, az nem számítható ki előre. Ha az első utasítás a *d*-t *2*-re állítja be, akkor a második utasítás elkezdi növelni azt. Ellenkező esetben, *d=5* esetén, csökkenni kezd a *d* értéke addig, amíg az nem lesz osztója az *n*-nek. Ezért az egyik esetben – amikor *d* értéke *2* – rögtön meg is áll a program, a másik esetben pedig érintve a (*6,5*), (*6,4*) állapotokat a (*6,3*) állapotban fog terminálni. Bármelyik olyan kiinduló állapotból elindulva, ahol az első komponens *6*, az előzőekhez hasonló végrehajtási sorozatokat kapunk: a program a működése során egy (*6,*∗) alakú állapotból elindulva vagy a ⟨(*6,*∗) (*6,2*)⟩ állapotsorozatot, vagy a ⟨(*6,*∗) (*6,5*) (*6,4*), (*6,3*)⟩ állapotsorozatot futja be. Ezek a sorozatok csak a legelső állapotukban különböznek egymástól, hiszen ***egy végrehajtási sorozat első állapota mindig a kiinduló állapot***.

Induljunk el most egy (*5,*∗) alakú állapotból. Ekkor az első utasítás eredményétől függően vagy a ⟨(*5,*∗) (*5,2*) (*5,3*) (*5,4*) (*5,5*)⟩, vagy a ⟨(*5,*∗) (*5,4*) (*5,3*) (*5,2*) (*5,1*)⟩ végrehajtás következik be. A (*4,*∗) alakú állapotból is két végrehajtás indulhat, de mindkettő ugyanabban az állapotban áll meg: ⟨(*4,*∗) (*4,2*)⟩, ⟨(*4,*∗) (*4,3*) (*4,2*)⟩

Érdekes végrehajtások indulnak az (*1,*∗) alakú állapotokból. Az első utasítás hatására vagy az (*1,2*) , vagy az (*1,0*) állapotba kerülünk. Az első esetben a második utasítás elkezdi a *d* értékét növelni, és mivel az *1*-nek nincs *2*-nél nagyobb osztója, ez a folyamat soha nem áll le. Azt mondjuk, hogy a program a ⟨(*1,*∗) (*1,2*) (*1,3*) (*1,4*) … ⟩ végtelen végrehajtást futja be, mert végtelen ciklusba esik. A másik esetben viszont értelmetlenné válik a második utasítás, hiszen azt kell vizsgálni, hogy a nulla értékű *d* osztja-e az *n* értékét, de a nullával való osztás nem értelmezhető. Ez az utasítás tehát itt illegális, abnormális lépést eredményez. Ilyenkor a program „abnormálisan” működik tovább, amelyet az ⟨(*1,*∗) (*1,0*) (*1,0*) … ⟩ végtelen végrehajtási sorozattal modellezünk. Ez azt fejezi ki, mintha a program megpróbálná újra és újra végrehajtani az illegális lépést, majd mindig visszatér az utolsó legális állapothoz. Hasonló a helyzet a (*0,*∗) kiinduló állapotokkal is, amelyekre az első utasítás megkísérli a *d* értékét (ennek tetszőleges értékék rögzítsük és jelöljük *y*-nal) *-1*-re állítani. A (*0,-1*) ugyanis nem tartozik bele az állapottérbe, így az első utasítás a program abnormális működését eredményezheti, amelyet a ⟨(*0,y*) (*0,y*) (*0,y*) … ⟩ végtelen sorozattal jellemezhetünk. Mintha újra és újra megpróbálnánk a hibás utasítás végrehajtását, de a (*0,-1*) illegális állapot helyett mindig visszatérünk a (*0,y*) kiinduló állapothoz.. Természetesen itt is lehetséges egy másik végrehajtás, amikor az első lépésben a (*0,2*) állapotba kerülünk, ahol a program rögtön le is áll, hiszen a *2* osztja a nullát.

A program által egy állapothoz rendelt állapotsorozatot végrehajtásnak, működésnek, a program futásának hívunk. A ***végrehajtások lehetnek végesek vagy végtelenek***; a végtelen

működések lehetnek normálisak (***végtelen ciklus***) vagy abnormálisak (***abortálás***). Ha a végrehajtás véges, akkor azt mondjuk, hogy a program végrehajtása megáll, azaz ***terminál***. A normális végtelen működést nem lehet biztonsággal felismerni, csak abból lehet sejteni, ha a program már régóta fut. Az abnormális működés akkor következik be, amikor a végrehajtás során egy értelmezhetetlen utasítást (például nullával való osztás) kellene végrehajtani. Ezt a programunkat futtató környezet (operációs rendszer) fel szokta ismerni, és erőszakkal leállítja a működést. Innen származik az abortálás kifejezés. A modellünkben azonban nincs futtató környezet, ezért az abnormális működést olyan végtelen állapotsorozattal ábrázoljuk, amelyik az értelmetlen utasítás végrehajtásakor fennálló állapotot ismétli meg végtelen sokszor. A későbbiekben bennünket egy programnak csak a véges működései fognak érdekelni, és mind a normális, mind az abnormális végtelen végrehajtásokat megpróbáljuk majd elkerülni.

Egy program minden állapotból indít végrehajtást, sőt egy állapotból akár több különbözőt is. Az, hogy ezek közül éppen melyik végrehajtás következik be, nem definiált, azaz ***a program nem-determinisztikus***.

Vegyünk szemügyre egy másik programot! Legyen ennek az állapottere az (a:ℕ, b:ℕ, c:ℕ). Az állapotok leírásához rögzítsük a továbbiakban ezt a sorrendet.

1. Vezessünk be egy új változót (i:ℕ) és legyen az értéke *1*.
2. Legyen *c* értéke kezdetben *0*.
3. Amíg az *i* értéke nem haladja meg az *a* értékét addig
   adjuk a *c* értékéhez az *b* értékét, majd
   növeljük meg *i* értékét *1*-gyel.

Ez a program például a (*2*,*3*,*2009*) állapotból indulva az alábbi állapotsorozatot futja be: ⟨(*2*,*3*,*2009*), (*2*,*3*,*2009*,*1*), (*2*,*3*,*0*,*1*), (*2*,*3*,*3*,*1*), (*2*,*3*,*3*,*2*), (*2*,*3*,*6*,*2*), (*2*,*3*,*6*,*3*), (*2*,*3*,*6*)⟩. A végrehajtási sorozat első lépése kiegészíti a kiinduló állapotot egy új komponenssel és kezdőértéket is ad neki. Megállapodás szerint az ilyen futás közben felvett új komponensek a program befejeződésekor megszűnnek: ezért jelenik meg a fenti végrehajtási sorozat legvégén egy speciális lépés.

A program állapottere tehát a végrehajtás során dinamikusan változik, mert a végrehajtásnak vannak olyan lépései, amikor egy állapotból attól eltérő komponens számú (eltérő dimenziójú) állapotba kerülünk. Az új állapot lehet bővebb, amikor a megelőző állapothoz képest extra komponensek jelennek meg (lásd a fenti programban az i:ℕ megjelenését előidéző lépést), de lehet szűkebb is, amikor elhagyunk korábban létrehozott komponenseket (erre példa a fenti végrehajtás utolsó lépése). Megállapodunk abban, hogy a végrehajtás kezdetén létező komponensek, azaz a kiinduló állapot komponensei végig megmaradnak, nem szűnhetnek meg.

A program kiinduló- és végállapotainak közös terét a program ***alap-állapotterének*** nevezzük, változói az ***alapváltozók***. A működés során megjelenő újabb komponenseket ***segédváltozóknak*** fogjuk hívni. A program tartalmazhat segédváltozót megszüntető speciális lépéseket is, de az utolsó lépés minden segédváltozót automatikusan megszüntet. A segédváltozó megjelenésekor annak értéke még nem feltétlenül definiált.

A program az általa befutható összes lehetséges végrehajtás együttese. Rendelkezik egy alap-állapottérrel, amelyiknek bármelyik állapotából el tud indulni, és ahhoz olyan véges vagy végtelen végrehajtási sorozatokat rendel, amelyik első állapota a kiinduló állapot. Ugyanahhoz a kiinduló állapothoz akár több végrehajtási sorozat is tartozhat. Egy végrehajtási sorozat további állapotai az alap-állapottér komponensein kívül segéd komponenseket is tartalmazhatnak, de ezek a véges hosszú végrehajtások esetén legkésőbb az utolsó lépésében megszűnnek, így a véges végrehajtások az alap-állapottérben terminálnak.

Egy program működése nem változik meg attól, ha módosítjuk az alap-állapoterét. Egy alapváltozó ugyanis egyszerűen átminősíthető segédváltozóvá (ezt nevezzük az alap-állapottér leszűkítésének), és ekkor csak a program elindulásakor jön létre, a program befejeződésekor pedig megszűnik. Fordítva, egy segédváltozóból is lehet alapváltozó (ez az alap-állapottér kiterjesztése), ha azt már eleve létezőnek tekintjük, ezért nem kell létrehozni és nem kell megszüntetni. A program alap-állapotterét megállapodás alapján jelöljük ki. Ebbe nemcsak a program által ténylegesen használt, a program utasításaiban előforduló változók szerepelhetnek, hanem egyéb változók is. Az ilyen változóknak a kezdő értéke tetszőleges, és azt végig megőrzik a program végrehajtása során. Azt is könnyű belátni, hogy az alap-állapottér változóinak neve is lecserélhető anélkül, hogy ettől a program működése megváltozna.

1.5. Megoldás fogalma

Mivel mind a feladat fogalmát, mind a program fogalmát az állapottér segítségével fogalmaztuk meg, ez lehetővé teszi, hogy egyszerű meghatározást adjunk arra, mikor old meg egy program egy feladatot.

Tekintsük újra azt a feladatot, amelyben egy összetett nem-nulla természetes szám egyik valódi osztóját keressük. A feladat állapottere: (n:$\mathbb{N}$, d:$\mathbb{N}$), és egy (x,$*$) kezdőállapothoz (ahol x pozitív és összetett) azokat az (x,y) célállapotokat rendeli, ahol a y értéke osztja az x-et, de nem *1* és nem x. Az előző alfejezet első programjának állapottere megegyezik e feladatéval. A feladat bármelyik kezdő állapotából is indítjuk el a program mindig terminál, és olyan állapotban áll meg, ahol a d értéke vagy a legkisebb, vagy a legnagyobb valódi osztója az n kezdőértékének. Például a program a (*12*,*8*) kiinduló állapotból elindulva nem-determinisztikus módon vagy a (*12*,*2*), vagy a (*12*,*6*) végállapotban fog megállni, miközben a feladat a (*12*,*8*) kezdőállapothoz a (*12*,*2*), (*12*,*3*), (*12*,*4*), (*12*,*6*) célállapotokat jelöli ki. A program egy végrehajtása tehát ezek közül talál meg mindig egyet, azaz megoldja a feladatot.

Egy program akkor old meg egy feladatot, ha a feladat bármelyik kezdőállapotából elindulva biztosan terminál, és olyan állapotban áll meg, amelyet célállapotként a feladat az adott kezdőállapothoz rendel.[3] Ahhoz, hogy a program megoldjon egy feladatot, nem szükséges, hogy egy adott kezdőállapothoz tartozó összes célállapotot megtalálja, elég csak az egyiket.

A program nem-determinisztikussága azt jelenti, hogy ugyanazon állapotból indulva egyszer ilyen, másszor olyan végrehajtást futhat be. A megoldáshoz az kell, hogy bármelyik végrehajtás is következik be, az megfelelő célállapotban álljon meg. Nem oldja meg tehát a program a feladatot akkor, ha a feladat egy kezdőállapotából indít ugyan egy célállapotba vezető működést, de emellett egy másik, nem célállapotban megálló vagy egyáltalán nem terminálló végrehajtást is produkál.

A megoldás vizsgálata szempontjából közömbös, hogy azokból az állapotokból indulva, amelyek nem kezdőállapotai a feladatnak, hogyan működik a program. Ilyenkor még az sem számít, hogy megáll-e egyáltalán a program. Ezért nem baj, hogy a $(0,*)$ és az $(1,*)$ alakú állapotokból indulva a vizsgált program nem is biztos, hogy terminál. Az is közömbös, hogy egy olyan $(x,*)$ állapotból indulva, ahol x egy prímszám, a program vagy az x-et, vagy az 1-et „találja meg”, azaz nem talál valódi osztót.

A megoldás tényét az sem befolyásolja, hogy a program a működése közben mit csinál, mely állapotokat érinti, milyen segédváltozókat vezet be; csak az számít, hogy honnan indulva hová érkezik meg, azaz a feladat kezdőállapotaiból indulva megáll-e, és hol. A $(4,*)$ alakú állapotokból két különböző végrehajtás is indul, de mindkettő a $(4,2)$ állapotban áll meg, hiszen a 2 a 4-nek egyszerre a legkisebb és legnagyobb valódi osztója. A $(6,*)$ alakú állapotokból két olyan végrehajtás indul, amelyeknek a végpontja is különbözik, de mindkettő a $(6,*)$-hoz tartozó célállapot.

A programnak azt a tulajdonságát, amely azt mutatja meg, hogy mely állapotokból indulva fog biztosan terminálni, és ezen állapotokból indulva mely állapotokban áll meg, a ***program hatásának*** nevezzük. Egy program hatása figyelmen kívül hagyja azokat az állapotokat és az abból induló összes végrehajtást, amely állapotokból végtelen hosszú végrehajtás is indul, a többi végrehajtásnak pedig nem mutatja meg a „belsejét”, csak a végrehajtás kiinduló és végállapotát. Egy program hatása természetesen attól függően változik, hogy mit választunk a program alap-állapotterének. ***Ekvivalensnek*** akkor mondunk két programot, ha bármelyik közös alap-állapottéren megegyezik a hatásuk.

Ahhoz, hogy egy feladatot megoldjunk egy programmal, a program alapváltozóinak a feladat (bemenő és eredmény) változóit kell választanunk. A program ugyanis csak az alapváltozóin keresztül tud a környezetével kapcsolatot teremteni, csak egy alapváltozónak tudunk „kívülről” kezdőértéket adni, és egy alapváltozóból tudjuk a terminálás után az

[3] Ismert a megoldásnak ennél gyengébb meghatározása is, amelyik nem követeli meg a leállást, csak azt, hogy ha a program leáll, akkor egy megfelelő célállapotban tegye azt.

eredményt lekérdezni. Ezért a megoldás definíciójában feltételeztük, hogy a feladat állapottere megegyezik a program alap-állapotterével. A programozási gyakorlatban azonban sokszor előfordul, hogy egy feladatot egy olyan programmal akarunk megoldani, amelyik állapottere nem azonos a feladatéval, bővebb vagy szűkebb annál. A megoldás fogalmát ezekben az esetekben úgy ellenőrizzük, hogy először a program alap-állapotterét a feladatéhoz igazítjuk, vagy kiterjesztjük, vagy leszűkítjük (esetleg egyszerre mindkettőt) a feladat állapotterére.

Abban az esetben, amikor a program alap-állapottere bővebb, mint feladaté, azaz van a programnak olyan alapváltozója, amely nem szerepel a feladat állapotterében (ez a változó nem kommunikál a feladattal: nem kap tőle kezdő értéket, nem ad neki eredményt), akkor ezt a változót a program segédváltozójává minősítjük. Jó példa erre az, amikor rendelkezünk egy programmal, amely egy napon keresztül óránként mért hőmérsékleti adatokból ki tudja számolni a napi átlaghőmérsékletet és a legnagyobb hőingadozást, ugyanakkor a megoldandó feladat csak az átlaghőmérséklet kiszámolását kéri. A program alap-állapottere tehát olyan komponenst tartalmaz, amely a feladatban nem szerepel (ez a legnagyobb hőingadozás), ez a feladat szempontjából érdektelen. A program első lépésében hozzáveszi a feladat egy kezdőállapotához a legnagyobb hőingadozást, ezután a program ugyanúgy számol vele, mint eredetileg, de a megállás előtt elhagyja ezt a komponenst, hiszen a feladat célállapotában ennek nincs helye; a legnagyobb hőingadozás tehát segédváltozó lett.

A másik esetben – amikor a feladat állapottere bővebb a program alap-állapotterénél – a program állapotterét kiegészítjük a feladat állapotterének azon komponenseivel, amelyek a program állapotterében nem szerepelnek. Ha egy ilyen új komponens segédváltozóként szerepelt az eredeti programban, akkor ez a kiegészítés a segédváltozót a program alapváltozójává emeli. Ha az új változó segédváltozóként sem szerepelt eddig a programban, akkor ez a változó a program által nem használt új alapváltozó lesz, azaz kezdeti értéke végig megőrződik egy végrehajtás során. De vajon van-e olyan feladat, amelyet így meg lehet oldani? Talán meglepő, de van. Ilyen az, amikor egy összetett feladat olyan részének a megoldásával foglalkozunk, ahol az összetett feladat bizonyos komponenseit ideiglenesen figyelmen kívül hagyhatjuk, de attól még azok a részfeladat állapotterének komponensei maradnak, és szeretnénk, ha az értékük nem változna meg a részfeladatot megoldó program működése során.

Az itt bevezetett megoldás definíció egy logikus, a gyakorlatot jól modellező fogalom. Egyetlen probléma van vele: a gyakorlatban történő alkalmazása szinte lehetetlen. Nehezen képzelhető ugyanis az el, hogy egy konkrét feladat és program ismeretében (miután a program alap-állapotterét a feladat állapotteréhez igazítottuk), egyenként megvizsgáljuk a feladat összes kezdőállapotát, hogy az abból indított program általi végrehajtások megfelelő célállapotban állnak-e meg. Ezért a következő két fejezetben bevezetünk mind a feladatok, mind a programok leírására egy-egy sajátos eszközt, majd megmutatjuk, hogy ezek segítségével hogyan lehet igazolni azt, hogy egy program megold egy feladatot.

2. Specifikáció

Ebben a fejezetben azzal foglalkozunk, hogy hogyan lehet egy feladat kezdőállapotait a megoldás ellenőrzésének szempontjából csoportosítani, halmazokba rendezni úgy, hogy elég legyen egy-egy ilyen halmaznak egy tetszőleges elemére ellenőrizni azt, hogy onnan indulva az adott program megfelelő helyen áll-e meg. Ehhez a feladatnak egy sajátos formájú megfogalmazására, a feladat specifikációjára lesz szükség.

2.1. Kezdőállapotok osztályozása

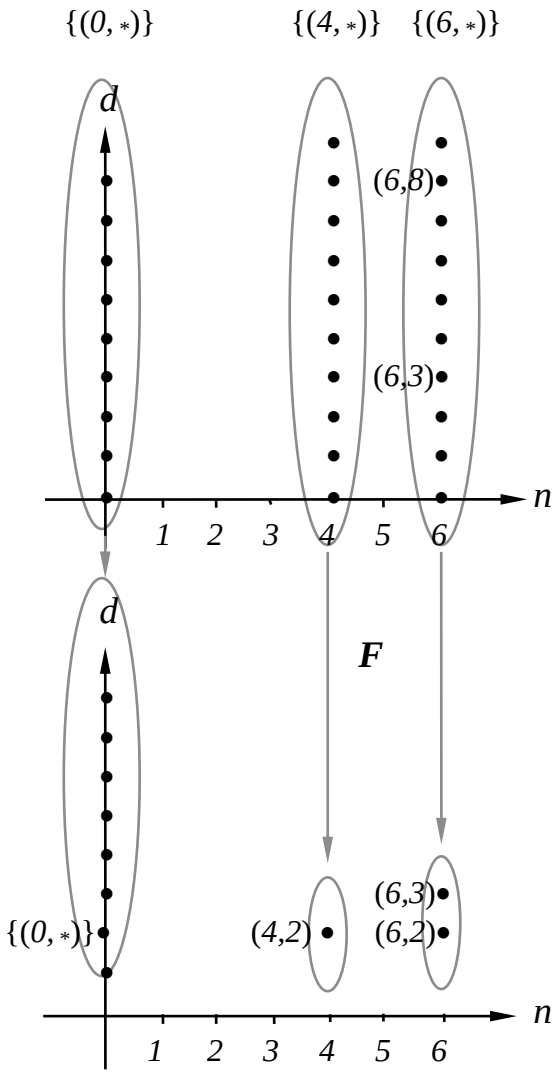
A megoldás szempontjából egy feladat egy kezdőállapotának legfontosabb tulajdonsága az, hogy milyen célállapotok tartoznak hozzá. Megfigyelhető, hogy egy feladat különböző kezdőállapotokhoz sokszor ugyanazokat a célállapotokat rendeli. Kézenfekvőnek látszik, hogy ha egy csoportba vennénk azokat a kezdőállapotokat, amelyekhez ugyanazok a célállapotok tartoznak, akkor a megoldás ellenőrzését nem kellene a kezdőállapotokon külön-külön elvégezni, hanem elég lenne az így keletkező csoportokat egyben megvizsgálni: azt kell belátni, hogy a feladat kezdőállapotainak egy-egy ilyen csoportjából elindulva a program megfelelő célállapotban áll-e meg.

Tekintsünk példaként egy már korábban szereplő feladatot: „*Egy adott összetett számnak keressük egy valódi osztóját*”! A feladat állapottere (n:ℕ, d:ℕ). Az állapotok egyszerűbb jelölése érdekében rögzítsük a fenti sorrendet a komponensek között: egy állapot első komponense a megadott szám, a második a valódi osztó. Kössük ki, hogy az első komponens kezdőértéke a célállapotokban is megmaradjon.

Az *2.1. ábrán* egy derékszögű koordinátarendszer I. negyedével ábrázoljuk ezt az állapotteret, hiszen ebben minden egész koordinátájú rácspont egy-egy állapotot szimbolizál. (A feladat csak azokban az állapotokban értelmezett, ahol az első komponens összetett szám.) Külön koordinátarendszerbe rajzoltuk be a feladat néhány kezdőállapotát, és külön egy másikban az ezekhez hozzárendelt célállapotokat. Az ábra jól tükrözi vissza azt, hogy a feladat állapotokhoz állapotokat rendelő leképezés.

A feladat például a (*6,8*) kezdőállapothoz a (*6,3*) és a (*6,2*) célállapotokat rendeli. De ugyanezek a célállapotok tartoznak a (*6,6*), a (*6,3*) vagy a (*6,127*) kezdőállapotokhoz is, mivel ezek első komponensében ugyancsak a *6* áll. Tehát az összes (*6,*$*$) alakú állapot (a $*$ itt egy tetszőleges értéket jelöl) ugyanazon csoportba (halmazba) sorolható az alapján, hogy hozzájuk egyformán a (*6,3*) és a (*6,2*) célállapotokat rendeli a feladat. Úgyis fogalmazhatunk, hogy a feladat a {(*6,*$*$)} halmaz elemeihez a {(*6,3*), (*6,2*)} halmaz elemeit rendeli. Ezen megfontolás alapján külön csoportot képeznek a (*0,*$*$), a (*4,*$*$), a (*6,*$*$) alakú állapotok is, általánosan fogalmazva azok az (*n',*$*$) alakú kezdőállapotok, ahol az *n'* összetett szám. Ezt láthatjuk *2. ábra* felső részén. Nem kerül viszont bele egyetlen ilyen halmazba sem például az (*1,1*) vagy a (*3,5*)

állapot, hiszen ezek nem kezdőállapotai a feladatnak. (Könnyű megmutatni, hogy ez a csoportosítás matematikai értelemben egy osztályozás: minden kezdőállapot pontosan egy halmazhoz (csoporthoz) tartozik.)



2.1. Ábra
Egy feladat: Keressük egy összetett szám valódi osztóit.

Amikor a feladathoz megoldó programot keresünk, azt kell belátnunk, hogy minden ilyen halmaz egyik (tetszőleges) eleméből (ez tehát a feladat egy kezdőállapota) indulva a program a

feladat által kijelölt valamelyik célállapotban áll meg. Ha ügyesek vagyunk, akkor elég ezt a vizsgálatot egyetlen jól megválasztott halmazra elvégezni, mondjuk az általános (*n'*,∗) alakú kezdőállapotok halmazára, ahol az *n'* összetett szám.

Általában egy feladat kezdőállapotainak a fent vázolt halmazait nem könnyű előállítani. Ennek illusztrálására tekintsük azt a példát (lásd *1.1. feladat*), amikor *egy egyenes vonalú egyenletes mozgást végző testnek kell az átlagsebességét kiszámolni* a megtett út és az eltelt idő függvényében. Ennek a feladatnak az állapottere (*s*:ℝ, *t*:ℝ, *v*:ℝ), ahol a komponensek rendre az út (*s*), az idő (*t*), és a sebesség (*v*). (Rögzítjük ezt a sorrendet a komponensek között.) A kezdőállapotokban az első komponens nem lehet negatív, a második komponensnek pedig pozitívnak kell lennie.

Melyek azok a kezdőállapotok, amelyekhez ez a feladat a (∗,∗,*50*) alakú célállapotokat rendeli? Nem is olyan könnyű erre a válasz. A (*100*,*2*,∗) alakú kezdőállapotok ilyenek, a (*200*,*4*,∗) alakúak is. Ismerve a feladat megoldásához használt *v=s/t* képletet (ezt a képletet csak az ilyen egyszerű feladatok esetében látjuk ilyen világosan), kis gondolkodás után kitalálhatjuk, hogy az (*50a*,*a*,∗) alakú kezdőállapotokhoz (ahol *a* egy tetszőleges pozitív valós szám), és csak azokhoz rendeli a feladat a (∗,∗,*50*) alakú célállapotokat.

De nem lehetne-e ennél egyszerűbben, az eredményt kiszámoló képlet alkalmazása nélkül csoportosítani a kezdőállapotokat? Miért ne tekinthetnénk külön csoportnak a (*100*,*2*,∗) kezdőállapotokat, és külön a (*200*,*4*,∗) kezdőállapotokat? Mindkettő halmazra teljesül, hogy a benne levő kezdőállapotokhoz ugyanazon célállapotok tartoznak. Igaz, hogy ezen célállapotok mindkét csoport esetében ugyanazok, de miért lenne ez baj? Ezek kijelöléséhez csak azt kell tudnunk, hogy a feladat első két adata a bemenő adat, az eredmény kiszámolásának képlete nem kell. Azonos bemenő adatokhoz ugyanis ugyanazon eredmény tartozik, tehát azonos bemenő adatokból felépített kezdőállapotokhoz a feladat ugyanazokat a célállapotokat rendeli. Az azonos bemenő adatokból felépített kezdőállapotok halmazai is osztályozzák a kezdőállapotokat, hiszen minden kezdőállapot pontosan egy ilyen halmazhoz tartozik. Ezért ha a feladat kezdőállapotainak minden ilyen halmazáról belátható, hogy belőle indulva a program megfelelő célállapotban áll-e meg, akkor igazoltuk, hogy a vizsgált program megoldja a feladatot.

Természetesen ilyenkor sem kell az összes halmazt megvizsgálni. Elég egy általános halmazzal foglalkozni. Rögzítsük a példánk bemenő adatainak értékeit az *s'*, *t'* tetszőlegesen kiválasztott nem negatív paraméterekkel, ahol *t'* nem lehet nulla. Egy vizsgált program akkor oldja meg a feladatot, ha az {(*s'*,*t'*,∗)} állapot-halmazból kiindulva a program az {(∗,∗, *s'/t'*)} állapot-halmaz valamelyik állapotában áll meg.

Néha találkozhatunk olyan feladattal is, amelynek nincsenek bemeneti adatai. Ilyen például az, amikor egy prímszámot keresünk. Ennek a feladatnak az állapottere a lehetséges válaszoknak az ℕ halmaza. Itt a válasz nem függ semmilyen bemeneti értéktől. Az összes állapot tehát egyben kezdőállapot is, amelyeket egyetlen halmazba foghatunk össze, hiszen mindegyikhez ugyanazon célállapotok, a prímszámok tartoznak.

2.2. Feladatok specifikációja

Vegyük elő újra azt a feladatot, amikor *az egyenes vonalú egyenletes mozgást végző testnek a megtett út és az eltelt idő függvényében kell az átlagsebességét kiszámolni*! A feladat lényeges adatai a megtett út, az eltelt idő és az átlagsebesség. Ennek megfelelően a feladat állapottere a változókkal: $A = (s{:}\mathbb{R}, t{:}\mathbb{R}, v{:}\mathbb{R})$. Bemenő adatai a megtett út (s) és az eltelt idő (t). Rögzítsünk a bemenő adatok számára egy-egy tetszőlegesen kiválasztott értéket! Legyen az út esetén ez az s', az idő esetén a t', ahol egyik sem lehet negatív, sőt a t' nulla sem. A kezdőállapotoknak az s', t' bemenő adatokkal rendelkező részhalmaza az $\{(s',t',*)\}$, az ezekhez tartozó célállapotok pedig a $\{(*,*, s'/t')\}$.

A kezdőállapotoknak és a célállapotoknak a fent vázolt halmazait logikai állítások (feltételek) segítségével is megadhatjuk. A logikai állítások minden esetben az állapottér állapotait minősítik: ha egy állapot kielégíti a logikai állítást, azaz a logikai állítás igaz értéket rendel hozzá, akkor az benne van a logikai állítás által jelzett halmazban. Például a $\{(100,2,*)\}$ halmazt az $s=100$ és $t=2$ logikai állítással, a $\{(200,4,*)\}$ halmazt az $s=200$ és $t=4$ állítással is jelölhetjük. Általában az $\{(s',t',*)\}$ halmazt az $s=s'$ és $t=t'$ állítás írja le, a $\{(*,*, s'/t')\}$ halmazt pedig a $v=s'/t'$. Ha ki szeretnénk kötni, hogy az út hossza nem negatív és az idő pozitív, akkor a fenti állítások mellé oda kell írnunk azt is, hogy $s'\geq 0$ és $t'>0$. A kezdőállapotokat leíró előfeltételt *Ef*-fel, a célállapotokat leíró utófeltételt *Uf*-fel fogjuk jelölni. Megfigyelhető, hogy mindkettő feltétel a változókra fogalmaz meg megszorításokat. Ha egy változót egy feltétel nem említ, az azt jelzi, hogy arra a változóra nézve nincs semmilyen megkötés, azaz annak az értéke tetszőleges lehet. Példánk előfeltételben ilyen a v, az utófeltételben az s és a t.

Tömören leírva az eddig elmondottakat a feladatnak az alábbi úgynevezett specifikációjához jutunk:

$A = (s{:}\mathbb{R}, t{:}\mathbb{R}, v{:}\mathbb{R})$
$Ef = (s=s' \text{ és } t=t' \text{ és } s'\geq 0 \text{ és } t'>0)$
$Uf = (v= s'/t')$

A feladat specifikálása során meghatározzuk a bemenő adatok tetszőlegesen rögzített értékeihez tartozó kezdőállapotok halmazát, valamint az ehhez tartozó célállapotok halmazát. A feladat specifikációja tartalmazza:

1. az állapotteret, azaz a feladat lényeges adatainak típusérték-halmazait az egyes adatokhoz tartozó változó nevekkel együtt;
2. az előfeltételt, amely a kezdőállapotok azon halmazát leíró logikai állítás, amely rögzíti a bemenő változók egy lehetséges, de tetszőleges kezdőértékét (ezeket általában a megfelelő változónév vesszős alakjával jelöljük);
3. az utófeltételt, amely a fenti kezdőállapotokhoz rendelt célállapotok halmazát megadó logikai állítás.

Meg lehet mutatni, hogy ha rendelkezünk egy feladatnak a specifikációjával és találunk olyan programot, amelyről belátható, hogy egy az előfeltételt kielégítő állapotból elindulva a program az utófeltételt kielégítő valamelyik állapotba kerül, akkor a program megoldja a feladatot. Ezt mondja ki a ***specifikáció tétele***.

Egy feladat állapotterében megjelenő változók minősíthetőek abból a szempontból, hogy bemenő illetve kimenő változók-e vagy sem. Ugyanaz a változó lehet egyszerre bemenő is és kimenő is. A bemenő változók azok, amelyeknek a feladat előfeltétele kezdőértéket rendel, amelyek explicit módon megjelennek az előfeltételben. Az utófeltételben explicit módon megjelenő változók közül azok a kimenő változók, amelyekre nincs kikötve, hogy értékük megegyezik az előfeltételben rögzített kezdőértékükkel.

* * *

Specifikáljuk most azt a feladatot, amikor *egy összetett szám legnagyobb valódi osztóját keressük*. A feladatnak két lényeges adata van: az adott természetes összetett szám és a keresett valódi osztó. Mindkét adat természetes szám típusú. Ezt a tényt rögzíti az állapottér.

$A = (n{:}\mathbb{N}, d{:}\mathbb{N})$

A két adat közül az első a bemeneti, a második a kimeneti adat. Legyen *n'* a tetszőlegesen kiválasztott bemenő érték. Mivel a feladat nem értelmes a *0*-ra, az *1*-re és a prímszámokra; ezért az *n'* csak összetett szám lehet. Az előfeltétel:

$Ef =$ ($n = n'$ *és* n' *összetett szám*)

Kiköthetjük, hogy a célállapot első komponensének, azaz a bemeneti adatnak értéke ne változzon meg, maradjon továbbra is *n'*; a második adat pedig legyen az *n'* legnagyobb valódi osztója:

$Uf =$ (*Ef és d valódi osztója n-nek és bármelyik olyan k szám esetén, amelyik nagyobb mint d, a k nem valódi osztója n-nek*)

Ha elemi matematikai jelöléssel akarjuk az állításainkat megfogalmazni, azt is megtehetjük. Az utófeltételben egy osztó "valódi" voltát egyszerűen kifejezhetjük a $d \in [2..n-1]$ állítással, hiszen az *n* szám valódi osztói csak itt helyezkedhetnek el. (Választhatnánk a $[2..n \text{ div } 2]$ intervallumot is, ahol az $n \text{ div } 2$ az *n* felének egészértéke.) Ha az intervallum egy eleme osztója az *n*-nek, akkor valódi osztója is. Alkalmazzuk a $d \mid n$ jelölést arra, hogy a *d* osztója az *n*-nek. Ennek megfelelően például azt az állítást, hogy a "*d* valódi osztója *n*-nek" úgy is írhatjuk, hogy "$d \in [2..n-1]$ és $d \mid n$". Aki otthonosan mozog az elsőrendű logika nyelvében, tömörebben is felírhatja a specifikáció állításait, ha használja a logikai műveleti jeleket. Ez természetesen nem változtat a specifikáció jelentésén. (Az előfeltételnél kihasználjuk azt, hogy egy szám összetettsége azt jelenti, hogy létezik a számnak valódi osztója.) Ennek megfelelően az elő- és az utófeltételt az alábbi formában is felírhatjuk:

$Ef = (n = n' \wedge \exists k \in [2..n-1]: k \mid n)$

$Uf = (Ef \wedge d \in [2..n-1] \wedge d \mid n \wedge \forall k \in [d+1..n-1] : \neg k \mid n)$

Az elő- és utófeltételben megjelenő *k* a specifikáció egy olyan eszköze (formula változója), amelyet állításaink precíz megfogalmazásához használunk. A $\exists k \in [a..b]$:tulajdonság(*k*) (van olyan, létezik olyan *k* egész szám az *a* és *b* között, amelyre igaz a tulajdonság(*k*)) formula azt az állítást írja le, hogy *a* és *b* egész számok közé eső egész számok egyike kielégíti az adott tulajdonságot. A $\forall k \in [a..b]$:tulajdonság(*k*) (bármelyik", "minden olyan" *k* egész szám az *a* és *b* között, amelyre igaz a tulajdonság(*k*)) formula azt jelenti, hogy *a* és *b* egész számok közé eső mindegyik egész szám kielégíti az adott tulajdonságot.

A formulák helyes értelmezéséhez rögzíteni kell a kifejezésekben használt műveleti jelek közötti precedencia sorrendet. Az általunk alkalmazott sorrend kissé eltér a matematikai logikában szokásostól, amennyiben az implikációt szorosabban kötő műveletnek tekintjük, mint a konjunkciót (logikai „és"-t). Ennek az oka az, hogy viszonylag sokszor fogunk olyan állításokat írni, ahol implikációs formulák lesznek „összeéselve", és zavaró lenne, ha kifejezéseinkben túl sok zárójelet kellene használni. Például a $d \mid n \wedge l \rightarrow d \in [2..n-1]$ kifejezést a matematikai logikában a $d \mid n \wedge (l \rightarrow d \in [2..n-1])$ formában kellene írni.. A logikai műveleti jelek precedencia sorrendje tehát: $\forall$, $\exists$, $\neg$, $\rightarrow$, $\wedge$, $\vee$. A könyvben használt jelöléseknél a kettőspont a kvantorok ($\forall$, $\exists$) hatásának leírásához tartozó jel. Ezért alkot egyetlen összetartozó kifejezést a $\exists k \in [2..n-1] : k \mid n$.

A kifejezések tartalmazhatnak a logikai műveleteken kívül egyéb műveleteket is. Ezek prioritása megelőzi a két-argumentumú logikai műveleti jelekét. Ilyen például a nem logikai értékű kifejezések egyenlőségét vizsgáló (=) operátor is, ezért írhatjuk az $x = y \wedge z = y$ kifejezést az $(x = y) \wedge (z = y)$ kifejezés helyett. Az egyéb, nem logikai műveletek között ez az egyenlőség operátor az utolsó a precedencia sorban, ezért senki ne olvassa például az $x+y = z$ kifejezést $x + (y = z)$ kifejezésnek. Nem fogjuk megkülönböztetni a nem logikai értékek egyenlőség operátorát a logikai értékek egyenlőség operátorától Erre a matematikai logikában az ekvivalencia ($\leftrightarrow$) műveleti jelet szokták használni. A szövegkörnyezetből mindig kiderül, hogy melyik egyenlőség operátorról van szó, legfeljebb zárójelezéssel tesszük majd a formuláinkat egyértelművé. Ez egyben azt is jelenti az egyenlőség operátor precedenciája jobb, mint az implikációjé.

Sokszor segít, ha a specifikáció felírásához bevezetünk néhány saját jelölést is. Bár ezeket külön definiálni kell, de használatukkal olvashatóbbá, tömörebbé válik a formális leírás. Ha például az *összetett*(*n*) azt jelenti, hogy az *n* természetes szám egy összetett szám (az *összetett*(*n*) pontosan akkor igaz, ha $\exists k \in [2..n-1]: k \mid n$), az *LVO(n)* pedig megadja egy összetett *n* természetes szám legnagyobb valódi osztóját, akkor az előző specifikáció az alábbi formában is leírható.

$Ef = (n = n' \wedge összetett(n))$
$Uf = (Ef \wedge d = LVO(n))$

Módosítsuk az előző feladatot az alábbira: *Keressük egy természetes szám legnagyobb valódi osztóját*. (Tehát nem biztos, hogy van az adott számnak egyáltalán valódi osztója.) Itt az állapottérbe az előző feladathoz képest egy logikai komponenst is fel kell vennünk, ami azt jelzi majd a célállapotban, hogy van-e egyáltalán legnagyobb valódi osztója a megadott természetes számnak. Az előfeltétel most nem tartalmaz semmilyen különleges megszorítást. Az utófeltétel egyrészt bővül az *l* változó értékének meghatározásával (*l* akkor igaz, ha *n* összetett), másrészt a *d* változó értékét csak abban az esetben kell specifikálnunk, ha az *l* értéke igaz. Ha összevetjük a specifikációt az előző feladat specifikációjával, akkor azt vehetjük észre, hogy az *n* összetettségét vizsgáló állítás ott az előfeltételben, itt az utófeltételben jelenik meg.

$A = (n{:}\mathbb{N}, l{:}\mathbb{L}, d{:}\mathbb{N})$
$Ef = (n = n')$
$Uf = (n = n' \wedge l = \exists k \in [2..n-1]: k \mid n$
$\quad \wedge l \rightarrow (d \in [2..n-1] \wedge d \mid n \wedge \forall k \in [d+1..n-1]: \neg k \mid n))$

A fenti specifikáció olvashatóbbá válik, ha az előző feladatnál bevezetett *összetett* és *LVO* jelöléseket használjuk:

$A = (n{:}\mathbb{N}, l{:}\mathbb{L}, d{:}\mathbb{N})$
$Ef = (n=n')$
$Uf = (n=n' \wedge l=összetett(n) \wedge l \rightarrow d=LVO(n))$

Ugyanazt a feladatot többféleképpen is lehet specifikálni. Minél összetettebb egy probléma, annál változatosabb lehet a leírása. A következő igen egyszerű példát háromféleképpen specifikáljuk: *növeljünk meg egy egész számot eggyel*. Az első két változatban külön komponens képviseli a bemenő (*a* változó) és külön a kimenő adatot (*b* változó). Az első változat nem tartalmaz semmi különöset a korábbi specifikációkhoz képest. A bemeneti adat megőrzi a kezdő értékét a célállapotban is, ezért a kimeneti változó értékének meghatározásánál mindegy, hogy a bemeneti változó vagy a paraméter változó értékére hivatkozunk. A második specifikációnál nem követeljük meg, hogy az *a* változó ne változzon meg, ezért a *b* változó értékének meghatározásánál az *a* változó kezdő értékét jelző *a'* paramétert kell használnunk. Egészen más felfogáson alapul a harmadik specifikáció. Ez úgy értelmezi a feladatot, hogy van egy adat, amelynek értékét „helyben” kell megnövelni eggyel. Ilyenkor az állapottér egykomponensű és az utófeltétel megfogalmazásánál elengedhetetlenül fontos, hogy a kezdőértéket rögzítő paraméter használata. Ez a specifikáció rávilágít arra is, hogy egy feladatban nem feltétlenül különülnek el a bemeneti és a kimeneti adatok.

$A = (a{:}\mathbb{Z}, b{:}\mathbb{Z})$	$A = (a{:}\mathbb{Z}, b{:}\mathbb{Z})$	$A = (a{:}\mathbb{Z})$
$Ef = (\ a{=}a'\ )$	$Ef = (\ a{=}a'\ )$	$Ef = (\ a{=}a'\ )$
$Uf = (\ a{=}a' \wedge b{=}a{+}1\ )$	$Uf = (\ b{=}a'{+}1\ )$	$Uf = (\ a{=}a'{+}1\ )$

Végül egy bemeneti adat nélküli feladat: Adjunk meg egy prímszámot!

$A = (p{:}\mathbb{Z})$

$Ef =(\ igaz\ )$

$Uf =(\ p\ egy\ prímszám\ ) = (\forall k{\in}[2..p{-}1] : \neg(k \mid p)\ )$

Ez a specifikáció az összes állapotot kezdőállapotnak tekinti, amelyek mindegyikéhez az összes prímszámot rendeli. Az előfeltétel egy azonosan igaz állítás, hiszen semmilyen korlátozást nem kell bevezetni a *p* változóra, az kezdetben tetszőleges értéket vehet fel, amelytől nem függ az eredmény. (Az utófeltételben a $[2..p{-}1]$ intervallum helyett írhatnánk a $[2..\sqrt{n}\,]$ intervallumot is.)

3. Strukturált programok

Ebben a fejezetben azt vizsgáljuk meg, hogy ha egy programot meglevő programokból építünk fel meghatározott szabályok alapján, akkor hogyan lehet eldönteni róluk, hogy megoldanak egy kitűzött feladatot. Ennek érdekében először meghatározzuk az elemi program fogalmát, definiálunk néhány nevezetes elemi programot, majd háromféle építési szabályt, úgynevezett programszerkezetet (szekvencia, elágazás, ciklus) mutatunk az összetett programok készítésére. Ezekkel a szabályokkal fokozatosan építhetünk fel egy programot: elemi programokból összetettet, az összetettekből még összetettebbeket. Az ilyen programokat sajátos, egymásba ágyazódó, más néven strukturált szerkezetük miatt ***strukturált programoknak*** hívják.

Strukturált programokat többféleképpen is le lehet írni. Mi egy sajátos, a program szerkezetét, struktúráját a középpontba állító, a fent említett három szerkezeten kívül más szerkezetet meg nem engedő absztrakt leírást, a ***struktogrammokat*** fogjuk erre a célra használni. Ez a programleírás egyszerű, könnyen elsajátítható, és természetes módon fordítható le ismert programozási nyelvekre, azaz bármelyik konkrét programozási környezetben jól használható; egyszerre illeszkedik az emberi gondolkodáshoz és a magas szintű programozási nyelvekhez.[4]

A struktogramm egy program leírására szolgáló eszköz, de felfogható a megoldandó feladat egyfajta leírásának is. Egy program készítésekor ugyanis a feladatot próbáljuk meg részfeladatokra bontani és meghatározni, hogy a részfeladatok megoldásait, hogyan, melyik programszerkezet segítségével kell egy programmá építeni. A programtervezés közbülső szakaszában tehát egy struktogramm nem elemi programokból, hanem megoldandó részfeladatokból építi fel a programot. Ez azért nem ellentmondás, mert elméletben minden feladat megoldható egy elemi programmal (lásd *3.3. alfejezet*), és ezért csak nézőpont kérdése, hogy egy részfeladatot feladatnak vagy programnak tekintünk-e.

A programkészítés során lépésről lépésre jelennek meg az egyre mélyebben beágyazott programszerkezetek, ami a feladat egyre finomabb részletezésének eredménye. Ezáltal a program kívülről befelé haladva, úgynevezett ***felülről-lefele*** (top-down) elemzéssel születik. A feladat

[4] A modellünkben bevezetett program fogalma annyira általános, hogy nem minden ilyen program írható le struktogramm segítségével. Ez nem a struktogramm-leírás hibája, ugyanis ezen programok más eszközökkel (folyamatábra, C++ nyelv vagy Turing gép) sem írhatók le. A Church-Turing tézis szerint csak a parciális rekurzív függvényeket kiszámító (megoldó) programok algoritmizálhatók. Ezek mind leírhatók folyamatábrákkal; a Bhöm-Jaccopini tétele szerint pedig, bármelyik folyamatábrával megadott program struktogrammal is leírható. A struktogramm tehát azon túl, hogy a programokat egy jól áttekinthető szerkezetet segítségével adja meg, egy kellően általános eszköz is: bármely algoritmizálható program leírható vele.

finomításának végén a struktogrammban legbelül levő, szerkezet nélküli egységeknek már magától értetődően megoldható részfeladatokat kell képviselniük.

3.1. Elemi programok

Elemi programoknak azokat a programokat nevezzük, amelyek végrehajtásai nagyon egyszerűek; működésüket egy vagy kettő hosszú azonos állapotterű állapotsorozatok vagy a kiinduló állapotot végtelen sokszor ismétlő (abnormálisan) végtelen sorozatok jellemzik.

3.1.1. Üres program

Az üres program a legegyszerűbb elemi program. Ez az úgynevezett „semmit sem csináló” program ***bármelyik állapotból indul is el, abban az állapotban marad***; végrehajtásai a kiinduló állapotból álló egyelemű sorozatok. Az üres programot a továbbiakban *SKIP*-pel jelöljük. Nyilvánvaló, ahhoz, hogy semmit sem csinálva eljussunk egy állapotba, már eleve abban az állapotban kell lennünk.

A gyakorlatban kitűzött feladatok azonban jóval bonyolultabbak annál, hogy azokat üres programmal megoldhassuk. Gyakori viszont az, hogy egy feladat részekre bontásánál olyan részfeladatokhoz jutunk, amelyet már az üres programmal meg lehet oldani. Az üres program tehát gyakran jelenik meg valamilyen programszerkezetbe ágyazva. (Szerepe a programban ahhoz hasonlítható, amikor a kőműves egy üres teret rak körbe téglával azért, hogy ablakot, ajtót készítsen az épülő házon. A „semmi” ezáltal válik az épület fontos részévé.)

Nézzünk most egy erőltetett példát arra, amikor egy feladatot az üres program old meg. Legyen a feladat specifikációja az alábbi:

$A = (a{:}\mathbb{N}, b{:}\mathbb{N}, c{:}\mathbb{N})$
$Ef = (a=2 \wedge b=5 \wedge c=7)$
$Uf = (Ef \wedge (c=a+b \vee c=10))$

Itt az *Ef* állításból következik az *Uf* állítás ($Ef \Rightarrow Uf$). Ez azt jelenti, hogy ha egy *Ef*-beli állapotból (tehát amire teljesül az *Ef* állítás) elindulunk, akkor erre rögtön teljesül az *Uf* állítás is: ezért nem kell semmit sem tenni ahhoz, hogy *Uf*-ben tudjunk megállni, hiszen már ott vagyunk. Ezt a feladatot tehát megoldja a *SKIP*. Általánosítva az itt látottakat azt mondhatjuk, ha egy feladat előfeltételéből következik az utófeltétele, akkor a *SKIP* program biztosan megoldja azt.

3.1.2. Rossz program

Rossz program a mindig abnormálisan működő program. ***Bármelyik állapotból indul is el, abnormális végrehajtást végez***: a kiinduló állapotot végtelen sokszor ismétlő végtelen végrehajtást fut be. A rossz programot a továbbiakban *ABORT*-tal jelöljük.

A rossz program nem túl hasznos, hiszen semmiképpen nem terminál, így sosem tud megállni egy kívánt célállapotban. A rossz programmal csak az üres feladatot, azaz a sehol sem értelmezett, tehát értelmetlen feladatot lehet megoldani: azt, amelyik egyik állapothoz sem rendel célállapotokat. A feladatok megoldása szempontjából az *ABORT*-nak tehát nincs túl nagy jelentősége. Annak az oka, hogy mégis bevezettük az, hogy program leírásunkat minél általánosabbá tegyük, segítségével minél több programot, még a rosszul működő programokat is le tudjuk írni.

3.1.3. Értékadás

A programozásban ***értékadásnak azt hívjuk, amikor a program egy vagy több változója egy lépésben új értéket kap***. Mivel a program változói az aktuális állapot komponenseit jelenítik meg, ezért az értékadás során megváltozik az aktuális állapot.

Vegyük példaként az (x:$\mathbb{R}$, y:$\mathbb{R}$) állapottéren azt a közönséges értékadást, amikor az x változónak értékül adjuk az $x*y$ kifejezés értékét. Ezt az értékadást bármelyik állapotra végre lehet hajtani, és minden esetben két állapotból (a kiinduló- és a végállapotból) álló sorozat lesz a végrehajtása. Például < (*2*,*-5*) (*-10*,*-5*) >, < (*2.3*,*0.0*) (*0.0*,*0.0*) >, < (*0.0*,*4.5*) (*0.0*,*4.5*) > vagy <(*1*,*1*) (*1*,*1*) >. Ezt az értékadást az x:=$x*y$ szimbólummal jelöljük. Az $x*y$ kifejezés hátterében egy olyan leképezés (függvény) áll, amely két valós számhoz (az aktuális állapothoz) egy valós számot (az x változó új értékét) rendeli és minden valós számpárra értelmezett.

Tekintsük most az (x:$\mathbb{R}$, y:$\mathbb{R}$) állapottéren az x:=x/y parciális értékadást. Azokban a kiinduló állapotokban, amikor az y-nak az értéke *0*, az értékadás jobboldalon álló kifejezésének nincs értelme. A jobboldali kifejezés tehát nem mindenhol, csak részben (parciálisan) értelmezett. Ebben az esetben úgy tekintünk az értékadásra, mint egy rossz programra, azaz a nem értelmezhető kiinduló állapotokból indulva abortál.[5] Ezzel biztosítjuk, hogy ez az értékadás is program legyen: minden kiinduló állapothoz rendeljen végrehajtási sorozatot.

A feladatok megoldásakor gyakran előfordul, hogy egy időben több változónak is új értéket akarunk adni. Ez a szimultán értékadás továbbra is egy állapotból egy másik állapotba vezet, de az új állapot több komponensében is eltérhet a kiinduló állapottól.[6] A háttérben itt egy többértékű függvény áll, amelyik az aktuális állapothoz rendeli azokat az értékeket, amelyeket a megfelelő változóknak kell értékül adni.

Egy értékadás jobboldalán álló kifejezés értéke nem mindig egyértelmű. Ekkor a baloldali változó véletlenszerűen veszi fel a kifejezés által adott értékek egyikét. Erre példa az, amikor egy

---

[5] Ezzel a jelenséggel a programozó például akkor találkozik, amikor az operációs rendszer "division by zero" hibaüzenettel leállítja a program futását.

[6] A szimultán értékadásokat a kódoláskor egyszerű értékadásokkal kell helyettesíteni (lásd *6.3. alfejezet*), ha a programozási nyelv nem támogatja ilyenek írását.

változónak egy halmaz valamelyik elemét adjuk értékül. Ekkor az állapottér az (x:$\mathbb{N}$, h:$\boldsymbol{2}^{\mathbb{N}}$) (a h értéke egy természetes számokat tartalmazó halmaz), az értékadás során pedig az x a h valamelyik elemét kapja. Ezt az értékadást értékkiválasztásnak (vagy nem-determinisztikus értékadásnak) fogjuk hívni és az x:$\in h$ kifejezéssel jelöljük.[7] Ez a példa ráadásul parciális értékadás is, hiszen ha a h egy üres halmaz, akkor a fenti értékkiválasztás értelmetlen, tehát abortál. Ez rávilágít arra, hogy az értékadásnak fent bemutatott általánosításai (parciális, szimultán, nem-determinisztikus) egymást kiegészítve halmozottan is előfordulhatnak. A háttérben itt egy nem-egyértelmű leképezés (tehát nem függvény), egy reláció áll.

Általános értékadásnak egy szimultán, parciális, nem-determinisztikus értékadást tekintünk. A nem-szimultán értékadást egyszerűnek, a nem parciálist teljesnek (mindenhol értelmezettnek) nevezzük, az értékkiválasztás ellentéte a determinisztikus értékadás. A közönséges értékadáson az egyszerű, teljes és determinisztikus értékadást értjük.

Az értékadás kétséget kizáróan egy program, amelyik az állapottér egy állapotához az adott állapotból induló kételemű vagy abnormálisan végtelen állapotsorozatot rendel. (Nem-determinisztikus esetben egy kiinduló állapothoz több végrehajtás is tartozhat.) Jelölésekor a „:=” szimbólum baloldalán feltüntetjük, hogy mely változók kapnak új értéket, a jobboldalon felsoroljuk az új értékeket előállító kifejezéseket. Ezek a kifejezések az állapottér változóiból, azok típusainak műveleteiből, esetenként konkrét típusértékekből (konstansokból) állnak. A kifejezés értéke a változók értékétől, azaz az aktuális állapottól függ, amit a kifejezés egy értékké alakít és azt a baloldalon álló változónak adódik értékül. Nem-determinisztikus értékadás, azaz értékkiválasztás esetén a „:=” szimbólum helyett a „:$\in$” szimbólumot használjuk.

Az értékadást nemcsak azért tekintjük elemi programnak, mert végrehajtása elemi lépést jelent az állapottéren, hanem azért is, mert könnyű felismerni, hogy milyen feladatot old meg. Ha például egy x, y, z egész változókat használó feladat utófeltétele annyival mond többet az előfeltételnél, hogy teljesüljön a $z = x+y$ állítás is, akkor nyilvánvaló, hogy ezt a z:$=x+y$ értékadás segítségével érhetjük el. Ha a megoldandó feladat az, hogy cseréljük ki két változó értékét, azaz $A = (x{:}\mathbb{Z}, y{:}\mathbb{Z})$, $Ef = (x=x' \wedge y=y')$ és $Uf = (x=y' \wedge y=x')$, akkor ezt az x,y:$=y,x$ értékadás oldja meg.

[7] A nem-determinisztikus értékadásokat a mai számítógépeken determinisztikus értékadások helyettesítik. Ez a helyettesítés azonban nem mindig a programozó feladata, hanem gyakran az a programozási környezet végzi, ahol a programot használjuk. Ez elrejti a programozó elől azt, hogyan válik determinisztikussá egy értékadás, így az kvázi nem-determinisztikus. Ez történik például a véletlenszám-generátorral történő értékadás során. Habár ez a nevével ellentétben nagyon is kiszámítható, az előállított érték a program szempontjából mégis nem-determinisztikus lesz.

3.2. Programszerkezetek

A strukturált programok építésénél három nevezetes programszerkezetet használunk: szekvenciát, elágazást, ciklust.

3.2.1. Szekvencia

Két program szekvenciáján azt értjük, amikor az egyik program végrehajtása után – feltéve, hogy az terminál – a másikat hajtjuk végre.

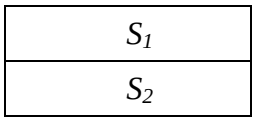
Ez a meghatározás azonban nem elegendő a szekvencia pontos megadásához. Ahhoz ugyanis, hogy egy szekvencia végrehajtási sorozatait ez alapján felírhassuk, meg kell előbb állapodni abban, hogy mi legyen a szekvencia alap-állapottere. A tagprogramok alap-állapotterei ugyanis különbözhetnek egymástól, és ilyenkor nyilvánvalóan meg kell állapodni egy közös alap-állapottérben, amely majd a szekvencia állapottere lesz. A tagprogramokat majd erre a közös állapottérre kell átfogalmazni, azaz változóik alap- illetve segédváltozói státuszát ennek megfelelően kell megváltoztatni, esetleg bizonyos változókat át kell nevezni. (Említettük, hogy az ilyen átalakítás nem változat egy program lényegén.)

A közös alap-állapottér kialakítása során előfordulhat az is, hogy az eredetileg kizárólag csak az egyik tagprogram alap- vagy segédváltozója a szekvencia közös alapváltozójává válik, vagy a szekvencia alap-állapotterébe olyan komponensek kerülnek be, amely az egyik tagprogramnak alap- vagy segédváltozói, a másik tagprogramban pedig segédváltozók voltak. (A segédváltozóra vonatkozó létrehozó és megszüntető utasítások ilyenkor érvényüket vesztik.) A közös alap-állapottér kialakítása még akkor sem egyértelmű, ha a két tagprogram alap-állapottere látszólag megegyezik. Elképzelhető ugyanis például az, hogy egy mindkét alap-állapotérben szereplő ugyanolyan nevű és típusú változót nem akarunk a szekvenciában közös változónak tekinteni. Végül szögezzük le, hogy a közös alap-állapottér komponensei kizárólag a szekvenciába fűzött tagprogramok változói közül kerülhetnek ki. Egy azoknál nem szereplő változót nem veszünk fel a közös alap-állapottérbe (ennek ugyanis nem lenne semmi értelme).

A szekvencia végrehajtási sorozatai, miután mindkét tagprogramot a közös alap-állapottéren újraértelmeztük, úgy képződnek, hogy az első tagprogram egy végrehajtásához – amennyiben a végrehajtás véges hosszú – hozzáfűződik a végrehajtás végpontjából a második tagprogram által indított végrehajtás. A szekvencia is program, hiszen egyrészt a közös alap-állapottér minden állapotából indít végrehajtást (hiszen az első tagprogram is így tesz), másrészt ezeknek a végrehajtásoknak az első állapota a kiinduló állapot (ugyancsak amiatt, hogy az első tagprogram egy program). Harmadrészt minden véges végrehajtási sorozat a közös alap-állapottérben áll meg, hiszen ezek utolsó szakaszát a második tagprogram generálja, amelynek véges végrehajtási sorozatai a közös alap-állapottérre való átalakítása után ebben az állapottérben terminálnak (lévén a második tagprogram is program).

Természetesen ezen az elven nemcsak kettő, hanem tetszőleges számú programot is szekvenciába fűzhetünk.

Az S_1 és S_2 programokból képzett szekvenciát az $(S_1;S_2)$ szimbólummal jelölhetjük vagy az alábbi rajzzal fejezhetjük ki. Ezek a jelölések értelemszerűen használhatók a kettőnél több tagú szekvenciákra is.



Milyen feladat megoldására használhatunk szekvenciát? Ha egy feladatot fel lehet bontani részfeladatokra úgy, hogy azokat egymás után megoldva az eredeti feladat megoldását is megkapjuk, akkor a megoldó programot a részfeladatot megoldó programok szekvenciájaként állíthatjuk elő. Formálisan: ha adott egy feladat (*A,Ef,Uf*) specifikációja, amelyből elő tudunk állítani egy (*A,Ef,Kf*) specifikációjú részfeladatot, valamint egy (*A,Kf,Uf*) specifikációjú részfeladatot, ahol *Kf* az úgynevezett közbeeső feltétel, akkor a részfeladatokat megoldó programok szekvenciája megoldja az eredeti feladatot. Ugyanis ekkor az első részfeladatot megoldó program elvezet *Ef*-ből (egy *Ef* által kielégített kezdőállapotból) *Kf*-be, a második a *Kf*-ből *Uf*-be, azaz összességében a szekvenciájuk eljut *Ef*-ből *Uf*-be.

Tekintsük például azt a feladatot, amikor két egész értékű adat értékét kell kicserélni. (Korábban már megoldottuk ezt a feladatot egy szimultán értékadással. Ha azonban az adott programozási környezetben nem alkalmazható szimultán értékadás, akkor mással kell próbálkozni.) Emlékeztetőül a feladat specifikációja:

$A = (x{:}\mathbb{Z}, y{:}\mathbb{Z})$
$Ef = (x=x' \wedge y=y')$
$Uf = (x=y' \wedge y=x')$

Bontsuk fel a feladatot három részre az alábbi, kicsit trükkös Kf_1 és Kf_2 közbeeső állítások segítségével. Ezáltal három, ugyanazon az állapottéren felírt részfeladat specifikációjához jutottunk. A részfeladatok állapottere azonos az eredeti állapottérrel, az első feladat előfeltétele az *Ef*, utófeltétele a Kf_1, a második előfeltétele a Kf_1, utófeltétele a Kf_2, a harmadik előfeltétele a Kf_2, utófeltétele az *Uf*.

$Kf_1 = (x=x' - y' \wedge y=y')$
$Kf_2 = (x= x' - y' \wedge y=x')$

Először próbáljunk meg az első részfeladatot megoldani, azaz eljutni a *Ef*-ből a Kf_1-be. Nyilvánvaló, hogy ezt a részfeladatot az $x:=x-y$ megoldja, hiszen kezdetben (*Ef*) x az x', y az y' értéket tartalmazza. A második részfeladatban az y-nak kell az x eredeti értékét, az x'-t megkapni, ami $(x'-y')+y'$ alakban is felírható. Mivel a Kf_1 (ez a második feladat előfeltétele) szerint kezdetben $x=x'-y'$ és $y=y'$, az ami $(x'-y')+y'$-t az $x+y$ alakban is írhatjuk. A második feladat megoldásához tehát megfelel az $y:=x+y$ értékadás. Hasonlóan okoskodva jön ki, hogy a harmadik lépéshez az $x:=y-x$ értékadás szükséges. A teljes feladat megoldása tehát az alábbi szekvencia lesz, amelyben mindhárom tagprogram és a szekvencia közös alap-állapottere is a feladat állapottere:

$x:=x-y$
$y:=x+y$
$x:=y-x$

Oldjuk meg most ugyanezt a feladatot másképpen: segédváltozóval. Bontsuk fel a feladatot három részre úgy, hogy először tegyük „félre" egy $z:\mathbb{Z}$ változóba az x változó értékét, (ezt a kívánságot rögzíti a Kf_1), ezt követően az x értéke az y változó értékét vegye fel, miközben a z változó őrzi az értékét (ezt mutatja a Kf_2), végül kerüljön át a z változóba félretett érték az y változóba. Mindhárom részfeladat állapottere az eredeti állapottérnél bővebb: $A' = (x:\mathbb{Z}, y:\mathbb{Z}, z:\mathbb{Z})$.

$$Kf_1 = (x=x' \wedge y=y' \wedge z=x')$$
$$Kf_2 = (x=y' \wedge y=y' \wedge z=x')$$

Az első részfeladatot könnyű megoldani: a Kf_1 a $z=x'$ állítással mond többet az *Ef*-nél, és az x' éppen az x változó értéke, ezért a $z:=x$ értékadás vezet el az *Ef*-ből a Kf_1-be. Ennek a programnak a z egy eredmény (nem pedig segéd) változója, az y pedig akár el is hagyható az állapotteréből. Mivel a z változó a programban ennek az értékadásnak a baloldalán jelenik meg először, ezért rá a struktogramm első értékadása mellett külön felhívjuk a figyelmet a típusának megadásával. A második részfeladatnál Kf_1-ből a Kf_2-be az $x:=y$ értékadás visz át, hiszen itt az egyetlen különbség a két állítás között az, hogy az x változó értékét az y-ban tárolt y' értékre kell átállítani. A későbbiek miatt fontos viszont, hogy ennek a programnak állapottere tartalmazza a z (bemenő és eredmény) változót is. A harmadik részfeladatban Kf_2-ből a *Uf*-be kell eljutni. Itt az y-t kell megváltoztatni úgy, hogy vegye fel az x eredeti értékét, az x'-t, amit a Kf_2 szerint a z-ben találunk. Tehát: $y:=z$. Most a z egy bemenő változó.

Mindezek alapján könnyű látni, hogy az alábbi szekvencia megoldja a kitűzött feladatot. Ehhez azonban először a program alap-állapotterét kell a feladatéhoz igazítani, leszűkíteni, amely hatására a z változó szekvencia egy segédváltozójává válik:

$z:=x$
$x:=y$
$y:=z$

A programtervezés során az a legfontosabb kérdés, hogy miről lehet felismerni, hogy egy feladatot szekvenciával kell megoldani, milyen lépésekből álljon a szekvencia, hogyan lehet megfogalmazni azokat a részcélokat, amelyeket a szekvencia egyes lépései oldanak meg, és mi lesz ezen lépéseknek a sorrendje. Ezekre a kérdésekre a feladat specifikációjában kereshetjük a választ. Például, ha az utófeltétel egy $\wedge$ kapcsolatokkal felírt összetett állítás, akkor gyakran az állítás egyes tényezői szolgálnak részcélként egy szekvenciához. (De vigyázat! Nem törvényszerű, hogy az utófeltétel $\wedge$ kapcsolata mindenképpen szekvenciát eredményez.) A részcélok közötti sorrend azon múlik, hogy az egyes részek utalnak-e egymásra, és melyik használja fel a másik eredményét. Kijelölve az utófeltételnek azt a részét, amelyet először kell teljesíteni, előáll az a közbülső állapotokat leíró logikai állítás, amely az első részfeladat utófeltétele és egyben a második részfeladat előfeltétele lesz. Hozzávéve ehhez a feladat utófeltételének itt még nem érintet felét vagy annak egy részét, megkapjuk a második részfeladat utófeltételét, és így tovább. Az eredeti előfeltétel az első részfeladat előfeltétele, az eredeti utófeltétel az utolsó részfeladat utófeltétele lesz.

3.2.2. Elágazás

Amikor egy feladatot nem egymás után megoldandó részfeladatokra, hanem alternatív módon megoldható részfeladatokra tudunk csak felbontani, akkor elágazásra van szükségünk.

Elágazás az, amikor két vagy több programhoz egy-egy feltételt rendelünk, és mindig azt a programot – úgynevezett programágat – hajtjuk végre, amelyhez tartozó feltétel az aktuális állapotra teljesül. Ha egyszerre több feltétel is igaz, akkor ezek közül valamelyik programág fog végrehajtódni, ha egyik sem, akkor az elágazás abortál.[8]

[8] A magas szintű programozási nyelvek többsége ettől eltérő módon definiálja az elágazást. Egyrészt rögzítik az ágak közti sorrendet azzal, hogy mindig az első olyan ágat hajtják végre, amelyiknek feltételét kielégíti az aktuális állapot. Másrészt, ha egyik feltétel sem teljesül, akkor az elágazás az üres programmal azonos. Mi ezt azért nem vesszük át, hogy a programjaink

Az elágazás alap-állapotterét megállapodás alapján alakítjuk ki az elágazás feltételeit alkotó kifejezésekben használt változókból és a programágak változóiból, többnyire azok uniójából. A feltételek változói a közös állapottér bemenő változói lesznek. Előfordulhat itt is, hogy két programág látszólag ugyanazon változóit, előbb átnevezéssel meg kell egymástól különböztetni.

Készítsük el az alábbi elágazást! Vegyük az (x:$\mathbb{Z}$, y:$\mathbb{Z}$, max:$\mathbb{Z}$) állapottéren a max:=x és a max:=y értékadásokat. Rendeljük az elsőhöz az $x \geq y$, a másodikhoz az $x \leq y$ feltételt. Tekintsük azt a programot, amelyik az $x \geq y$ feltétel teljesülése esetén a max:=x, az $x \leq y$ feltétel bekövetkezésekor a max:=y értékadást hajtja végre. Az így konstruált elágazás a (*8,2,0*) állapothoz (mivel erre az első feltétel teljesül) a $\langle$(*8,2,0*),(*8,2,8*)$\rangle$ sorozatot, az (*1,2,0*)-hoz (erre a második feltétel teljesül) az $\langle$(*1,2,0*),(*1,2,2*)$\rangle$ sorozatot rendeli. A (*2,2,0*) állapot mindkét feltételt kielégíti, ezért rá tetszés szerint bármelyik értékadás végrehajtható. A példában ezek végrehajtásának eredménye azonos; mindkét esetben a $\langle$(*2,2,0*),(*2,2,2*)$\rangle$ sorozatot kapjuk.

Az elágazás ágai között semmiféle elsőbbségi sorrend nincs, ezért ha egy kiinduló állapotra egyszerre több feltétel is teljesül, akkor az elágazás véletlenszerűen választja ki azt az ágat, amelynek feltételét az adott állapot kielégíti. Az elágazás tehát annak ellenére nem-determinisztikussá válhat, ha az ágai egyébként determinisztikus programok. Az már a programozó felelőssége, hogy ilyen esetben is biztosítsa, hogy az elágazás jól működjön. (A példában a feltételeknek legalább egyike mindig teljesül.) Olyan elágazásokat is lehet szerkeszteni, ahol egy kiinduló állapot egyik feltételt sem elégíti ki. Ilyenkor azt kell feltételeznünk, hogy a programozó nem számolt ezzel az esettel, az elágazásnak tehát "hivatalból" rosszul kell működnie. Ezért ilyenkor az elágazás definíció szerint abortál. Sokszor a feltételeket úgy alakítjuk ki, hogy azok közül az állapottér minden elemére pontosan az egyik teljesüljön. Ezzel kizárjuk mind a nem-determinisztikussá válás, mind az abortálás esetét.

Az elágazás nyilvánvalóan program, hiszen minden állapothoz rendel önmagával kezdődő végrehajtási sorozatot: ha egy állapot valamelyik feltételt kielégíti, akkor az ahhoz tartozó programág által előállított sorozatot, ha egyik feltételt sem elégíti ki, akkor az abortáló sorozatot.

Az $S_1, \ldots, S_n$ A állapottér feletti programokból és az $f_1, \ldots, f_n$ A állapottér feletti feltételekből képzett elágazást az $IF(f_1{:}S_1, \ldots, f_n{:}S_n)$-nel jelöljük és az alábbi ábrával rajzoljuk:

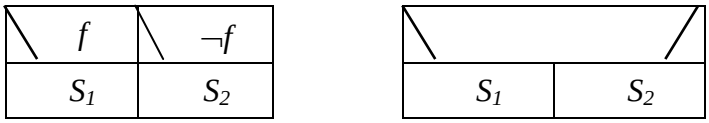
f_1	. . .	f_n
S_1	. . .	S_n

---

helyessége nem függjön attól, hogy egyrészt az elágazás eseteit milyen sorrendben gondoltuk végig, másrészt, ha egy esetre elfelejtettünk gondolni (egyik feltétel sem teljesül), akkor a program erre figyelmeztetve hibásan működjön.

Ha olyan *n* ágú elágazást kell jelölnünk, ahol az *n*-edik ág feltétele akkor teljesül, ha az előző ágak egyik feltétele sem, azaz $f_n = \neg f_1 \wedge \ldots \wedge \neg f_{n-1}$, akkor az f_n–t helyettesíthetjük az *else* jelöléssel.

A leggyakrabban előforduló elágazások kétágúak, amelyek között gyakran fogunk találkozni olyanokkal, amelyekben egyik feltétel a másik ellentéte. Az alábbi ábrák egymással egyenértékű módon fejezik ki az ilyen elágazásokat.



Milyen feladat megoldására használhatunk elágazást? Például, ha a feladat utófeltétele esetszétválasztással határozza meg a célt. Ezt gyakran a „ha-akkor” szófordulatokkal (logikai implikáció jelével) fejezzük ki. Ilyenkor először meg kell határoznunk az egyes eseteket leíró feltételeket (f_i). Ezután meg kell bizonyosodnunk arról, hogy az *Ef*-nek eleget tevő bármelyik állapotra legalább ez egyik feltétel teljesül (azaz $Ef \Rightarrow f_1 \wedge \ldots \wedge f_n$), mert különben az elágazás abortál. A részfeladatok kialakításánál az alábbiak szerint járunk el. Az *i*-edik esethez (ághoz) tartozó részfeladat kezdőállapotai az eredeti feladat azon a kezdőállapotai, amelyek eleget tesznek az *i*-edik feltételének is. A részfeladat célállapotai az eredeti feladat célállapotai. Az *i*-edik ág részfeladatának specifikációja tehát: $(A, Ef \wedge f_i, Uf)$. Mindezek alapján belátható, hogy ha $Ef \Rightarrow f_1 \wedge \ldots \wedge f_n$ és minden $i \in \{1..n\}$-re az S_i programág megoldja az $(A, Ef \wedge f_i, Uf)$ részfeladatot, akkor az $IF(f_1:S_1, \ldots, f_n:S_n)$ elágazás megoldja az (A, Ef, Uf) feladatot.

Példaként oldjuk meg azt a feladatot, amelyben egy egész számot a szignumával kell helyettesítenünk. A pozitív számok szignuma *1*, a negatívoké *–1*, a nulláé pedig *0*. A feladat specifikációja:

$A = (x:\mathbb{Z})$
$Ef = (x = x')$
$Uf = (x = sign(x'))$

ahol $sign(z) = \begin{cases} 1 & \text{ha} \quad z > 0 \\ 0 & \text{ha} \quad z = 0 \\ -1 & \text{ha} \quad z < 0 \end{cases}$

Az utófeltétel három esettel specifikálja a *x* változó új értékét: $x>0$, $x=0$ és $x<0$. Ez a három eset lefedi az összes lehetséges esetet és egyszerre közülük csak az egyik lehet igaz. A szignum függvény definíciójából látszik, hogy az első esetben az *x* új értéke az *1* lesz, második esetben a *0*, harmadik esetben a *–1*. Világos tehát, hogy az alábbi elágazás megoldja a feladatot.

$x>0$	$x=0$	$x<0$
$x:=1$	$x:=0$	$x:=-1$

3.2.3. Ciklus

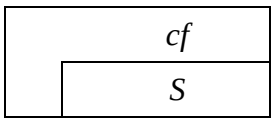
Ciklus működése során egy adott programot, az úgynevezett ciklusmagot egymás után mindannyiszor végrehajtjuk, valahányszor olyan állapotban állunk, amelyre egy adott feltétel, az úgynevezett ciklusfeltétel teljesül.

Tekintsünk egy példát. Legyenek az állapottér állapotai a $[-5 .. +\infty[$ intervallumba eső egész számok. Válasszuk ciklusmagnak azt a programot, amely az aktuális állapotot (egész számot) mindig a szignumával növeli meg, azaz ha az állapot negatív, akkor csökkenti eggyel, ha pozitív, növeli eggyel, ha nulla, nem változtatja az értékét. Legyen a ciklusfeltétel az, hogy az aktuális állapot nem azonos a *20*-szal. Amikor az *5* állapotból indítjuk a ciklust, akkor a ciklusmag első végrehajtása elvezet a *6* állapotba, majd másodszorra a *7*-be és így tovább egészen addig, amíg tizenötödször is végrehajtódik a ciklusmag, és a *20* állapotba nem jutunk (ez már nem elégíti ki a ciklusfeltételt), itt a ciklus leáll. A ciklus tehát az *5* kiinduló állapotból az $\langle 5,6,7, \ldots ,19,20\rangle$ állapotsorozatot futja be. Amennyiben a *20* a kiinduló állapot, a ciklus rögtön leáll, a *20*-hoz tehát a $\langle 20\rangle$ egy elemű végrehajtás tartozik. A *21* vagy annál nagyobb állapotokról indulva a ciklus sohasem áll le: a ciklusmag minden lépésben növeli eggyel az aktuális állapotot, és az egyre növekedő állapotok ($\langle 21,22,23, \ldots \rangle$) egyike sem lesz egyenlő *20*-szal, azaz nem elégítik ki a ciklusfeltételt. Ugyanez történik amikor a *0* állapotból indulunk el, hiszen ekkor a ciklusmag sohasem változtatja meg az aktuális állapotát, azaz a $<0,0,0, \ldots >$ végtelen sorozat hajtódik végre. (Mindkét esetre azt mondjuk, hogy „a program végtelen ciklusba esett”.) A *–1*-ről indulva viszont lépésről lépésre csökkenti a ciklus az aktuális állapotot, de amikor a *–5*-höz ér, a ciklus működése elromlik. A ciklusmag ugyanis nem képes a *–5*-öt csökkenteni, hiszen ez kivezetne az állapottérből; a ciklusmag a *–5*-ben abortál. A teljes végrehajtási sorozat a $\langle -1,-2,-3,-4,-5,-5,-5, \ldots \rangle$.

Egy ciklus végrehajtását jelképező állapotsorozat annyi *szakaszra* bontható, ahányszor a ciklusmagot végrehajthattuk egymás után. Egy-egy szakasz a ciklusmag egy-egy végrehajtása. A példánkban ez egyetlen lépés volt (tehát egyetlen érték képviselt egy szakaszt), de általában a ciklusmag egy végrehajtása hosszabb állapotsorozatot is befuthat. Egy-egy szakasz kiinduló állapota mindig kielégíti a ciklusfeltételt. (A szakasz többi állapotára ennek nem kell fennállnia.) Amennyiben a szakasz véges hosszú és a végpontja újra kielégíti a ciklusfeltételt, a ciklusmag ebből a pontból újabb szakaszt indít.

A ciklus egy végrehajtása lehet véges vagy végtelen. A véges végrehajtások véges sok véges hosszú szakaszból állnak, ahol a szakaszok kiinduló állapotai kielégítik a ciklusfeltételt, az utolsó szakasz végállapota viszont már nem. A véges végrehajtások egy speciális esete az, amikor már a ciklus kiinduló állapota sem elégíti ki a ciklusfeltételt, így nem kerül sor a ciklusmag egyetlen végrehajtására sem. Ilyenkor a végrehajtás egyelemű sorozat lesz. A végtelen végrehajtások kétfélék. Vannak olyanok, amelyek végtelen sok véges hosszú szakaszból állnak. Itt minden szakasz egy ciklusfeltételt kielégítő állapotban végződik. („végtelen ciklus"). Más végtelen végrehajtások ellenben véges sok szakaszból állnak, de a legutolsó szakasz végtelen hosszú. Ilyenkor a ciklusmag egyszer csak valamilyen okból nem terminál. Ez a „hibás ciklusmag" esete.

A ciklus alap-állapotterét a ciklusmag alap-állapottere és a ciklusfeltételt adó kifejezés változóiból megállapodás alapján alakítjuk ki. A ciklus tényleg program, hiszen az állapottér minden pontjához rendel önmagával kezdődő sorozatot. A ciklust a továbbiakban a *LOOP*(*cf, S*) szimbólummal, illetve a



ábrával fogjuk jelölni.

Felismerni, hogy egy feladatot ciklussal lehet megoldani, majd megtalálni ehhez a megfelelő ciklust, nem könnyű. Ez csak sok gyakorlás során megszerezhető intuíciós készség segítségével sikerülhet. Bizonyos „ökölszabályok" természetesen felállíthatóak, de a szekvencia illetve az elágazásokhoz képest összehasonlíthatatlanul nehezebb egy helyes ciklus konstruálása. Ahhoz, hogy egy ciklus megold-e egy *Ef* és *Uf*-fel specifikált feladatot, két kritériumot kell belátni. Az egyik, az úgynevezett ***leállási kritérium***, amely azt biztosítja, hogy az *Ef*-ből (*Ef* által kijelölt állapotból) elindított ciklus véges lépés múlva biztosan leáll, azaz olyan állapotba jut, amelyre a ciklusfeltétel hamis. A másik, az úgynevezett ***haladási kritérium***, azt biztosítja, hogy ha az *Ef*-ből elindított ciklus valamikor leáll, akkor azt jó helyen, azaz *Uf*-ben (*Uf*-beli állapotban) teszi.

A leállási kritérium igazolásához például elegendő az, ha a ciklus végrehajtása során a ciklusfeltétel ellenőrzésekor érintett állapotokban meg tudjuk mondani, hogy a ciklus magját még legfeljebb hányszor fogja a ciklus végrehajtani, és ez a szám a ciklusfeltétel teljesülése esetén mindig pozitív és a ciklusmag egy végrehajtása után legalább eggyel csökken. Ekkor ugyanis csak véges sokszor fordulhat elő, hogy ezen állapotokban a ciklusfeltétel igazat ad, tehát a ciklusnak véges lépésen belül meg kell állni. Vannak olyan feladatokat megoldó ciklusok is, amelyek leállásának igazolása ennél jóval nehezebb, van, amikor nem is dönthető el.

A haladási kritérium igazolásához a ciklus úgynevezett ***invariáns tulajdonságát*** kell megtalálnunk. Ez a ciklusra jellemző olyan állítás (jelöljük *I*-vel), amelyet mindazon állapotok kielégítenek, ahol a ciklus akkor tartózkodik, amikor sor kerül a ciklusfeltételének kiértékelésére. Megfordítva, egy *I* állítást akkor tekintünk invariánsnak, ha teljesül rá az, hogy az *I*-t és a ciklusfeltételt egyszerre kielégítő állapotból elindulva a ciklusmag végrehajtása egy *I*-t kielégítő állapotban fog megállni, azaz $I \wedge cf$-ből *I*-be jut. Az invariáns állítást kielégítő állapotokból indított ciklus működése tehát ebben az értelemben jól kiszámítható, hiszen biztosak lehetünk abban, hogy amennyiben a ciklus leáll, akkor *I*-beli állapotban fog megállni. Egy ciklusnak több invariáns állítása is lehet, de ezek közül nekünk arra van szükségünk, amely segítségével meg tudjuk mutatni, hogy a ciklus megoldja az *Ef* és *Uf*-fel specifikált feladatot, azaz – ha leáll – akkor eljut az *Ef*-ből az *Uf*-be. Ehhez egyrészt a *I* invariánsnak érvényesnek kell lennie már minden *Ef*-beli kezdőállapotra (azaz $Ef \Rightarrow I$), hiszen a *I*-nek már a ciklus elindulásakor (a ciklusmag első végrehajtásának kezdőállapotára) teljesülnie kell. Másrészt bizonyítani kell, hogy a megálláskori állapot, amely még mindig kielégíti a *I*-t, de már nem elégíti ki a ciklusfeltételt (hiszen ezért állt meg a ciklus), az egyben az *Uf*-et is kielégíti (tehát $I \wedge \neg cf \Rightarrow Uf$).

Alkalmazzuk az elmondottakat egy konkrét példára. Legyen a megoldandó feladat az, amikor keressük egy összetett szám legnagyobb valódi osztóját. Ezt a feladatot korábban már specifikáltuk, de attól eltérő módon az előfeltételben most tegyük fel azt is, hogy a *d* eredmény változó értéke kezdetben az *n–1* értékkel egyezik meg. (Ennek hiányában a ciklus előtt egy értékadást is végre kellene hajtani, de a példában most nem akarjuk a szekvenciákról tanultakat is alkalmazni.)

$A = (n{:}\mathbb{N}, d{:}\mathbb{R})$

$Ef = (n=n' \wedge összetett(n) \wedge d=n-1)$

$Uf = (n=n' \wedge összetett(n) \wedge d=LVO(n))$

Tekintsük az *A* állapottéren az alábbi programot:

	$\neg\, d \mid n$
	$d{:=}d-1$

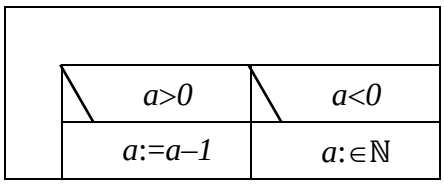
Invariáns állításnak válasszuk azt, amely azokat az állapotokat jelöli ki, amelyekben a *d* változó értéke valahol *2* és *n-1* közé esik, és a *d*-nél nagyobb egész számok biztosan nem osztói az *n*-nek.

$I = (n=n' \wedge összetett(n) \wedge LVO(n) \le d \le n-1)$

Ez tényleg invariáns a ciklusmag működésére, hiszen ha egy állapotra teljesül a *I* és még a ciklusfeltétel is (azaz a *d* maga sem osztja az *n*-t, és emiatt *d* biztos nagyobb, mint az *n* legnagyobb valódi osztója, és persze nagyobb *2*-nél is), akkor eggyel csökkentve a *d* értékét (ezt

teszi a ciklus magja) ismét olyan állapotba jutunk, amely kielégíti az *I*-t. Nyilvánvaló, hogy *I* gyengébb, mint *Ef*, tehát $Ef \Rightarrow I$. Az is teljesül, hogy $I \wedge \neg cf \Rightarrow Uf$ (ha $LVO(n) \le d \le n-1$ és $d \mid n$, akkor $d=LVO(n)$). Összességében tehát a ciklus egy *Ef*-beli állapotból elindulva, ha megáll valamikor, akkor *Uf*-beli állapotban fog megállni. A leállási kritérium igazolása érdekében minden állapotban tekintsünk a *d* értékére úgy, mint a hátralevő lépések számára adott felső korlátra: ez, mint látjuk a ciklusfeltétel teljesülése esetén pozitív és minden lépésben eggyel csökken az értéke, tehát a program biztosan meg fog állni.

Tekintsük most azt a feladatot, amelynek állapottere az $a:\mathbb{Z}$ és bármelyik kiinduló egész számhoz a nullát rendeli célállapotként. Ezt nyilván megoldaná az $a:=0$ értékadás is, de mi most az alábbi ciklust vizsgáljuk meg.



A leállási kritérium igazolásához itt a hátralevő lépések számára adott felső becslés módszere (amit az előző példában használtunk) nem alkalmazható. Egy tetszőleges kiinduló állapotra ugyanis csak akkor tudjuk felülről megbecsülni a hátralevő lépések számát, ha a kiinduló érték nem negatív. Ezt ugyanis a ciklusmag újra és újra eggyel csökkenti, amíg az nulla nem lesz. Ha azonban az *a* változó kiinduló értéke negatív, akkor ilyen becslés nem adható. A ciklusmag első végrehajtása során a változó értéke egy előre meghatározhatatlan nem-negatív egész szám lesz, amelyből tovább haladva a ciklus (az előzőekben elmondottak szerint) már biztosan le fog állni a nullában. A ciklus tehát véges lépésben biztos a kívánt célállapotban (a nullában) fog befejeződni, de nem minden kiinduló állapotra lehet megadni a hátralevő lépések számának felső korlátját. A haladási kritériumhoz viszont elegendő a semmitmondó azonosan igaz invariánst választani. Nyilvánvaló ugyanis, hogy ha a ciklus leáll, akkor azt kíván célban, a nulla értékű *a*-val teszi.

3.3. Helyes program, megengedett program

A programtervezés célja az, hogy egy feladathoz olyan absztrakt (például struktogrammal leírt) programot adjon, amelyik megoldja a feladatot és ugyanakkor könnyen kódolható a konkrét programozási nyelveken. Ez előbbi kritériumot a program helyességének, az utóbbit a megengedhetőségének nevezzük.

Ha egy program megold egy feladatot, akkor azt a feladat szempontjából ***helyes programnak*** hívjuk.

Talán első olvasásra meglepőnek tűnik, de helyes programot nagyon könnyen lehet készíteni, hiszen minden feladat megoldható egyetlen alkalmas értékadás segítségével. Amikor ugyanis egy feladat specifikációjában – közvetve vagy közvetlenül – meghatározzuk minden változónak a cél (célállapotbeli) értékét, tulajdonképpen kijelölünk egy értékadást, amelynek a feladat által előírt értékeket kell átadnia a megfelelő változóknak. Elméletben tehát minden feladathoz megkonstruálható az őt megoldó értékadás, amelyet a ***feladat triviális megoldásának*** nevezünk.[9] Ilyen lehetne például a korábban ciklussal megoldott legnagyobb valódi osztót előállító feladat esetében a $d{:=}LVO(n)$ értékadás.

A programozási gyakorlatban azonban igen ritka, hogy egy feladatot egyetlen értékadással oldanánk meg. Ennek az oka az, hogy a triviális megoldás jobboldalán többnyire olyan kifejezés áll, amelyik a rendelkezésünkre álló programozási környezetben nem megengedett, azaz nem ismertek a kifejezésben szereplő műveletek vagy érték-konstansok, tehát a kifejezés közvetlenül nem kódolható. Habár a programtervezés során szabad, sőt célszerű elvonatkoztatni attól a programozási környezettől, ahol majd a programunknak működnie kell, de a végleges programterv nem távolodhat el túlságosan tőle, mert akkor nehéz lesz kódolni. Számunkra azok a programtervek a kívánatosak, amelyek egyfelől kellően általánosak (absztraktak) ahhoz, hogy előállításuknál ne ütközzünk minduntalan a programozási környezet által szabott korlátokba, másfelől bármelyik programozási nyelven könnyen kódolhatóak. Szerencsére a programozási környezetek fejlődésük során egyre inkább közelítettek és közelítenek egy hivatalosan ugyan nem rögzített, de a gyakorlatban nagyon is jól körvonalazott szabványhoz, amely kijelöli a legtöbb programozási nyelv által biztosított nyelvi elemeket (adattípusokat, programszerkezeteket). Célszerű a tervezés során készített programokat e szabványhoz, ezen ideális környezethez igazítani. Az ilyen programokat fogjuk a továbbiakban megengedett programoknak nevezni.

Programozási modellünkben megállapodás alapján jelöljük ki a ***megengedett típusokat***. Ide tartoznak a számok alaptípusai (természetes-, egész-, valós számok) a karakterek és a logikai értékek típusa. Megengedett típusokból megfelelő (azaz megengedett) eljárásokkal (úgynevezett konstrukciókkal) olyan összetett típusokat készíthetünk (például ugyanazon megengedett típusú értékeket tartalmazó egy vagy többdimenziós tömb, karakterekből összefűzött lánc vagy sztring), amelyeket ugyancsak megengedettnek tekintünk. Később (lásd *7. fejezet*) tovább bővítjük a megengedett típusok körét.

[9] A modellünkben bevezetett feladatok általános leképezések, amelyekből jóval több van, mint az úgynevezett parciális rekurzív függvényekből. Említettük már, hogy csak ez utóbbiak megoldásához lehet olyan programokat készíteni, amelyek algoritmizálhatóak is. Modellünkben ellenben bármelyik feladathoz kreálható megoldó program (például a triviális megoldás), ezek között tehát bőven vannak olyanok, amely nem algoritmizálhatók.

Egy kifejezés megengedett, ha azt megengedett típusok értékeiből, műveleteiből és ilyen típusú változókból alakítunk ki a megadott alaki szabályoknak (szintaktikának) megfelelően.[10]

Egy értékadás akkor megengedett, ha a jobboldalán szereplő kifejezés megengedett. A bevezetőben említett feladat finomítás során olyan elemi részfeladatokra kívánjuk bontani a megoldandó feladatot, amelyek megengedett értékadással megoldhatók, azaz olyan programmal, amelyet nem kell már mélyebben részletezni.

Egy megengedett tagprogramokból felépülő szekvenciát megengedettnek tekintünk. Az elágazások és ciklusok feltételei logikai értékű kifejezések, amelyek szintén lehetnek megengedettek. Ha egy elágazást vagy ciklust megengedett feltételekből és megengedett programokból konstruálunk, akkor megengedettnek mondjuk. ***A megengedett program szekvencia, megengedett feltételeket használó elágazások és megengedett ciklusfeltételű ciklusok segítségével az üres programból és megengedett kifejezésű értékadásokból felépített program.***

Összefoglalva, egy programozási feladat megoldásán azt a programot értjük, amelyik helyes és megengedett.

[10] A típusműveletek valójában leképezések, ezért általános esetben relációként írhatók fel. Maga a kifejezés is egy leképezés, amely a típusműveleteknek, mint leképezéseknek a kompozíciójaként (összetételeként) áll elő. A fenti meghatározásban említett alaki szabályok tehát jól definiáltak.

II. RÉSZ
PROGRAMTERVEZÉS MINTÁK ALAPJÁN

A programtervezés során az a célunk, hogy egy feladat megoldására helyes és megengedett programot találjunk. A helyesség azt biztosítja, hogy a program megoldása legyen a feladatnak, a megengedettség pedig arra ad garanciát, hogy a programot különösebb nehézség nélkül kódolni tudjuk a választott konkrét programozási nyelvre.

Önmagában már az is nehéz, ha adott feladat és adott program viszonyában el kell döntenünk, hogy a program helyes-e, azaz megoldja-e a feladatot. Erre különféle módszerek vannak kezdve a teszteléstől a szigorú matematikai formalizmust igénylő helyességbizonyításig. Mindkét említett módszer azonban feltételezi, hogy már van egy programunk, pedig egy feladat megoldása kezdetén nem rendelkezünk semmilyen programmal, csak a feladat ismert. Megfelelő program előállításánál érdemes egy eleve helyes megoldásból, a feladat triviális megoldásából (értékadásból) kiindulni. Mivel azonban ez többnyire nem-megengedett, fokozatosan, lépésről lépésre (top-down) kell azt átalakítunk úgy, hogy a program helyessége minden lépés után megmaradjon, és e ***lépésenkénti finomítás*** végére megengedetté váljon. Az átalakítás közbülső formái, amelyeket éppúgy tekinthetünk a feladat egyfajta változatának, mint programnak, egyre összetettebb programszerkezetek, amelynek nem-megengedett értékadásait részfeladatoknak foghatjuk fel, és amelyeket mélyebben részletezett programmal kell kiváltani. A finomítási eljárás közbülső formáinak eme kettős jelentése, miszerint a benne szereplő nem-megengedett értékadások feladatok is és programok is, a feladat-orientált programozási módszertan egyik fő jellegzetessége.

Egy nem-megengedett értékadást vagy egy bonyolultabb szerkezetű programmal kell helyettesíteni, vagy az értékadás jobboldali kifejezésének kiszámításánál használt típusműveleteket kell megengedetté tenni, amihez adattípusokat kell konstruálunk. Az első eljárást ***funkció orientált programtervezésnek*** hívjuk (ehhez kapcsolódik a könyv *II. része*); a másodikat pedig ***típus orientált programtervezésnek*** nevezzük (ezzel a *III. részben* találkozunk). Ez a két eljárás nem zárja ki egymást, vegyesen alkalmazhatóak.

Ebben a részben egy olyan lépésenkénti finomítást mutatunk be, amely korábbi megoldások analógiájára épül. Ennek lényege az, hogy egy feladat megoldását egy ahhoz hasonló, korábban már megoldott feladat megoldására támaszkodva állítjuk elő. Ennek az analóg programozásnak egyik fajtája a ***visszavezetés***, amihez a *4. fejezetben* nevezetes minta-megoldásokat (úgynevezett programozási tételeket) vezetünk majd be, és ezek gyakorlatban történő alkalmazását az *5. fejezet* példáival illusztráljuk. A *6. fejezetben* olyan összetett feladatokat vizsgálunk, amelyek megoldása visszavezetéssel előállított részprogramokból épül fel.

4. Programozási tételek

Az analóg módon történő programozás a programkészítés általánosan ismert és alkalmazott technikája. Amikor egy olyan feladatot kell megoldani, amelyhez hasonló feladatot már korábban megoldottunk, akkor a megoldó programot a korábbi feladat megoldó programja alapján állítjuk elő. Ez az ötlet, nem új keletű, nem kizárólag a programozásnál alkalmazott gondolat, hanem olyan általános problémamegoldó módszer, amellyel az élet minden területén találkozhatunk. Amikor egy tevékenységgel kapcsolatban gyakorlati tapasztalatról, megtanult ismeretekről beszélünk, akkor olyan korábban megoldott problémákra és azok megoldásaira gondolunk, amelyek az adott tevékenységgel kapcsolatosak. Minél nagyobb egy emberben ez a fajta ismeret, minél biztosabban tudja ezt alkalmazni egy új feladat megoldásánál, annál inkább tekintjük őt a szakmája mesterének. Nem meglepő hát, hogy a programozásban az itt tárgyalt úgynevezett analóg programozás szinte kivétel nélkül minden gyakorló programozó eszköztárában jelen van.

Az analóg programozást lehet alkalmazni ösztönösen vagy tudatosan, elnagyoltan vagy precízen, laza ajánlásként vagy pontos technológiai szabványként. Az ebben a könyvben alkalmazott visszavezetéses módszerénél mind a kitűzött feladatnak, mind a korábban megoldott mintafeladatnak ismerjük a precíz leírását, a specifikációját, amely alkalmas arra, hogy a két feladat közötti analógiát pontosan felfedjük: könnyen felismerhetjük a hasonlóságot, ugyanakkor világosan felderíthetjük az eltéréseket. ***A visszavezetés során a mintafeladat megoldó programját (a mintaprogramot) sablonként használva állítjuk elő a kitűzött feladat megoldó programját úgy, hogy figyelembe vesszük a kitűzött- és a mintafeladat specifikációkban felfedett eltéréseket.***

Ebben a fejezetben először összevetjük a visszavezetést más analóg programozási technikákkal, majd megvizsgálunk néhány olyan nevezetes mintafeladatot és annak megoldó algoritmusát, (***programozási tételt***), amelyeket a visszavezetések során használhatunk.

4.1. Analóg programozás

Az analóg programozási technikák alapgondolata közös, de azok gyakorlatában jelentős különbségek fedezhetőek fel. Ha az analóg programozásban érvényesülő eltérő szemléletmódokat gondolatban egy egyenes mentén képzeljük el, akkor ennek egyik végén az a gondolkodásmód áll, amelyiknél az új feladat megoldásakor a mintafeladat megoldásának előállítási folyamatát másoljuk le; a másik véglet viszont csak az eredményt, a megoldó programot veszi át, és azt igazítja hozzá az új feladathoz.

Az első esetre példa az, amikor a mintafeladatot megoldó programot algoritmikus gondolkodás útján (milyen programszerkezetet használjunk, mi után mi következzen, mi legyen a ciklus vagy elágazás feltétele, stb.) állítottuk elő, és az ehhez hasonló feladatokat a

mintafeladatnál alkalmazott gondolatsorral, annak ötleteit kölcsönözve, lemásolva, analóg algoritmikus gondolkodással oldjuk meg.

Ettől lényeges különbözik az, amikor nem a mintaprogram előállítási folyamatát, az előállításának lépéseit másoljuk le, ismételjük meg, hanem csak magát a mintaprogramot adaptáljuk a kitűzött feladatra. Az analóg programozásnak ezt a technikáját hívjuk visszavezetésnek. Ennek hatékony alkalmazásához elengedhetetlen a feladatok pontos leírása, hiszen csak így lehet felfedni azokat a részleteket, amelyekben egyik feladat a másiktól eltér (az ördög a részletekben bújik meg), és csak ezek után tudjuk megmondani azt is, hogy a már megoldott mintafeladat programjában mely részleteket mire kell kicserélni ahhoz, hogy a kitűzött feladat megoldását megkapjuk. Ebben segít a feladat specifikációja. Ha a feladatokat (mind a kitűzött, mind a mintákat) specifikáljuk, akkor nemcsak a közöttük levő hasonlóságot tudjuk könnyen felismerni, hanem pontosan felderíthetőek az eltérések is. Amíg tehát az előbbi esetben a programtervezés hangsúlya az algoritmikus gondolkodáson van, addig az utóbbiban már nem úgy tekintünk a megoldó programra, mint egy működő folyamatra, annak tervezése a feladat elemzésén, tehát egyfajta statikus szemléleten alapul. Könyvünkben bemutatott módszertannak talán a legnagyobb vívmánya ez a precíz specifikációra épülő visszavezetési technika, amely az analóg programozást lényegesen gyorsabbá és biztonságosabbá teszi.

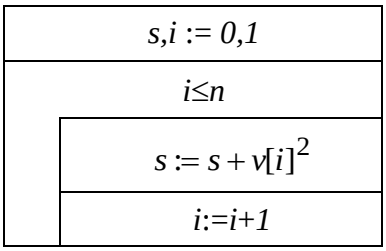
Most algoritmikus gondolkodás segítségével megoldunk egy olyan feladatot, amely mintafeladatként fog szolgálni a későbbi feladatok megoldásához, amelyek közül egyet analóg algoritmikus gondolkodással, a többit visszavezetéssel oldunk majd meg.

4.1. Példa. Adjuk össze egy n elemű egész értékű tömb elemeinek négyzeteit.

Specifikáljuk először a feladatot.

$$A = (v{:}\mathbb{Z}^n, s{:}\mathbb{Z})$$
$$Ef = (v{=}v')$$
$$Uf = (Ef \wedge s = \sum_{i=1}^{n} v[i]^2)$$

A megoldás során a kívánt összeget fokozatosan állítjuk. Ehhez végig kell vezetnünk egy i egész típusú változót a tömb indexein, és minden lépésben hozzá kell adni a kezdetben nullára beállított s változóhoz a $v[i]^2$ értéket. A megoldó program tehát a kezdeti értékadások után (ahol az s változó értékét nullára, az i változóét egyre állítjuk) egy ciklus, amely az i változót egyesével növeli egészen n-ig. A ciklus működése alatt az s változó mindig a v tömb első $i-1$ elemének négyzetösszegét tartalmazza, Kezdetben, amikor az i értéke 1, ez az érték még nulla, befejezéskor, amikor az i eléri az $n+1$ értéket, már a végleges eredmény.



Az *n* nem új változó, hanem a tömb hossza

4.2. Példa. Számítsuk ki az *n* pozitív egész szám faktoriálisát, azaz az *1*2*...*n* szorzatot!

Írjuk fel először a feladat specifikációját. Az utófeltételben használjuk azt a produktum-kifejezést, amelynek segítségével egy $m*(m+1)*...*(n-1)*n$ szorzat a $\prod_{i=m}^{n} i$ zárt alakban felírható. A jelölésről tudni kell, hogy ha *m>n*, akkor a produktum értéke *1*. (Az *1* ugyanolyan szerepet tölt be az egész számok szorzásánál, mint a *0* az összeadásnál. Egy *0*-hoz adott szám vagy egy *1*-gyel megszorzott szám maga a szám lesz. Azt mondjuk, hogy a *0* az összeadásra vett nulla elem, míg az *1* a szorzás nulla eleme. Egy nulla elemmel történő műveletvégzés nem változtat azon az értéken, amelyre a művelet irányul.)

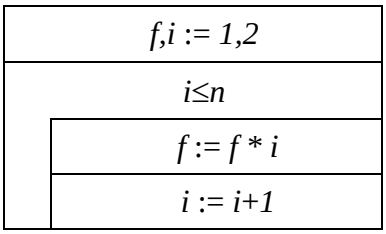
$A = (n{:}\mathbb{N}, f{:}\mathbb{N})$

$Ef = (n=n')$

$Uf = (Ef \wedge f = \prod_{i=2}^{n} i)$

A kitűzött feladat és az *4.1. példa* feladata hasonlít egymásra, ami a specifikációk összevetéséből kiválóan látszik. Ott a $v[i]^2$ kifejezések (*i=1..n*) értékeit kellett összeadni és ezt az s változóban elhelyezni, itt a *i* számokat (*i=2..n*) összeszorozni és az *f* változóban tárolni. Készítsünk megoldást analóg algoritmikus gondolkodással! Figyeljük meg, hogy csak az aláhúzott szavak változtak meg *4.1. példa* megoldásának gondolatmenetéhez képest.

A megoldás során a kívánt szorzatot fokozatosan állítjuk elő egy ciklussal. Ehhez végig kell vezetnünk egy *i* egész típusú változót a *2..n* tartomány indexein, és minden lépésben hozzá kell szorozni a kezdetben egyre beállított *f* változóhoz az *i* értéket. A megoldó program tehát a kezdeti értékadások után (ahol az *f* változó értékét egyre, az *i* változóét kettőre állítjuk) egy ciklus, amely az *i* változót egyesével növeli egészen *n*-ig. A ciklus működése alatt az *f* változó mindig az *i-1* faktoriálisát tartalmazza, Kezdetben, amikor az *i* értéke *2*, ez az érték még egy, befejezéskor, amikor az *i* eléri az *n+1* értéket, már a végleges eredmény.



4.3. Példa. Adott két azonos dimenziójú vektor, amelyek valós számokat tartalmaznak. Számítsuk ki ezek skaláris szorzatát!

A feladat bemenő adatait két *1*-től *n*-ig indexelt tömbben tároljuk, eredménye a skaláris szorzat; ezt tükrözi az állapottér. Az utófeltételben a skaláris szorzat definíciója jelenik meg.

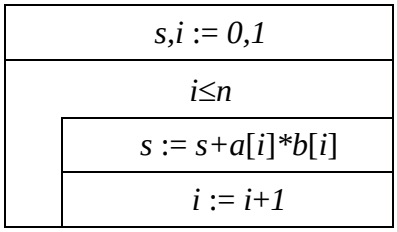
$A = (a{:}\mathbb{R}^n, b{:}\mathbb{R}^n, s{:}\mathbb{R})$

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

$$Uf = \left(Ef \wedge s = \sum_{i=1}^{n} a[i]*b[i] \right)$$

Ez a feladat nagyon hasonlít az első összegzési feladathoz (*4.1. példa*). Készítsünk megoldást visszavezetéssel! Vegyük számba két feladat közötti eltéréseket. A különbség az, hogy itt nem egy egész elemű tömb elemeinek önmagával vett szorzatait, a négyzeteit ($v[i]^2$), hanem a két valós elemű tömb megfelelő elem párjainak szorzatait ($a[i]*b[i]$) kell összegezni úgy, hogy az *i* az *1..n* tartományt futja be.

Fogjuk meg a mintaként szolgáló *4.1. példa* programját, és cseréljük le benne a $v[i]^2$ kifejezést az $a[i]*b[i]$ kifejezésre. Ezzel készen is vagyunk:



4.4. Példa. Igaz-e, hogy egy adott g:$\mathbb{Z}\to\mathbb{Z}$ függvény az egész számok egy [m..n] nem üres intervallumán mindig páros számot vesz fel értékül, azaz g(m) is páros, g(m+1) is páros, és így tovább, g(n) is páros?

Az $(g(m)\ páros) \wedge (g(m+1)\ páros) \wedge ... \wedge (g(n)\ páros)$ összetett feltételt a feladat specifikációjának utófeltételében az $\bigwedge_{i=m}^{n}(g(i)\ páros)$ zárt alakban írjuk fel. A jelölésről tudni kell, hogy ha m>n, akkor a kifejezés értéke igaz.

$A = (m{:}\mathbb{Z}, n{:}\mathbb{Z}, l{:}\mathbb{L})$

$Ef = (m=m' \wedge n=n')$

$Uf = (Ef \wedge s = \bigwedge_{i=m}^{n}(g(i)\ páros))$

A kitűzött feladat visszavezethető *4.1. példára*. A specifikációk abban különböznek, hogy itt egész számok ($v[i]^2$) összeadása helyett logikai állítások (*g(i)* páros) „összeéselésről" van szó. A logikai állítások „összeéselésének" nulla eleme a logikai igaz érték – ez az, amelyhez bármit „éselünk" is hozzá a bármit kapjuk meg. Eltér még ezen kívül az intervallum is: a korábbi *1..n* helyett most az általánosabb *m..n*. Az eredményváltozó most *s* helyett az *l*.

Tekintsük az *4.1. példa* programját, és cseréljük le benne a $v[i]^2$ kifejezést az *g(i)* páros logikai kifejezésre, a „+"-t az „∧"-re, a kezdeti értékadásban az *l* változónak a *0* helyett a logikai *igaz* értéket, az *i* változó kezdőértékének pedig az *m*-et adjuk. Táblázatos formában:

[*1..n*]	~	[*m..n*]
+, 0	~	∧, *igaz*
$s{:=}s+v[i]^2$	~	$l{:=}\ l \wedge$ (az *g(i)* páros)

l,i := igaz, m
i≤n
l:=*l* ∧ (*g(i) páros*)
i := i+1

4.5. Példa. Gyűjtsük ki egy *n* elemű (az *n* pozitív) egész számokat tartalmazó tömb páros értékeit egy halmazba!

$A = (x{:}\mathbb{Z}^n,\ h{:}\mathbf{2}^{\mathbb{Z}})$

$Ef = (\ x{=}x'\ )$

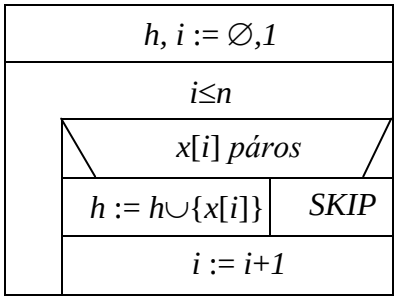
$Uf = (\ h = \bigcup_{\substack{i=1 \\ x[i]\,páros}}^{n} \{x[i]\}\ )$

A halmazok uniója is rokon művelet a számok feletti összeadással. Egységeleme az üres halmaz. A műveletben szereplő kifejezés most egy úgynevezett feltételes kifejezés: ha az $x[i]$ páros feltétel teljesül, akkor az $\{x[i]\}$ egy elemű halmazt, különben az üres halmazt „uniózzuk” a h-hoz.

Oldjuk meg <u>visszavezetéssel</u> a feladatot! Soroljuk fel az eltéréseket a *4.1. példa* mintafeladata és a most kitűzött feladat között:

$+,\ 0$	~	$\cup,\ \varnothing$
s	~	h
$s{:=}s+v[i]^2$	~	$h{:=}\ h \cup$ (ha $x[i]$ páros akkor $x[i]$ különben $\varnothing$)

Ezeket az eltéréseket átvezetve a mintaprogramra a kitűzött feladat megoldását kapjuk. Ez azonban nem megengedett, hiszen a feltételes értékadást nem tekintjük annak. A feltételes értékadás azonban nyilvánvalóan helyettesíthető egy elágazással. Ez a helyettesítés természetesen már nem a visszavezetés része, hanem egy azután alkalmazott átalakítás.



4.2. Programozási tétel fogalma

Az előző alfejezetben látott feladatoknál úgy találtuk, hogy az eltérések ellenére ezek a feladatok annyira hasonlóak, hogy ugyanazon séma alapján lehet őket megoldani. De melyek is azok a tulajdonságok, amelyekkel feltétlenül rendelkeznie kell egy feladatnak, hogy az eddig látott feladatok közé sorolhassuk? Hogyan lehetne általánosan megfogalmazni egy ilyen feladatokat?

A fenti feladatok egyik közös tulajdonsága az volt, hogy mindegyikben fontos szerepet töltött be az egész számok egy zárt intervalluma. Vagy egy *1*-től *n*-ig indexelt tömbről volt szó,

vagy egy adott $g:\mathbb{Z}\to\mathbb{Z}$ függvénynek az egész számok egy [*m..n*] intervallumán felvett értékeiről, vagy egyszerűen csak az egész számokat kellett *1*-től *n*-ig összeszorozni. Ezek az intervallumok határozták meg, hogy a feladatok megoldásában oroszlánrészt vállaló ciklus úgynevezett ciklusváltozója mettől meddig „fusson", azaz hányszor hajtódjon végre a ciklus magja.

A másik fontos tulajdonság, hogy az intervallum elemeihez tartozik egy-egy érték, amit a számításban kell felhasználni. Az első feladatban az intervallum *i* eleméhez rendelt érték a $v[i]^2$ szám, a másodikban maga az *i*, a harmadikban az *a*[*i*]**b*[*i*], a negyedikben a *g(i) páros* logikai kifejezés, az ötödikben az {*x*[*i*]} egy elemű halmaz. Mindegyik esetben megadható tehát egy leképezés, egy függvény, amely az egész számok egy intervallumán értelmezett, és az intervallum elemeihez rendel valamilyen értéket: számot, logikai értéket, esetleg halmazt.

A harmadik lényeges tulajdonság az a számítási művelet, amely segítségével az intervallum elemeihez rendelt értékeket összesíteni lehetett: összeadni, összeszorozni, „összeéselni", uniózni. Ez a művelet éppen azon értékekre értelmezett, amelyeket az intervallum elemeihez rendelünk. Mindig létezett egy olyan érték, amelyhez bármelyik másik értéket adjuk-szorozzuk-éseljük-uniózzuk hozzá, az eredmény ez a másik érték lett: $0+s=s$, $1*s=s$, $igaz\wedge l=l$, $\varnothing\cup h=h$. Úgy mondjuk, hogy mindegyik műveletnek van (baloldali) nulla eleme. A vizsgált műveletek egymás utáni alkalmazásának eredménye nem függött a végrehajtási sorrendtől, az tetszőlegesen csoportosítható, zárójelezhető (például az *a*+*b*+*c* értelmezhető *a*+(*b*+*c*)-ként és (*a*+*b*)+*c*-ként is, és mi ez utóbbit használtuk), azaz a vizsgált műveletek asszociatívak voltak.

Foglaljuk össze ezeket a közös tulajdonságokat, és fogalmazzuk meg azt az általános feladatot, amely az előző alfejezetben közölt összes feladatot magába foglalja.

Legyen adott az egész számok egy [*m..n*] intervallumán értelmezett $f:[m..n]\to H$ függvény (erről általános esetben nem kell tudni többet), amelynek *H* értékkészletén definiálunk egy műveletet. Az egyszerűség kedvéért hívjuk ezt összeadásnak, de ez nem feltétlenül a számok megszokott összeadási művelete, minthogy a *H* sem feltétlenül egy számhalmaz. A művelet lehet a számok feletti szorzás, logikai állítások feletti „éselés", halmazok feletti uniózás, stb. Ami a lényeg: a művelet legyen asszociatív és rendelkezzen (baloldali) nulla elemmel. A feladat az, hogy határozzuk meg az *f* függvény [*m..n*]-en felvett értékeinek az összegét (szorzatát, „éseltjét", unióját stb.) azaz a $\sum_{k=m}^{n} f(k) = f(m) + f(m+1) + ... + f(n)$ kifejezés értékét! ($m>n$ esetén ennek az értéke definíció szerint a nulla elem). Az előző alfejezet példái mind ráillenek erre a feladatra.

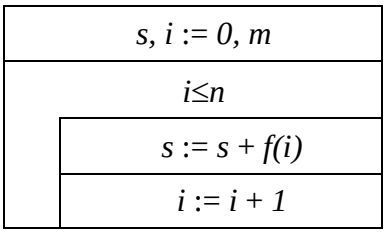
Ezt az általánosan megfogalmazott feladatot specifikálhatjuk is. Az állapottérbe a függvény értelmezési tartományának végpontjait és az eredményt vesszük fel, előfeltétele rögzíti a bemenő adatokat, utófeltétele pedig az összegzési feladatot írja le.

$A = (m{:}\mathbb{Z}, n{:}\mathbb{Z}, s{:}H)$

$Ef = (m{=}m' \wedge n{=}n')$

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

A kívánt összeget fokozatosan állítjuk elő. Ehhez végig kell vezetnünk egy *i* egész típusú változót a tömb indexein, és minden lépésben hozzá kell adni a kezdetben nullára beállított *s* változóhoz az *f(i)* értéket. A megoldó program tehát a kezdeti értékadások után (ahol az *s* változó értékét nullára, az *i* változóét *m*-re állítjuk) egy ciklus, amely az *i* változót egyesével növeli egészen *n*-ig. A ciklus működése alatt az *s* változó mindig az *f* függvény [*m..i–1*] intervallumon felvett értékeinek összegét tartalmazza, Kezdetben, amikor az *i* értéke *1*, ez az érték még nulla, befejezéskor, amikor eléri az *n+1* értéket, már a végleges eredmény.



Az most bemutatott feladat-program pár egy nevezetes minta: az úgynevezett összegzés. Ez, mint azt az előző alfejezetben láthattuk, számos feladat megoldásához szolgálhat alapul.

A visszavezetés során mintaként használt feladat-program párt (ahol a program megoldja a feladatot) ***programozási tételnek*** hívjuk. Az elnevezés onnan származik, hogy egy mintafeladat-program párt egy matematikai tételhez hasonlóan alkalmazunk: ha egy kitűzött feladat hasonlít a minta feladatára, akkor a minta programja lényegében megoldja a kitűzött feladatot is. A "hasonlít" és a "lényegében" szavak értelmét az előző példák alapján már érezzük, de majd az *5.1. alfejezetben* részletesen is kitérünk értelmezésükre.

A visszavezetés sikere azon múlik, hogy rendelkezünk-e kellő számú, változatos és eléggé általános mintafeladattal ahhoz, hogy egy új feladat ezek valamelyikére hasonlítson. Ezért a mintafeladatokat és megoldásaikat gondosan kell megválasztani. Természetesen minden programozó maga dönti el azt, hogy munkája során milyen programozási tételekre támaszkodik – ezek mennyisége és minősége a programozói tudás, a szakértelem fokmérője –, ugyanakkor meghatározható a programozási tételeknek az a közös része, amit a többség ismer és használ, amelyek feladatai a programozási feladatok megoldásánál gyakran előfordulnak.

Alig több mint fél tucat ilyen programozási tétellel a gyakorlatban előforduló problémák többsége, nyolcvan-kilencven százaléka lefedhető. A különféle programozási módszertant tanító

iskolák lényegében ugyanazokat a programozási tételeket vezetik be, legfeljebb azok megfogalmazásában, csoportosításában és felhasználási módjukban van az iskolák között eltérés. Ott, ahol a programozási tételeket nem visszavezetésre használják, mert az analóg programozás másik irányzatát, az algoritmikus gondolkodás folyamatának lemásolását követik, egy tétel megfogalmazása kevésbé precíz: annak több változata is lehetséges. A visszavezetésnél viszont a tételeket sokkal pontosabban, és lehetőleg minél általánosabban kell megadni. Egy tétel ugyanis valójában nem egy konkrét feladatot, hanem egy feladatosztályt ír le, és minél általánosabb, annál több feladat illeszkedik ehhez a feladatosztályhoz. Vigyázni kell azonban arra, hogy ha a mintafeladat túl elvont, akkor nehéz lehet annak alkalmazása.

4.3. Nevezetes minták

Most nyolc, a szakirodalomban jól ismert programozási tételt mutatunk be. Mindegyiküknél az egész számok egy intervallumán értelmezett függvény (vagy függvények) értékeinek feldolgozása a cél. Ezért ezeket a tételeket ***intervallumos programozási tételeknek*** is szoktuk nevezni.

A nyolc tétel közül egyedül a lineáris kiválasztáshoz nem kapcsolható nyilvánvaló módon az egész számok egy intervalluma. Ez egy olyan keresés, amelyik az egész számok számegyenesen folyik, de elég megadni a keresés kezdőpontját, a keresés addig tart, amíg a keresési feltétel (egy egész számokon értelmezett logikai függvény) igaz nem lesz. Erről azonban tudjuk, hogy előbb-utóbb bekövetkezik. Itt tehát a keresés során egy olyan intervallumot járunk be, amelynek a végpontja csak implicit módon van megadva.

Az intervallumokra az egyes programozási tételek különféle előfeltételeket fogalmaznak meg. Nem lehet üres az intervallum a rekurzív függvény helyettesítési értékének meghatározásakor, illetve a közönséges maximum kiválasztásnál. A maximum kiválasztásnak van egy általánosabb változata is (feltételes maximumkeresés), amely csak egy logikai feltételnek megfelelő egész számokon felvett függvényértékek között keresi a legnagyobbat, és itt az is megengedett, ha az intervallum egyetlen egy egész száma sem elégíti ki ezt a feltételt, vagy az intervallum üres. Egy logikai változó jelzi majd, hogy egyáltalán volt-e a feltételt kielégítő vizsgált függvényérték.

Az alábbi programozási tételek $f(i)$, $\beta(i)$, $h(i, y, y_1, \ldots, y_{k-1})$ kifejezései az f, β, h függvények adott argumentumú helyettesítési értékeit jelölik. Feltesszük, hogy ezek a helyettesítési értékek kiszámíthatóak.

1. Összegzés

Az összegzés programozási tételét az előző alfejezetekben alaposan elemeztük. Most csak a teljesség igénye miatt ismételjük meg.

Feladat: Adott az egész számok egy [*m..n*] intervalluma és egy *f*:[*m..n*]→*H* függvény. A *H* halmaz elemein értelmezett egy asszociatív, baloldali nulla elemmel rendelkező művelet (nevezzük összeadásnak és jelölje ezt a +). Határozzuk meg az *f* függvény [*m..n*]-en felvett értékeinek az összegét, azaz a $\sum_{k=m}^{n} f(k)$ kifejezés értékét! (*m*>*n* esetén ennek az értéke definíció szerint a nulla elem).

Specifikáció:

$A = (m{:}\mathbb{Z}, n{:}\mathbb{Z}, s{:}H)$

$Ef = (m=m' \wedge n=n')$

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

Algoritmus:

s, i := 0, m
i≤n
s := s + f(i)
i := i + 1

2. Számlálás

Gondoljunk arra a feladatra, amikor egy *20* fős osztályban azon diákok számát kell megadnunk, akik ötöst kaptak történelemből. A diákokat *1*-től *20*-ig megsorszámozzuk, és a történelem jegyeiket egy *1*-től *20*-ig indexelt *t* tömbben tároljuk: az *i*-edik diák osztályzata a $t[i]$. Definiálunk egy $\beta:[1..20]\rightarrow\mathbb{L}$ logikai függvényt, amely azokhoz a diákokhoz rendel *igaz* értéket, akik ötöst kaptak, más szóval $\beta(i) = (t[i]=5)$. Azt kell meghatároznunk, hogy a β által felvett értékek között hány *igaz* érték szerepel. Ilyenkor valójában *0* illetve *1* értékeket összegzünk attól függően, hogy a logikai kifejezés hamis vagy igaz. A számlálás tehát az összegzés speciális eseteként fogható fel, amelyben az $s:=s+1$ értékadást csak a $\beta(i)$ feltétel teljesülése esetén kell végrehajtani.

Feladat: Adott az egész számok egy $[m..n]$ intervalluma és egy $\beta:[m..n]\rightarrow\mathbb{L}$ feltétel. Határozzuk meg, hogy az $[m..n]$ intervallumon a β feltétel hányszor veszi fel az *igaz* értéket! (Ha $m>n$, akkor egyszer sem.)

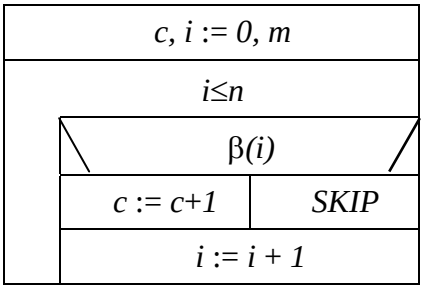
Specifikáció:

$A = (m:\mathbb{Z}, n:\mathbb{Z}, c:\mathbb{N})$

$Ef = (m=m' \wedge n=n')$

$$Uf = \left(Ef \wedge c = \sum_{\substack{i=m \\ \beta(i)}}^{n} 1 \right)$$

Algoritmus:



3. Maximum kiválasztás

Ha arra vagyunk kíváncsiak, hogy egy *20* fős osztályban kinek van a legjobb jegye történelemből, akkor ehhez a diákokat *1*-től *20*-ig megsorszámozzuk, és a történelem jegyeiket egy *1*-től *20*-ig indexelt *t* tömbben tároljuk: az *i*-edik diák osztályzata a $t[i]$. Ez a tömb tekinthető egy $f:[1..20]\to\mathbb{N}$ függvénynek ($f(i)=t[i]$), amely értékei között keressük a legnagyobbat.

Feladat: Adott az egész számok egy $[m..n]$ intervalluma és egy $f:[m..n]\to H$ függvény. A H halmaz elemein értelmezett egy teljes rendezési reláció. Határozzuk meg, hogy az f függvény hol veszi fel az $[m..n]$ nem üres intervallumon a legnagyobb értéket, és mondjuk meg, mekkora ez a maximális érték!

Specifikáció:

$A = (m:\mathbb{Z}, n:\mathbb{Z}, ind:\mathbb{Z}, max:H)$

$Ef = (m=m' \wedge n=n' \wedge n\geq m)$

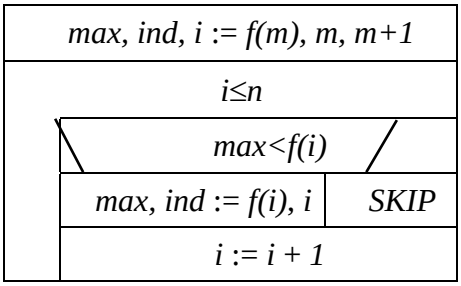
$Uf = (Ef \wedge ind\in[m..n] \wedge max=f(ind) \wedge \forall i\in[m..n]: max\geq f(i)) =$

másképpen jelölve, illetve rövidített jelölést használva:

$= (Ef \wedge ind\in[m..n] \wedge max = f(ind) = \max_{i=m}^{n} f(i))$

$= (Ef \wedge max,ind= \max_{i=m}^{n} f(i))$

Algoritmus:



Az *ind* változó értékadásai törölhetők az algoritmusból, ha nincs szükség az *ind*-re.

4. Feltételes maximumkeresés

A maximum kiválasztást sokszor úgy kell elvégezni, hogy a vizsgált elemek közül csak azokat vesszük figyelembe, amelyek eleget tesznek egy adott feltételnek. Például, egymás utáni napi átlaghőmérsékletek között azt a legmelegebb hőmérsékletet keressük, amelyik fagypont alatti.

Feladat: Adott az egész számok egy [*m..n*] intervalluma, egy f:[*m..n*]$\rightarrow H$ függvény és egy β:[*m..n*]$\rightarrow \mathbb{L}$ feltétel. A H halmaz elemein értelmezett egy teljes rendezési reláció. Határozzuk meg, hogy az [*m..n*] intervallum β feltételt kielégítő elemei közül az f függvény hol veszi fel a legnagyobb értéket, és mondjuk meg, mekkora ez az érték! (Lehet, hogy egyáltalán nincs β feltételt kielégítő elem az [*m..n*] intervallumban vagy $m>n$.)

Specifikáció:

$A = (m{:}\mathbb{Z}, n{:}\mathbb{Z}, l{:}\mathbb{L}, ind{:}\mathbb{Z}, max{:}H)$

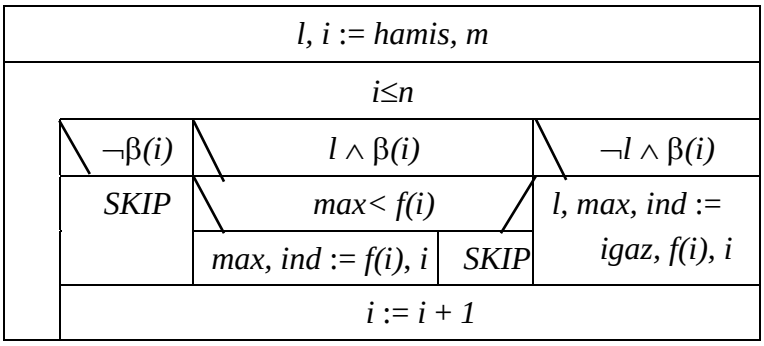
$Ef = (m=m' \wedge n=n')$

$Uf = (Ef \wedge l = \exists i \in [m..n]{:}\beta(i) \wedge l \rightarrow ind \in [m..n] \wedge max=f(ind)$
$\wedge \beta(ind) \wedge \forall i \in [m..n]{:}\beta(i) \rightarrow max \geq f(i))$

Itt is bevezetünk rövid jelölést.

$$Uf = (Ef \wedge l,max,ind) = \max_{\substack{i=m \\ \beta(i)}}^{n} f(i))$$

Algoritmus:



5. Kiválasztás (szekvenciális vagy lineáris kiválasztás)

Ennek a mintának kapcsán azokra a feladatokra gondoljunk, ahol egymás után sorba állított elemek között keressük az első adott tulajdonságút úgy, hogy tudjuk, hogy ilyen elem biztosan van. Például egy osztályban keressük az első ötös történelem jeggyel rendelkező diákot, ha tudjuk, hogy ilyen biztosan van. A diákok történelem jegyeit ilyenkor egy *1*-től *20*-ig indexelt *t* tömbben tároljuk (az *i*-edik diák osztályzata a *t*[*i*]), ha a diákokat *1*-től *20*-ig sorszámoztuk meg. A keresés szempontjából közömbös, hogy egy *20* fős osztállyal van dolgunk. Definiálunk egy $\beta:[1..20]\rightarrow\mathbb{L}$ logikai függvényt, amely azokhoz a diákokhoz rendel *igaz* értéket, akik ötöst kaptak, más szóval $\beta(i) = (t[i]=5)$.

Feladat: Adott egy *m* egész szám és egy *m*-től jobbra értelmezett $\beta:\mathbb{Z}\rightarrow\mathbb{L}$ feltétel. Határozzuk meg, hogy az *m*-től jobbra eső első olyan számot, amely kielégíti a β feltételt, ha tudjuk, hogy ilyen szám biztosan van!

Specifikáció:

$A = (m:\mathbb{Z}, ind:\mathbb{Z})$

$Ef = (m=m' \wedge \exists i \geq m: \beta(i))$

$Uf = (Ef \wedge ind \geq m \wedge \beta(ind) \wedge \forall i \in [m..ind-1]:\neg\beta(i))$

vagy rövidített jelölést alkalmazva:

$Uf = (Ef \wedge ind = \underset{i \geq m}{\textbf{select}}\ \beta(i))$

Algoritmus:

ind := *m*	
¬*β*(*ind*)	
	ind := *ind* + *1*

Az előző tételekkel szemben itt nincs szükség a bejárt száok intervallumának felső határára, mintahogy egy külön *i* változóra sem a számok bejárásához.

6. Keresés (szekvenciális vagy lineáris keresés)

Ez a minta hasonlít az előzőre, hiszen itt is egymás után sorba állított elemek között keressük az első adott tulajdonságút, de lehet, hogy ilyet egyáltalán nem találunk.

Feladat: Adott az egész számok egy [*m..n*] intervalluma és egy β:[*m..n*]$\rightarrow\mathbb{L}$ feltétel. Határozzuk meg az [*m..n*] intervallumban balról az első olyan számot, amely kielégíti a β feltételt!

Specifikáció:

$A = (m{:}\mathbb{Z}, n{:}\mathbb{Z}, l{:}\mathbb{L}, ind{:}\mathbb{Z})$

$Ef = (m=m' \wedge n=n')$

$Uf = (Ef \wedge l=(\exists i\in[m..n]{:}\ \beta(i)) \wedge l\rightarrow(ind\in[m..n] \wedge \beta(ind) \wedge \forall i\in[m..ind-1]{:}\neg\beta(i)))$

vagy rövidített jelölést alkalmazva

$Uf = (Ef \wedge l, ind = \underset{i=m}{\overset{n}{\textbf{search}}}\ \beta(i))$

Algoritmus[11]:

l, i := hamis, m	
$\neg l \wedge i\leq n$	
	l, ind := β*(i), i*
	i := i+1

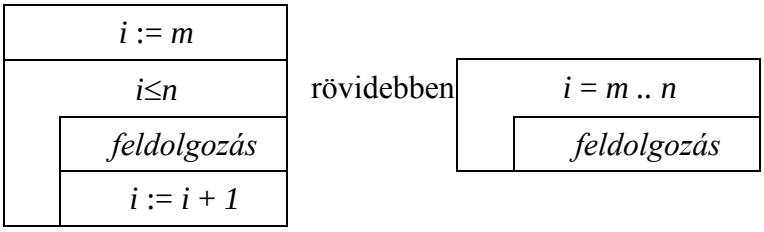
[11] A lineáris keresés algoritmusának más változatai is ismertek:

l, ind := hamis, m–1	
$\neg l \wedge ind<n$	
	ind := ind+1
	l := β*(ind)*

l, ind := hamis, m	
$\neg l \wedge ind\leq n$	
	β*(ind)*
l:=*igaz*	*ind := ind+1*

ind := m	
$ind\leq n \wedge \neg\beta(ind)$	
	ind := ind+1
l := ind$\leq$*n*	

A kiválasztás, lineáris keresés és a logaritmikus keresés kivételével a fenti algoritmus minták ciklusszervezése megegyezik: egy indexváltozót vezetnek végig az *m..n* intervallumon. Az ilyen esetekben alkalmazhatunk egy rövidített jelölést az algoritmus leírására. Ezt a változatot szokták számlálós ciklusnak nevezni. Az elnevezés kicsit félrevezető, hiszen, mint látjuk, ez egy kezdeti értékadásnak és egy ciklusnak a szekvenciája. A számlálós ciklus alkalmazhatóságának jelentőségét egyrészt az adja, hogy ilyenkor pontosan meg lehet mondani, hány iteráció után fog leállni a ciklus, másrészt a különféle programozási nyelvek külön nyelvi elemet (például „for”) szoktak biztosítani a kódolásukhoz.



A programozási tételek helyességét a *3. fejezetben* bemutatott eszközökkel be lehet bizonyítani.

5. Visszavezetés

Ebben az alfejezetben példákat látunk a visszavezetésre, nevezetesen arra, amikor a programozási tételek mintájára oldjuk meg a feladatainkat, és közben azt próbáljuk megvilágítani, hogy a visszavezetéssel előállított programok miért lesznek biztosan megoldásai a kitűzött feladatoknak. Az alább bemutatott esetek egymással kombinálva is megjelenhetnek a visszavezetések során.

Természetes visszavezetésről akkor beszélünk, amikor az alkalmazott programozási tétel feladata (a mintafeladat) átnevezésektől eltekintve teljesen megegyezik a megoldandó (a kitűzött) feladattal.

5.1. Példa. Hányszor vált előjelet (pozitívról negatívra vagy negatívról pozitívra) az [a..b] egész intervallumon az $f:\mathbb{Z}\to\mathbb{R}$ függvény?

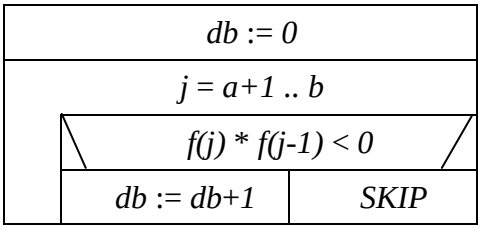
$A = (a:\mathbb{Z}, b:\mathbb{Z}, db:\mathbb{N})$

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

$$Uf = (Ef \wedge db = \sum_{\substack{j=a+1 \\ f(j)*f(j-1)<0}}^{b} 1)$$

Ez a specifikáció olyan, mint a számlálás programozási tételének specifikációja. A különbség csak annyi, hogy az ott szereplő β feltételt itt nem az [*m..n*], hanem az [*a+1..b*] intervallumon értelmezzük; az intervallumon nem az *i*, hanem a *j* változóval futunk végig; az *s* változót a *db* változó helyettesíti; és a β*(i)* helyén az *f(j)*f(j-1)<0* kifejezés áll. Alkalmazzuk ezt a megfeleltetést a számlálás programjára és megkapjuk a kitűzött feladat megoldását.

m..n ~ *a+1..b*
s ~ *db*
i ~ *j*
β(i) ~ *f(j)*f(j-1)<0*



Általános visszavezetésről akkor beszélünk, amikor a kitűzött feladat megengedőbb a mintafeladatánál. Ez kétféleképpen is előállhat. Egyfelől akkor, ha a kitűzött feladat kevesebb kezdőállapotra van értelmezve, mint a mintafeladat (azaz a kitűzött feladat előfeltétele szigorúbb a mintafeladaténál). Másfelől a kitűzött feladat egy kezdőállapothoz olyan célállapotot is rendelhet, amelyet a mintafeladat nem (azaz a feladat utófeltétele gyengébb a mintafeladaténál).

5.2. Példa. Keressük egy $f:[m..n]\to\mathbb{R}$ függvénynek azt az arumentumát, amelyre teljesül, hogy $f(k)=(f(k-1)+ f(k+1))/2$.

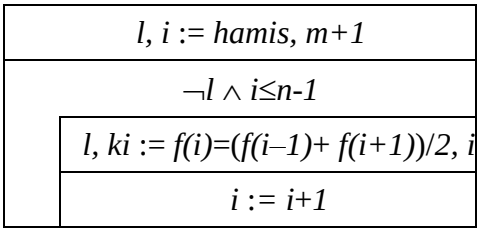
$$A = (m:\mathbb{Z},\ n:\mathbb{Z},\ l:\mathbb{L},\ ki:\mathbb{Z})$$
$$Ef = (\ m=m' \wedge n=n'\ )$$
$$Uf = (\ Ef \wedge l=(\ \exists i\in[m+1..n-1]:\ f(i)=(f(i-1)+f(i+1))/2\ ) \wedge$$
$$l\to(\ ki\in[m+1..n-1] \wedge f(ki)=(f(ki-1)+f(ki+1))/2\ )\ )$$

Szigorítsunk a feladaton. Ha a szigorúbb feladathoz találunk megoldást, akkor ez az eredeti feladatot is megoldja.

$$Uf = (\ Ef \wedge l=(\ \exists i\in[m+1..n-1]:\ f(i)=(f(i-1)+f(i+1))/2\ ) \wedge$$
$$l\to(\ ki\in[m+1..n-1] \wedge f(ki)=(f(ki-1)+f(ki+1))/2 \wedge$$
$$\forall i\in[m+1..ki-1]:\ f(i)\neq(f(i-1)+f(i+1))/2\ )\ )$$
$$= (\ Ef \wedge\ l,ki = \underset{i=m+1}{\overset{n-1}{\textbf{search}}}(f(i) = (f(i-1) + f(i+1))/2)\ )$$

Ez a feladat visszavezethető a lineáris keresés programozási tételére.

$m..n$ ~ $m+1..n-1$
$\beta(i)$ ~ $f(i)=(f(i-1)+ f(i+1))/2$
ind ~ ki



Alteres visszavezetésről akkor beszélünk, amikor a kitűzött feladat állapottere a mintafeladat állapotterének nem minden komponensét tartalmazza (altere a kitűzött feladat állapotterének), azaz a kitűzött feladat állapotterének kevesebb komponenssel bír.

5.3. Példa. Keressük egy g:[$a..b$]→ℝ függvénynek olyan argumentumát, ahol a függvény a maximális értékét veszi fel!.

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

$Ef = (\ a=a' \wedge b=b' \wedge a \leq b\ )$

$Uf = (\ Ef \wedge ind \in [a..b] \wedge g(ind) = \max_{i=a}^{b} g(i)\ )$

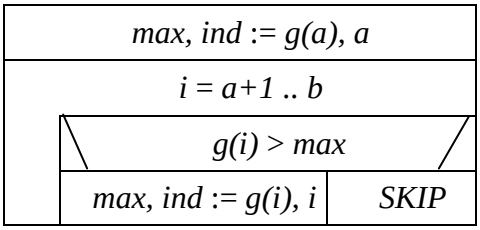
A kitűzött feladat állapottere kiegészíthető a max:ℝ komponenssel, az utófeltétele pedig szigorítható

$$Uf = (\ Ef \wedge max = \max_{i=a}^{b} g(i) \wedge ind \in [a..b] \wedge g(ind) = max\ )$$

és ez a szigorúbb feladat visszavezethető a maximum kiválasztás programozási tételére.

$m..n \sim a..b$

$f(i) \sim g(i)$



A max változó nem eredményváltozója, hanem segédváltozója a fenti programnak. Megfelelő átalakítás után el is lehetne hagyni, de ez egyrészt a program hatékonyságán ronthat, másrészt egy ilyen módosítás veszélyezteti a program helyességét. Kivételt jelent az, amikor a fordított feladatot kell megoldanunk: a maximális értéket keressük és annak indexére nem vagyunk kíváncsiak. Ilyenkor az index változót (ind), pontosabban az arra vonatkozó értékadásokat el hagyhatjuk, hiszen ezen kívül más átalakítást nem kell a programon végezni.

5.4. Példa. Adjuk meg egy természetes szám valódi osztóinak számát!

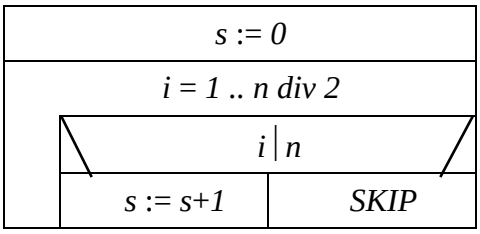
$A = (n{:}\mathbb{N}, s{:}\mathbb{N})$
$Ef = (n=n')$
$Uf = (Ef \wedge s = \sum_{\substack{i=2 \\ i \mid n}}^{n div 2} 1)$

Átalakíthatjuk ezt a specifikációt az alábbi formára:

$A = (m{:}\mathbb{N}, n{:}\mathbb{N}, s{:}\mathbb{N})$
$Ef = (m=2 \wedge n=n')$
$Uf = (Ef \wedge s = \sum_{\substack{i=m \\ i \mid n}}^{n div 2} 1)$

Itt már ugyanolyan állapotterű feladatról van szó, mint a számlálás mintafeladata, csak a feladat előfeltétele szigorúbb, de ezt az általános visszavezetés megengedi.

$m..n \sim 1..n \; div \; 2$
$\beta(i) \sim i \mid n$



Találkozhatunk olyan visszavezetésekkel is, ahol a megoldandó feladat állapottere a visszavezetésre kiválasztott programozási tétel mintafeladatánál nem szereplő, olyan bemeneti adatkomponenseket is tartalmaz, amelyeknek értéke állandó. (Ezen túlmenően a természetes visszavezetésnél megengedett eltérések is előfordulhatnak.)

5.5. Példa. Határozzuk meg a g:$[a..b]\rightarrow\mathbb{N}$ legnagyobb, k-val osztható értékét és az $[a..b]$ intervallumnak azt az elemét (argumentumát), ahol a g függvény ezt az értéket felveszi!

$A = (a{:}\mathbb{Z}, b{:}\mathbb{Z}, k{:}\mathbb{N}, l{:}\mathbb{L}, ind{:}\mathbb{Z}, max{:}\mathbb{Z})$

$Ef = (a=a' \wedge b=b' \wedge k=k')$

$$Uf = (Ef \wedge l,max,ind = \mathop{\mathbf{max}}_{\substack{i=a \\ k \mid g(i)}}^{b} g(i))$$

Ebben a feladatban a k változó értéke állandó. (Nem konstans, hiszen ez egy bemenő adat, de a kezdeti értéke már nem változik.) Ha ennek egy értékét rögzítjük, akkor a feladatnak egy speciális, a rögzített értékhez tartozó, azzal paraméterezett változatához jutunk. Ebben a változatban a paraméter már egy konstans, amelyet nem kell felvennünk az állapottér komponensei közé, így a paraméterezett feladat állapottere már meg fog egyezni a visszavezetésre kiválasztott mintafeladatéval. Például $k=2$ esetben:

$A = (a{:}\mathbb{Z}, b{:}\mathbb{Z}, l{:}\mathbb{L}, ind{:}\mathbb{Z}, max{:}\mathbb{Z})$

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

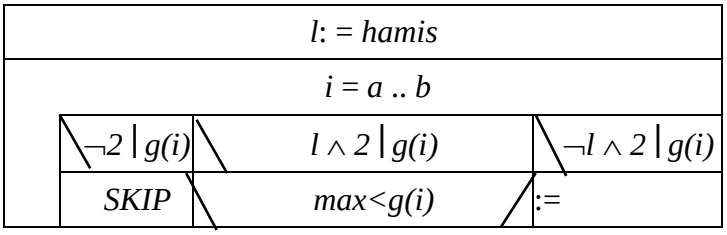
$$Uf = (Ef \wedge l,max,ind = \mathop{\mathbf{max}}_{\substack{i=a \\ 2 \mid g(i)}}^{b} g(i))$$

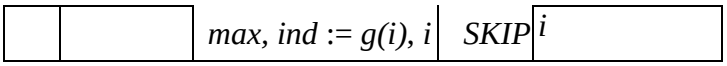
Ez a paraméterezett feladat már minden gond nélkül visszavezethető a feltételes maximumkeresés programozási tételére.

$m..n \sim a..b$

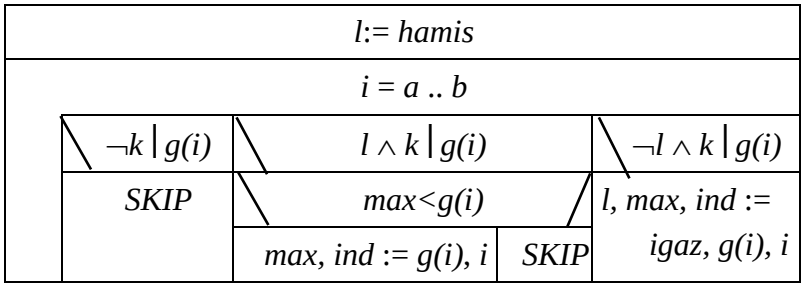
$f(i) \sim g(i)$

$\beta(i) \sim 2 \mid g(i)$





Mivel a *k* minden lehetséges értéke mellett ez a visszavezetés megtehető, ezért az eredeti feladatot is meg tudjuk oldani úgy, hogy összes paraméteres változatát megoldjuk. Természetesen ezt elég általánosan egy tetszőlegesen rögzített *k* értékkel paraméterezett feladatra megtenni.



Ebben a *k* egy olyan változó, amelyik a program legelején kap egy kezdőértéket, azaz a megoldó program alap-állapotterének komponense lesz.

Bizonyos feladatok visszavezetéssel történő megoldásának érdekében úgy kell eljárnunk, hogy vagy a választott programozási tételt általánosítjuk, vagy a feladatot megfogalmazásán alakítunk. Nézzünk erre példákat.

Azokat a feladatokat, amelyekben a "legtöbb", "legnagyobb", "legdrágább", "legtávolabbi" stb. szófordulatokkal találkozunk, a maximum kiválasztással társítjuk. A maximum kiválasztással társítható feladatok megítélésénél körültekintően kell eljárni. Egyfelől azért, mert hasonló szófordulatok jelezhetik a feltételes maximumkeresést is, ahol nem egy intervallumhoz hozzárendelt összes elem között, hanem csak bizonyos tulajdonságú elemek között keressük a maximumot. Másfelől a maximum kiválasztásnál ügyelni kell az előfeltételre, amely szerint a maximális elem kiválasztásához léteznie kell legalább egy elemnek, azaz az intervallum nem lehet üres. Ilyenkor vagy egy elágazásba építjük be a maximum kiválasztást azért, hogy csak nem-üres intervallum esetén kerüljön sor a maximum kiválasztásra, vagy feltételes maximumkeresést használunk.[12]

[12] Itt hívjuk fel a figyelmet az olyan feladatokra is, amelyekben egy intervallum pontjai között keressük a legnagyobb adott tulajdonságút. Ilyen például a legnagyobb közös osztó keresése. Ezeket a feladatokat nem érdemes feltételes maximumkeresésre visszavezetni, tökéletesen megfelel az intervallumban egy fordított irányú kiválasztás vagy lineáris keresés is.

Gondoljunk most arra a feladatra, amelynél egy adott f:[$m..n$]→H függvény értékei között a legkisebb elemet keressük. A "legkevesebb", "legnagyobb", "legolcsóbb", "legközelebb" melléknevek ***minimum kiválasztásra*** utalnak. A kérdés az, hogyan lehet egy minimum kiválasztásos feladatot a maximum kiválasztás programozási tételére visszavezetni. Azt mindenki érzi, hogy minimum kiválasztás programja mindössze annyiban tér el a maximum kiválasztásétól, hogy másik relációt használunk az elágazásának feltételében: A *max<f(i)* helyett a *max>f(i)*-t. (Célszerű lesz a *max* változó nevét *min*-re cserélni.)

A lineáris keresést az eldöntés jellegű feladatok megoldására is fel lehet használni (alteres visszavezetés). Ilyenkor csak azt vizsgáljuk, hogy a vizsgált feltétel bekövetkezik-e az intervallum valamelyik pontján. Mivel nem vagyunk kíváncsiak arra, hogy melyik ez a pont, az algoritmusból az *ind*:=*i* értékadást el is hagyhatjuk.

Sokszor keressük arra a kérdésre a választ, hogy vajon az intervallum minden eleme rendelkezik-e egy adott tulajdonsággal (l=∀i∈[$m..n$]:β*(i)*). Ezt az eldöntéses feladatot is vissza lehet vezetni a lineáris keresésre. Ha ezt egy olyan lineáris kereséssel oldjuk meg, amely azt vizsgálja, hogy van-e olyan elem, amelyik nem rendelkezik az adott tulajdonsággal (u=∃i∈[$m..n$]:¬β*(i)*), akkor a kapott válasz alapján az eredeti kérdés is könnyen megválaszolható (l=¬u). Másik megoldáshoz jutunk az alábbi gondolatmenettel. Tekintsük a megoldandó feladat utófeltételét, majd ezt alakítsuk át vele ekvivalens, de a lineáris keresésre visszavezethető formára:

$$
\begin{aligned}
Uf &= (Ef \wedge \; l = \forall k \in [m..n]: \beta(k)) = \\
&= (Ef \wedge \neg l = \neg(\forall k \in [m..n]: \beta(k))) = \\
&= (Ef \wedge \neg l = \exists k \in [m..n]: \neg\beta(k)) = \\
&= (Ef \wedge \; l = \neg \overset{n}{\underset{i=m}{\textbf{\textit{search}}}} \neg\beta(i))
\end{aligned}
$$

	¬l,i := *hamis,m*
	¬¬l ∧ i≤n
	¬l:= ¬β*(i)*
	i := i+1

A program ¬l:=hamis és ¬l:=¬β*(i)* pszeudo-értékadásait átalakítva és a ciklusfeltételt egyszerűsítve megkapjuk a végleges változatot, amelyet akár új programozási tételként is – tagadott vagy „∀-jeles” vagy ***optimista lineáris keresés*** néven – jegyezhetünk meg. Ezt

legkifejezőbben az $l = (\forall k \in [m..n]: \beta(k)))$ specifikációs formula fejezi ki, de bevezethetjük rá az $l = \forall \overset{n}{\underset{i=m}{\textbf{\textit{search}}}} \beta(i)$ jelölést is.

	l,i:= *igaz,m*
	$l \wedge i \leq n$
	$l := \beta(i)$
	$i := i+1$

6. Többszörös visszavezetés

A visszavezetés technikájának megértése, elsajátítása önmagában könnyű feladat. Alkalmazása akkor válik nehézzé, amikor egy összetett probléma megoldásához egyszerre több programozási tételt is fel kell használni. Már egy kezdő programozónál is kialakul az a képesség, hogy egy összetett feladatban észrevegye a megoldáshoz szükséges programozási tételeket, de ezek egymáshoz való viszonyát, az egyiknek a másikba való beágyazásának módját még nem látja tisztán. Ehhez arra van szükség, hogy a megoldandó feladatot részekre tudjuk bontani úgy, hogy a részfeladatokat a programozási tételek segítségével meg lehessen oldani, és az így kapott részmegoldásokat kell egy programmá összeépíteni.

A részfeladatok kijelölését hatékonyan támogatja, ha a feladat megfogalmazásakor az egyes fogalmak jelölésére egy-egy alkalmas függvényt vezetünk be. Ezek a kitalált, tehát absztrakt függvények a megoldás bizonyos fázisaiban elrejtik az általuk definiált részfeladatokat. A beágyazott részfeladatok elrejtése következtében világosabban látható, hogy magasabb szinten milyen programozási tétellel oldható meg a feladat. Csak ezt követően kell foglalkozni a részfeladatokkal, amelyek megoldását a feladat többi részétől függetlenül, a részfeladatot elrejtő függvény kiszámításával állíthatjuk elő.

Ezt a függvényabsztrakciós programozási technikát procedurális (modularizált) programkészítésnek is nevezik, mert az így készült programok kódolásakor elkülönítjük a részfeladatokat megoldó programokat, ezáltal a teljes program részprogramokra bomlik. Mivel a különválasztott programrészek (procedurák, modulok) jól átlátható, ellenőrzött kapcsolatban állnak, ezért a részprogramok végrehajtási sorrendje pontosan ki van jelölve. Amikor egy részprogramról átkerül a végrehajtási vezérlés egy másikra, akkor az első adatokat adhat át a másodiknak. Amennyiben egy részprogram leírását fizikailag is elválasztjuk a teljes programról, akkor azt ***alprogramnak*** hívjuk. Egy alprogram maga is több alprogramra bontható, ami által összetett, hierarchikus szerkezete lesz a programunknak.

Ebben a fejezetben először bevezetjük az absztrakt programok szintjén az alprogramok fogalmát, majd néhány példán keresztül bemutatjuk, hogyan alkalmazható többszörösen a visszavezetés technikája az összetett feladatok megoldásánál, és az így kapott programot a feladat részfeladatai mentén hogyan tördeljük alprogramokra. A harmadik alfejezetben áttekintjük azokat a program-átalakító szabályokat, amelyek alkalmazása kényelmesebbé teszi a programkészítést, hatékonyabbá az elkészített programot.

6.1. Alprogram

Egy program leírásában szereplő bármelyik részprogram egy jól meghatározott részfeladatot old meg. Ha egy részprogram fizikailag is elkülönül a program leírásában, akkor azt alprogramnak nevezzük. Struktogrammos jelölés esetén egy részprogramból úgy készíthetünk alprogramot, hogy a részprogramot kiemeljük az azt tartalmazó program struktogrammjából, a kiemelt struktogrammot egyedi azonosítóval látjuk el, és a részprogram helyét a program struktogrammjában ezzel az azonosítóval jelöljük. Ez lehetőséget ad majd arra is, hogy ugyanarra az alprogramra – ha kell – több helyen is hivatkozhassunk részprogramként.

Mivel minden feladat megoldható egy (általában nem megengedett) értékadással, ezért az alprogramjaink által megoldott részfeladatok is, ennél fogva minden alprogram azonosítható ezzel az értékadással. Ezért a továbbiakban az alprogramot leíró struktogrammot ezzel az értékadással azonosítjuk. Ez az ***alprogram feje***. Egy program struktogrammjában, ahol az alprogramot részprogramként látni kívánjuk, ugyanezt az értékadást írjuk. Ezt az alprogramot meghívó utasításnak, röviden ***hívásának*** nevezik.

Amikor egy program végrehajtása során elérünk egy alprogram hívásához, akkor onnan kezdve az alprogram határozza meg a végrehajtás menetét, azaz „átkerül a vezérlés" az alprogramhoz. Az alprogram befejeződésekor a vezérlés visszatér a hívó programhoz és a hívó utasítás után folytatódik. Más szavakkal, a hívó programnak (amely a hívást tartalmazza) a hívás pillanatáig befutott végrehajtása (állapotsorozata) felfüggesztődik, az ekkor elért állapotból az alprogram egy végrehajtásával (állapotsorozatával) folytatódik, majd az így elért állapotból a hívó programnak a hívás utáni utasításai alapján folytatódik végrehajtás.

Az alprogram kiinduló állapottere tartalmazza a hívó program állapotterének a hívás pillanatában (segédváltozókkal együtt) meglevő komponenseit. Más szavakkal az alprogram használhatja a hívó program azon változóit, amelyek az alprogram hívásakor élnek. Ezeket az alprogram szempontjából ***globális változóknak*** nevezzük. Az alprogram – mint minden program – bevezethet új változókat is, de ezek az alprogram befejeződésekor megszűnnek. Ezeket nevezzük az alprogram ***lokális változóinak***. Egy lokális változó neve megegyezhet egy globális változó nevével, de ekkor gondoskodni kell arról, hogy a két ugyanolyan nevű, de természetesen egymástól független változót megkülönböztessük valahogyan egymástól. Mi erre nem vezetünk be külön jelölést, de megállapodunk abban, hogy az ilyen esetekben, amikor egy változó neve nem azonosítja egyértelműen a változót, akkor azon mindig a lokális változót értjük.

Természetesen az alprogramnak nem kell minden globális változót használnia. Ennél fogva egy alprogram más, eltérő állapotterű programokból is meghívható, feltéve, hogy azok állapotterében a hívás pillanatában vannak olyan változók, amelyeket az alprogram globális változóként használni akar. A legrugalmasabb alprogramok ebből a szempontból azok, amelyek egyáltalán nem használnak globális változókat, mert ezeket bármelyik programból meghívhatjuk.

De hogyan tud ebben az esetben az alprogram a hívó program változóiban tárolt értékekhez hozzáférni, és hogyan tud azoknak új értékeket adni?

Az alprogramot meghívó utasításnak és az alprogram fejének nem kell betű szerint megegyeznie. Az alprogram fejeként használt értékadás különbözhet a hívó utasításként szerepeltetett értékadástól abban, hogy egyrészt az értékadás baloldalán álló változók neve eltérhet (típusaik és sorrendjük azonban nem), másrészt az értékadás jobboldalán levő részkifejezések helyén azokkal azonos típusú változók állhatnak. Ezek az alprogram ***paraméterváltozói***. A kizárólag az értékadás jobboldali kifejezésében lévő paraméterváltozókat szokták a ***bemenő-változóknak***, az értékadás jelétől balra levőket ***eredmény-változóknak*** nevezni. Egy paraméterváltozó lehet egyszerre bemenő és eredmény-változó is. Ebben az esetben a hívóutasításban a változó helyén csak egy változó állhat, konstans vagy kifejezés nem. A fej paraméterváltozóinak balról jobbra felsorolását (az esetleges ismétlődések megszűntetése után) ***formális paraméterlistának*** szokták hívni. A hívó utasításban a paraméterváltozóknak megfeleltetett változók és kifejezések (ismétlődés nélküli) felsorolását ***aktuális paraméterlistának,*** elemeit ***paramétereknek*** nevezzük. A formális és aktuális paraméter lista elemszáma és elemeinek típusa azonos kell hogy legyen.

A paraméterváltozók az alprogramban jönnek létre és az alprogram befejeződésekor szűnnek meg, tehát lokális változói az alprogramnak, de különleges kapcsolatban állnak az alprogram hívásával. Az alprogram hívásakor ugyanis a bemenő paraméterváltozók megkapják a hívó utasításban a helyükön álló kifejezések (speciális esetben változók) értékét kezdőértékként. Az alprogram befejeződésekor pedig az eredmény-változók értékei átmásolódnak a hívó értékadás baloldalán álló megfelelő változókba. (A bemenő változók kivételével az alprogram többi lokális változójának kiinduló értéke nem-definiált.)

Egy alprogram hívására úgy is sor kerülhet, hogy a hívó utasításnak csak a jobboldalát tüntetjük fel, de ez olyan környezetben jelenik meg, amely képes elkapni, feldolgozni az alprogram eredmény-változói által visszaadott értékeket. Hogy egy egyszerű példával megvilágítsuk ezt, képzeljünk el egy olyan alprogramot, amelyet az *l*:=*felt*(*i*) értékadás azonosít, ahol *l* egy logikai, az *i* pedig egy egész értékű változó. Ez az alprogram meghívható például egy *u*:=*felt*(5) értékadással (ahol *u* egy logikai változó), vagy egy olyan elágazással, ahol az elágazás feltétele *felt*(5), azaz a feltétel logikai értékét az alprogram eredménye adja, a *felt*(*i*). Az első esetben a hívás a teljes értékadás megadásával történt, a második esetben csak egy kifejezéssel. Valójában csak formai szempontból van különbség a kétféle hívás között: az első esetben ***hívó utasítással*** történő ***eljárásszerű hívásról***, a másodikban ***hívó kifejezéssel*** kiváltott ***függvényszerű hívásról*** beszélünk.[13] A hívó kifejezés lehet a hívó programnak önálló kifejezése (például egy elágazás vagy ciklus feltétele, vagy akár egy értékadás jobboldala) vagy egy kifejezés része. A hívó kifejezés értéke több eredmény-változó esetén több komponensű. Eljárásszerű hívás esetén

[13] A függvényszerű hívással elindított alprogramot a konkrét programozási nyelvek *függvénynek,* az önálló utasítással (értékadással) meghívott alprogramot *eljárásnak* nevezik.

az aktuális (a hívásban szereplő) eredmény-változók kapják meg a formális eredmény-változók értékét. Függvényszerű híváskor az alprogram eredmény-változóinak értéke a hívó kifejezés értékeként jelenik meg a hívó programban.

6.2. Beágyazott visszavezetés

Ebben az alfejezetben a többszörös visszavezetéssel megoldható feladatokkal foglalkozunk, pontosabban olyanokkal, amelyek megoldásánál egy programozási tételt egy másikba kell beágyazni. Ilyenkor a programnak azt a részét, amelyet a beágyazott tételre vezettünk vissza, gyakran alprogramként szerepeltetjük.

6.1. Példa. Egy iskola egyik *n* diákot számláló osztályában *m* különböző tantárgyból osztályoztak a félév végén. A jegyek egy táblázat formájában rendelkezésünkre állnak. (A diákokat és a tantárgyakat most sorszámukkal azonosítjuk.) Állapítsuk meg, van-e olyan diák, akinek csupa ötöse van!

A feladat egy "van-e olyan az elemek között, hogy ..." típusú eldöntésből áll, ami alapján sejthető, hogy a lineáris keresés programozási tételére lehet majd visszavezetni. Ebben a keresésben a diákokat kell egymás után megvizsgálni, hogy csupa ötösük van-e. Egy diák megvizsgálása is egy eldöntéses feladat, csakhogy itt arra keressük a választ, hogy a diáknak minden jegye ötös-e. Az ilyen "minden elem olyan-e, hogy ..." típusú eldöntéseknél az optimista lineáris kereséssel adhatjuk meg a választ. A feladat megoldása tehát elsősorban egy lineáris keresés lesz, amelyik magába foglalja a "csupa ötös" tulajdonságot eldöntő optimista lineáris keresést. Nézzük meg ezt alaposabban!

A feladat bemenő adata az osztályzatok táblázata: egy *n* sorú és *m* oszlopú mátrix, amelynek elemei *1* és *5* közé eső egész számok. Kimenő adata egy logikai érték, amely igaz, ha van kitűnő tanuló, hamis különben.

$$A = (t{:}\mathbb{N}^{n\times m}\ ,\ l{:}\mathbb{L}\ )$$
$$Ef = (\ t{=}t'\ )$$

Az utófeltétel megfogalmazásához bevezetünk egy függvényt, amelyik a *t* táblázat *i*-edik sora alapján megmondja, hogy az *i*-edik diáknak csupa ötöse van-e. Ez a *színjeles* függvény egy intervallumon értelmezett logikai függvény, amely akkor ad igazat az *i*-edik értékre, ha a táblázat *i*-edik sorának minden eleme ötös.

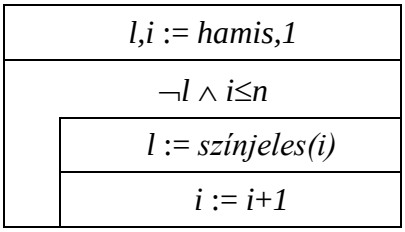
$$színjeles : [1..n] \rightarrow \mathbb{L}$$
$$színjeles(i) = \forall j \in [1..m]: t[i,j]=5$$

Ennek segítségével könnyen felírható a feladat utófeltétele: az *l* pontosan akkor igaz, ha van olyan diák, azaz *1* és *n* közé eső egész szám, amelyre a *színjeles* feltétel igazat ad.

$Uf = (Ef \wedge l = \exists i \in [1..n]: színjeles(i))$

Ez a feladat visszavezethető a lineáris keresés programozási tételére. A lineáris keresés specifikációs jelölésénél most csak a logikai komponens van eredményként feltüntetve, hiszen csak ennek megadása a feladat.

$m..n \sim 1..n$
$\beta(i) \sim színjeles(i)$



Ebben a programban az *l*:=*színjeles(i)* értékadás nem-megengedett. Hangsúlyozzuk, hogy a *színjeles(i)* önmagában csak egy kifejezés, nem tekinthető sem feladatnak, sem programnak, szemben az *l*:=*színjeles(i)* értékadással. Ezt az értékadást, mint részfeladatot, tovább kell finomítani, azaz egy megengedett programmal kell majd helyettesíteni, azaz meg kell oldani. A megoldást egy alprogram keretében írjuk most le, amelynek hívását az *l*:=*színjeles(i)* értékadás jelzi. (Dönthettünk volna úgy is, hogy az *l*:=*színjeles(i)* értékadást megoldó részprogramot bemásoljuk *l*:=*színjeles(i)* értékadás helyére, de most inkább az alprogramként való meghívást részesítjük előnyben.)

A részfeladatnak a specifikációja az alábbi lesz.

$A = (t:\mathbb{N}^{n\times m}, i:\mathbb{N}, l:\mathbb{L})$
$Ef = (t=t' \wedge i=i' \in [1..n])$
$Uf = (Ef \wedge l = színjeles(i))$

Figyelembe véve a *színjeles(i)* definícióját ez a részfeladat is visszavezethető a lineáris keresés programozási tételére, pontosabban az optimista lineáris keresésre. Ügyelni kell arra, hogy ciklusváltozónak nem használhatjuk az *i*-t, hiszen az a részfeladatnak egy bemenő adata, foglalt változónév. Használjuk helyette a *j*-t. Ennek megfelelően a visszavezetésnél valójában nem a $\beta(i)$-t, hanem a $\beta(j)$-t kell helyettesíteni a konkrét $t[i,j]=5$ feltétellel, amelyben az *i* a vizsgált diák sorszámára utal.

m..n ~ *1..m*
i ~ *j*
β*(i)* ~ *t*[*i,j*]=*5*

l*:=*színjeles(i)

l,j := *igaz,1*	
l ∧ *j*≤*m*	
	l := *t*[*i,j*]=*5*
	j := *j+1*

Az alprogram feje az *l*:=*színjeles(i)* értékadás, amely most szóról szóra megegyezik a hívással. Korábbi megállapodásunk szerint a fejben szereplő *l* és *i* változók (a formális paraméterváltozók) az alprogram lokális változói, nem azonosak a hívásnál szereplő ugyanolyan nevű globális változókkal (az aktuális paraméterekkel), de sajátos kapcsolatban állnak azokkal. (Természetesen használhattunk volna más lokális változó neveket.) A lokális *i* (bemenő-változó) a híváskor megkapja a globális *i* értékét, a lokális *l* (eredmény-változó) a hívás (azaz az alprogram) befejeződésekor átadja értékét a globális *l*-nek. A paraméterváltozók névegyezése miatt az alprogramban nem hivatkozhatunk a globális *l* és *i* változókra. Használhatja ellenben az alprogram a globális *t* változót. Az alprogram bevezet még egy lokális *j* változót is, amely az alprogram befejeződésekor megszűnik majd.

<u>*6.2. Példa.*</u> Egy *n* diákot számláló osztályban *m* különböző tantárgyból adtak jegyeket a félév végén. Ezek az osztályzatok egy táblázat formájában rendelkezésünkre állnak. (A diákokat és a tantárgyakat most is a sorszámukkal azonosítjuk.) Tudjuk, hogy az osztályban van kitűnő diák, adjuk meg az egyiknek sorszámát!

Ez a feladat nagyon hasonlít a *6.1. példához*. Ott nem tudtuk, hogy van-e kitűnő diák, ezért a lineáris keresés programozási tételére vezettük vissza a feladatot, itt viszont tudjuk, hogy van, és egy ilyen diákot keresünk, ezért elegendő a feladatot a kiválasztás tételére visszavezetni. (Természetesen a *6.1. példánál* kapott program is alkalmazható, hiszen a lineáris keresés nemcsak azt dönti el, hogy van-e kitűnő diák, hanem az elsőt meg is keresi.)

$A = (t{:}\mathbb{N}^{n\times m}, i{:}\mathbb{N})$

$Ef = (t=t' \wedge \exists k\in[1..n]{:}\ színjeles(k))$

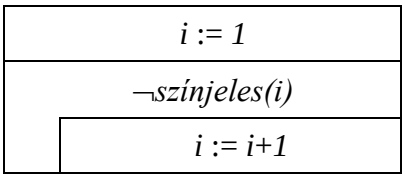
Itt már az előfeltétel megfogalmazásához bevezetjük a *színjeles* függvényt.

$$Uf = (Ef \wedge i \in [1..n] \wedge \textit{színjeles}(i)) =$$
$$= (Ef \wedge i = \underset{i=1}{\textbf{\textit{select}}}\ \textit{színjeles}(i))$$

Ez a feladat visszavezethető a kiválasztás programozási tételére.

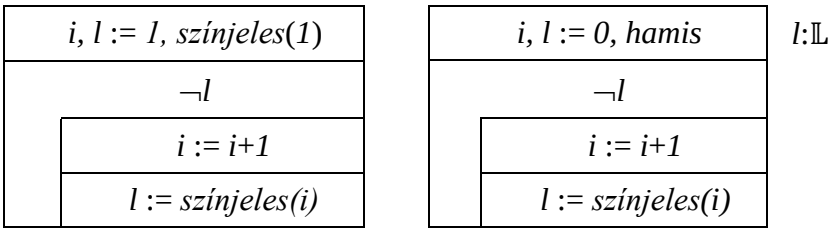
m ~ *1*

$\beta(i)$ ~ *színjeles(i)*



Ebben a programban nem-megengedett feltétel a *színjeles(i)* kifejezés, amely értékét az előző példában elkészített *l*:=*színjeles(i)* alprogram határozza meg. Most erre az alprogramra egy függvényszerű hívást adunk. Amikor a ciklusfeltétel kiértékelésére kerül sor, akkor meghívódik az alprogram, és a lokális eredmény-változójának (*l*:𝕃) értéke közvetlenül a ciklusfeltételnek adódik vissza. Ezzel tehát készen is vagyunk.

Egy alprogram nem lesz más attól, hogy függvényszerűen vagy eljárásszerűen hívjuk meg. Az alprogram mindig egy értékadás által kijelölt feladatot old meg, van eredmény-változója, hiszen egy értékadást valósít meg. A *6.1. példában* éppen ezért függvényszerű hívásról is és eljárásszerű hívásról is beszélhetünk egyszerre, nincs e kettő között látható különbség. Egy eljárásszerű hívás ugyanis mindig felfogható függvényszerű hívásnak is. Fordítva ez már nem igaz, de mindig átalakítható a hívó program úgy, hogy egy függvényszerű hívást eljárásszerű hívással helyettesíthessünk. Ezt akkor tesszük meg, ha ezt valamilyen más szempont (például hatékonyság) indokolja. Az aktuális példánkban nincs ilyen érv, de a játék kedvéért mutassuk meg ezt az átalakítást. A cél az, hogy a hívó programmal olyan ekvivalens programot készítsünk, ahol a ciklusfeltétel *színjeles(i)* nem-megengedett kifejezése egy *l*:=*színjeles(i)* nem-megengedett értékadásba kerül át. Ehhez be kell vezetnünk egy új logikai változót, és a hívó programot az alábbi változatok valamelyikére kell alakítanunk.



6.3. Példa. Egy iskola *n* diákot számláló osztályában *m* különböző tantárgyból osztályoztak a félév végén. Ezek a jegyek egy táblázat formájában rendelkezésünkre állnak. (A diákokat és a tantárgyakat sorszámukkal azonosítjuk.) Igaz-e, hogy minden diáknak van legalább három tárgyból négyese?

A feladat most egy " minden elem olyan-e, hogy ..." típusú eldöntés, ami az optimista lineáris keresés programozási tételére utal. Ebben a keresésben is a diákokat kell egymás után megvizsgálni, de most a vizsgálat tárgya az, hogy van-e legalább három négyese az illetőnek. Ehhez meg kell számolni minden diáknál a négyes osztályzatait. Vegyük észre, hogy ezeket a számlálásokat nem kell a feladatot megoldó optimista lineáris keresés előtt egy „előfeldolgozással" elvégezni, hiszen az optimista kereséshez mindig csak az aktuális diák négyeseinek száma kell, amit ott „helyben" kiszámolhatunk, majd utána el is felejthetünk. Az így kapott megoldás nemcsak a tárigény és futási idő szempontjából lesz hatékonyabb, hanem a program előállítása is egyszerűbb. Arra van csupán szükségünk, hogy a négyesek számát előállító részfeladatot egy absztrakt függvény mögé rejtsük. Ez a függvény a táblázat *i*-edik sora alapján megadja, hogy az *i*-edik diáknak hány négyese van. Arra a kérdésre, hogy miért pont erre a fogalomra vezettünk be egy új függvényt – miért nem arra, hogy van-e legalább három négyese az *i*-edik diáknak – az a válasz, hogy az általunk választott függvény kiszámolását közvetlenül visszavezethetjük majd a számlálás programozási tételére.

A feladat adatai hasonlítanak az előző feladatéra, ami az állapottér és az előfeltétel felírásában tükröződik:

$$A = (t{:}\mathbb{N}^{n\times m},\ l{:}\mathbb{L})$$
$$Ef = (\ t{=}t'\ )$$

A feladat utófeltétele a kimenő adatkomponenst írja le: az *l* akkor igaz, ha minden diáknak három vagy annál több négyese van. Bevezetve a *négyesdb* függvényt, amelyre a *négyesdb(i)* az *i*-edik diák négyeseinek számát adja meg a *t* táblázat *i*-edik sora alapján, az utófeltétel az alábbi módon írható fel:

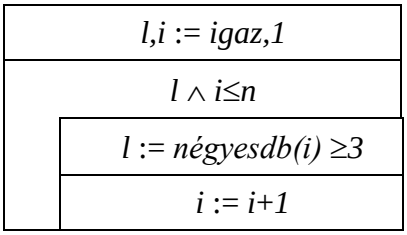
$$Uf = (\ Ef \wedge l = \forall i{\in}[1..n]{:}\ (négyesdb(i){\geq}3)\ ) =$$
$$= (\ Ef \wedge\ l = \forall \overset{n}{\underset{i=1}{\textbf{\textit{search}}}}\ (négyesdb(i) \geq 3)\ )$$

ahol $négyesdb : [1..m] \rightarrow \mathbb{N}$

$$négyesdb(i) = \sum_{\substack{j=1 \\ t[i,j]=4}}^{m} 1$$

Ezt a feladatot egy optimista lineáris keresés oldja meg:

$m..n \sim 1..n$
$\beta(i) \sim$ *négyesdb(i)*≥ 3



A visszavezetéssel előállított programban az *l*:=*négyesdb(i)*≥*3* értékadás nem-megengedett. Ha tüzetesebben megvizsgáljuk az értékadás jobboldalán álló kifejezést, akkor észrevehetjük, hogy annak *négyesdb(i)* részkifejezése a nem-megengedett. Készíthetünk ennek kiszámolására egy *s*:=*négyesdb(i)* értékadást megvalósított függvényszerűen meghívott alprogramot, ahol az *s* egy darabszámot tartalmazó eredmény-változó. (Alternatív megoldás, ha a hívó programban bevezetünk egy *s*:$\mathbb{N}$ segédváltozót, és az *l*:=*négyesdb(i)*≥*3* értékadást az (*s*:=*négyesdb(i)*; *l*:=*s*≥*3*) szekvenciára bontjuk fel, és az *s*:=*négyesdb(i)* alprogramot eljárásszerűen hívjuk.)

Az *s*:=*négyesdb(i)* értékadás által kijelölt részfeladat specifikációja:

$A = (t{:}\mathbb{N}^{n\times m}, i{:}\mathbb{N}, s{:}\mathbb{N})$

$Ef = (t=t' \wedge i=i' \in [1..n])$

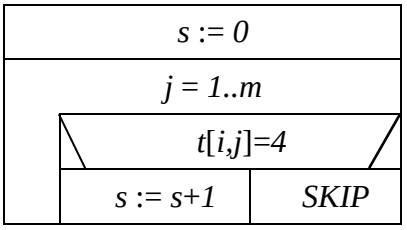
$Uf = (Ef \wedge s = négyesdb(i)) = (Ef \wedge s = \sum_{\substack{j=1 \\ t[i,j]=4}}^{m} 1)$

Ez visszavezethető a számlálás programozási tételére.

$m..n \sim 1..m$

$\beta(i) \sim t[i,j]=4$

'b(i)



6.4. Példa. Egy iskola *n* diákot számláló egyik osztályában *m* különböző tantárgyból adtak jegyeket a félév végén. Ezek az osztályzatok egy táblázat formájában állnak rendelkezésünkre. (A diákokat és a tantárgyakat sorszámukkal azonosítjuk.) Számoljuk meg, hány kitűnő diák van az osztályban!

A szövegéből egyértelműen kiderül, hogy egy számlálással lehet a feladatot megoldani. A számlálást a diákok között, tehát az [*1..n*] intervallum felett végezzük, és azt a feltételt vizsgáljuk, amely megmondja egy diákról, hogy csupa ötöse van-e. Ennek a feltételnek a jelölésére a korábban már bevezetett *színjeles* logikai függvényt használjuk.

Lássuk a specifikációt!

$A = (t{:}\mathbb{N}^{n\times m}, s{:}\mathbb{N})$

$Ef = (t=t')$

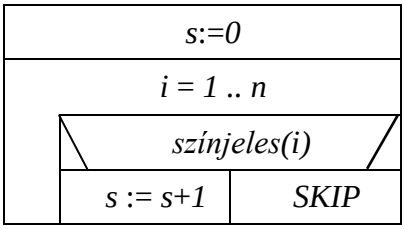
$$Uf = \left(Ef \wedge s = \sum_{\substack{i=1 \\ színjeles(i)}}^{n} 1 \right)$$

ahol

$színjeles : [1..n] \rightarrow \mathbb{L}$

$színjeles(i) = \forall j \in [1..m]: t[i,j]=5$

A feladatot tehát egy számlálásra vezetjük vissza, ahol az intervallum az [*1..n*], a β*(i)* feltétel a *színjeles(i)*.



A *színjeles(i)* nem-megengedett feltétel a korábban már definiált alprogram függvényszerű hívását jelöli.

6.5. Példa. Egy iskola *n* diákot számláló egyik osztályában *m* különböző tantárgyból osztályoztak a félév végén. Ezek a jegyek egy táblázat formájában állnak rendelkezésünkre.(A diákokat és a tantárgyakat sorszámukkal azonosítjuk.) Melyik diáknak van a legtöbb négyese?

Ennél a feladatnál a maximum kiválasztás programozási tételét kell használnunk. Mivel egy iskolai osztályhoz biztos tartoznak diákok, így a maximum kiválasztás értelmes. A maximális értéket most nem közvetlenül egy intervallum pontjai közül, hanem az intervallum

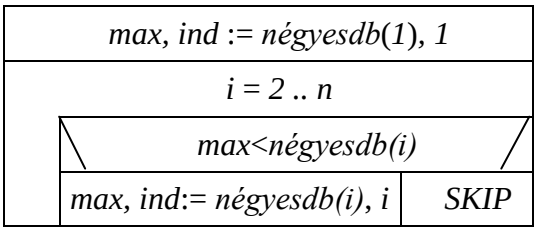
pontjaihoz (azaz a diákokhoz) rendelt értékek (egy diák négyeseinek száma) közül kell kiválasztani. A feladatban jól körülvonalazódik a *6.3. példából* már ismert részfeladat is, amely egy diák négyeseinek számát állítja elő.

$A = (t{:}\mathbb{N}^{n\times m},\ ind{:}\mathbb{N})$

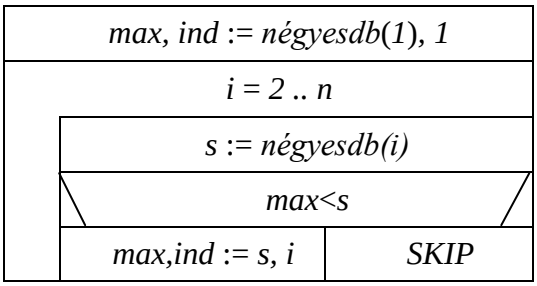
$Ef = (\ t=t' \wedge n>0 \wedge m>0\ )$

$Uf = (\ Ef \wedge négyesdb(ind) = \max_{i=1}^{n} négyesdb(i)\ )$

A feladatot egy maximum kiválasztásra vezetjük vissza, ahol az intervallum az [*1..n*], és az *f(i)* kifejezést a *négyesdb(i)* helyettesíti.



Mivel rendelkezünk az *s*:=*négyesdb(i)* alprogrammal, amelyet speciálisan a *négyesdb(1)* kiszámolására is felhasználhatunk, tulajdonképpen készen vagyunk: az alprogram függvényszerű hívására van szükség. Hatékonysági okból azonban célszerű a ciklusmagbeli elágazásból a *négyesdb(i)*-t egy értékadásba kiemelni és az alprogramot eljárásszerűen hívni, mert így az *i* egy rögzített értékére nem kell a *négyesdb(i)* értékét esetenként kétszer is kiszámolni.



6.6. Példa. Egy iskola *n* diákot számláló egyik osztályában *m* különböző tantárgyból osztályoztak a félév végén. Ezek a jegyek egy táblázat formájában állnak rendelkezésünkre.(A diákokat és a tantárgyakat sorszámukkal azonosítjuk.) Ki az a diák, aki a csak *4*-es illetve *5*-ös osztályzatú diákok között a legjobb átlagú? Ha van ilyen, adjuk meg az átlagát is!

Ez a feladat egy feltételes maximumkereséssel oldható meg, hiszen csak az adott tulajdonságú diákok átlagai között keressük a legjobbat. Ezen belül önálló részfeladatot alkot annak eldöntése, hogy egy diák csak *4*-es illetve *5*-ös osztályzatú-e, illetve önálló részfeladat egy diák átlagának kiszámolása is. Az előbbi a feltételes maximum keresés β feltételét adja, az utóbbi a maximumkeresésnél használt *f* függvényt. Ezeket a részfeladatokat egy-egy absztrakt függvénnyel jelöljük ki. Bevezetjük a "csak *4*-es illetve *5*-ös osztályzatú" tulajdonságot eldöntő *jó* logikai függvényt, amely az *i*-edik diák esetén akkor ad igaz értéket, ha a táblázat *i*-edik sorában csak négyesek vagy ötösök állnak. Bevezetjük továbbá az *összeg* függvényt, amely megadja az *i*-edik diák jegyeinek összegét. Azért az összegét, és nem az átlagát, mert a maximum keresés eredménye mindkét esetben ugyanaz a diák. Természetesen a végeredmény előállításához külön kell majd gondoskodni a legjobb diák átlagának kiszámolásáról.

$A = (t{:}\mathbb{N}^{n\times m},\ l{:}\mathbb{L},\ ind{:}\mathbb{N},\ átl{:}\mathbb{R})$

$Ef = (\ t=t'\ )$

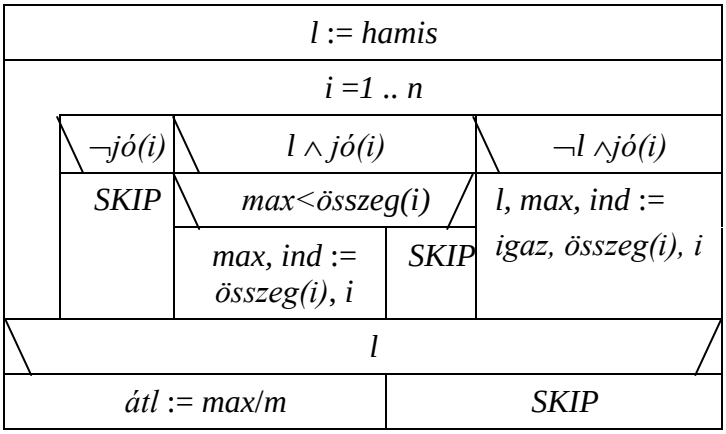
$$Uf = (\ Ef \wedge l,max,ind = \mathop{\boldsymbol{max}}_{i=1}^{n}{}_{jó(i)} összeg(i)\ \wedge (l \rightarrow átl=max/m)\ )$$

ahol

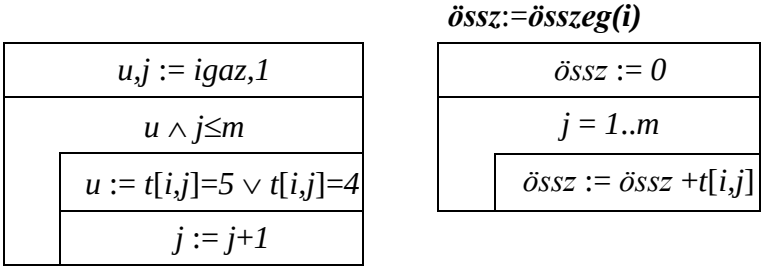
$jó{:}[1..n]\rightarrow\mathbb{L}$ $\qquad$ $összeg{:}[1..n]\rightarrow\mathbb{N}$

$jó(i) = \forall j\in[1..m]{:}\ (t[i,j]=5 \vee t[i,j]=4)$ $\qquad$ $összeg(i) = \sum_{j=1}^{m} t[i,j]$

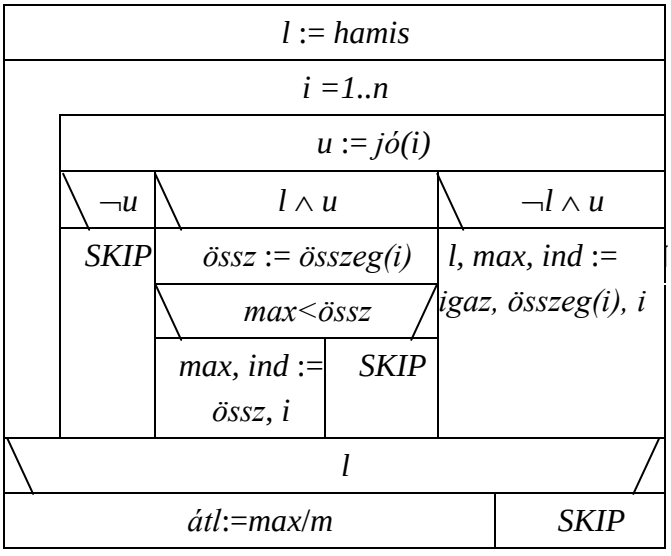
A feladat az utófeltétel alapján két részre bontható: egy [*1..n*] intervallumú, *jó(i)* feltételű és *összeg(i)* függvényű feltételes maximumkeresésnek, valamint egy elágazásnak a szekvenciájára. Az elágazás feltétele az *l*, igaz ága az *átl*:=*max*/*m* értékadás, hamis ága a *SKIP* lesz.



A *jó(i)* kiszámolásához az *u*:=*jó(i)* értékadást megoldó alprogramra van szükség, ahol az *u* egy logikai változó. Az *u*:=*jó(i)* nem-megengedett értékadás egy optimista lineáris kereséssel helyettesíthető, amelyben futóindexnek a *j*-t használjuk. Az *összeg(i)* meghatározásához az *össz*:=*összeg(i)* értékadást megoldó alprogramra van szükség, ahol az *össz* egy egészszám változó. Az *össz*:=*összeg(i)* részfeladat egy összegzésre vezethető vissza, amelyben futóindexnek ugyancsak a *j*-t használjuk, de ez nem ugyanaz a *j*, mint előbb.



A főprogram fenti verziója ezen alprogramok függvényszerű hívását mutatja, de a hatékonyság érdekében ezeket a hívásokat érdemes egy-egy értékadásba kiemelni. Az *u*:=*jó(i)* hívást a ciklusmag hármas elágazása előtt kell végrehajtani, az elágazás feltételeiben pedig az *u* változót használni. Az *össz*:=*összeg(i)* hívásra a középső ágban található másik elágazás előtt van szükség, amelyben elég az *össz* változóra hivatkozni. Mind az *u*, mind *össz* egy újonnan bevezetett segédváltozója a programnak.



6.3. Program-átalakítások

Korábbi feladat-megoldásainkban gyakran alkalmaztunk program-átalakításokat. A program-átalakítás során egy programból egy olyan másik programot állítunk elő, amely megoldja mindazokat a feladatokat, amelyeket az eredeti program is megoldott. Ezt gyakran ekvivalens átalakítással érjük el, azaz úgy, hogy az új program hatása teljesen megegyezik az eredeti program hatásával. Példaként emlékeztetünk arra, ahogyan 6.2. példában a kiválasztás programozási tételét kétféleképpen is átalakítottuk azért, hogy explicit módon egy értékadásba emeljük ki a ciklusfeltételében szereplő nem-megengedett kifejezést.

Egy-egy program-átalakításnak a helyessége a programozási modellünk eszközeivel belátható, de itt nem foglakozunk a bizonyítással, hanem a lustább „látszik rajta, hogy jó" magyarázatot használjuk.

Egy program-átalakításnak igen sokféle célja lehet. Esetenként csak szebbé, áttekinthetőbbé formálja a programot, néha az implementáció, azaz a konkrét számítógépes környezetbe ültetés végett van rá szükség, máskor a program hatékonyságán lehet javítani. Az előbb felsorolt szempontokon kívül különösen fontosak azok a program-átalakítások, amelyek a program előállítását segítik a programtervezés fázisában.

A programtervezést támogató átalakítások közé sorolható például az, amikor egy új változó bevezetésének segítségével emeltünk ki nem-megengedett kifejezést (másik értékadás jobboldali kifejezésének részét, egy elágazás vagy ciklus feltételét) egy önálló értékadásba, és ezáltal kijelöltük a programunknak egy olyan részfeladatát, amelyet aztán megoldottunk. Ugyancsak ilyen átalakításnak tekintjük a rekurzív függvények kibontását is, amelyre a következő alfejezetben mutatunk példákat.

Egy szimultán értékadás egyszerű értékadások szekvenciájára történő felbontása is támogathatja a programtervezést, de többnyire egy implementálást támogató átalakítás, amennyiben olyan programozási nyelven kell leírnunk a programunkat, amelyik nem ismeri (és többnyire nem ismerik) a szimultán értékadást. Eddig főleg olyan szimultán értékadásokkal találkoztunk, amelyet alkotó egyszerű értékadások függetlenek voltak egymástól, és ebben az esetben az átalakítás könnyű volt.

Egy egyszerű értékadás akkor függ egy másiktól, ha a jobboldali kifejezésében szerepel egy olyan változó, amely a másik értékadás baloldali változója. Ha ilyen függés nem áll fenn a szimultán értékadást alkotó egyszerű értékadások között, akkor azokat tetszőleges sorrendben végrehajtva a szimultán értékadással azonos hatású szekvenciához jutunk. Ha van függő viszony, de ezek rendszere nem alkot kört, azaz a szimultán értékadás egyszerű értékadásainak bármelyik csoportjában lehet találni olyat, amelyik nem függ a csoport többi egyszerű értékadásától, akkor az egyszerű értékadásokat olyan sorrendben kell végrehajtani, hogy előre azt az egyszerű értékadást vesszük, amelyiktől senki más nem függ, majd mindig azt, amelyiktől csak az előtte végrehajtottak függnek. Egy teljesen általános szimultán értékadás felbontásához azonban –

amikor a felbontással kapott egyszerű értékadások kölcsönösen függnek egymástól – új változók bevezetésére van szükség. Az

$$x_1, \dots, x_n := F_1(x_1, \dots, x_n), \dots, F_n(x_1, \dots, x_n)$$

értékadás helyett az alábbi két programot használhatjuk:

$y_1 := x_1$
…
$y_{n-1} := x_{n-1}$
$x_1 := F_1(x_1, x_2, \dots, x_{n-1}, x_n)$
$x_2 := F_2(y_1, x_2, \dots, x_{n-1}, x_n)$
…
$x_{n-1} := F_{n-1}(y_1, y_2, \dots, y_{n-1}, x_n)$
$x_n := F_n(y_1, y_2, \dots, y_{n-1}, x_n)$

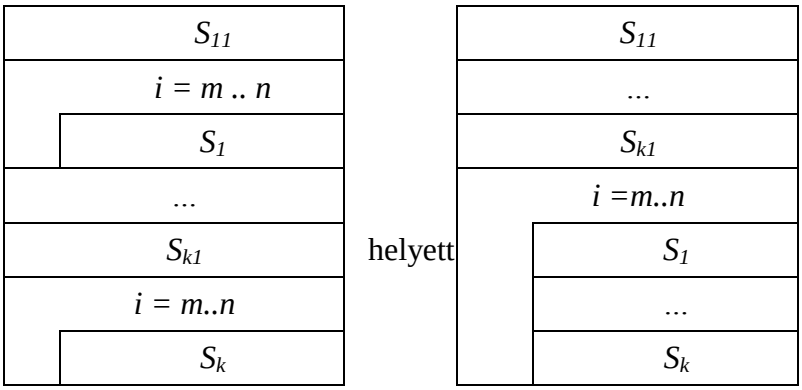
vagy

$y_1 := x_1$
…
$y_{n-1} := x_{n-1}$
$x_1 := F_1(y_1, y_2, \dots, y_{n-1}, x_n)$
$x_2 := F_2(y_1, y_2, \dots, y_{n-1}, x_n)$
…
$x_{n-1} := F_{n-1}(y_1, y_2, \dots, y_{n-1}, x_n)$
$x_n := F_n(y_1, y_2, \dots, y_{n-1}, x_n)$

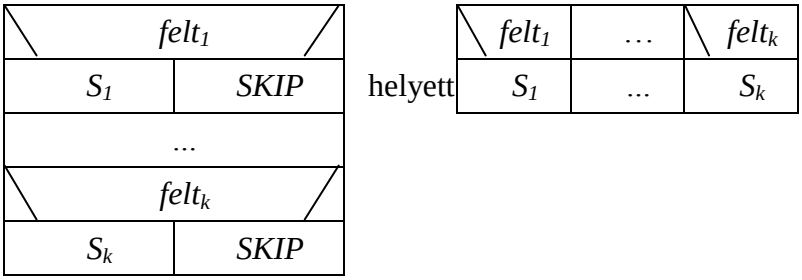
Az implementáció egyszerűsítését támogatja a szekvenciák sorrendjének megváltoztatása is. Természetesen, ha a szekvencia második tagja függ az elsőtől, akkor erre nincs lehetőség. Programrészek (itt a szekvenciák tagjai is) függetlenségén ugyanazt kell érteni, mint az értékadások esetében, ugyanis minden programrész helyettesíthető egy vele ekvivalens – többnyire nem-megengedett – értékadással.

Program hatékonyságát támogató átalakítások közé soroljuk a különféle *összevonási és kiemelési szabályokat*, a *rekurzív függvény kibontását* (lásd következő alfejezet), vagy alkalmas *segédváltozók bevezetését* többször felhasznált értékek tárolására.

Összevonásról például akkor beszélhetünk, amikor egymáshoz hasonló, de egymás működésétől független ciklus szekvenciájából egyetlen ciklust készítünk úgy, hogy az új ciklus magja az eredeti ciklusmagok szekvenciája lesz. Az alábbi átalakításnál szekvencia-cserét is végeztünk, amikor a ciklusok előtti inicializáló programrészeket mind előre hoztuk. Természetesen ezek sem függhetnek egymástól.

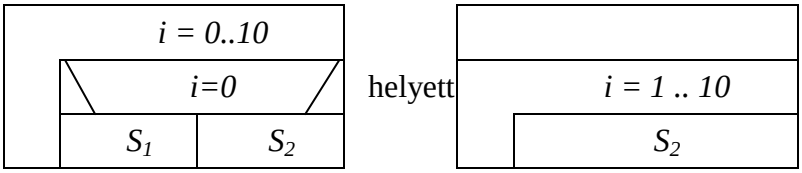


Sok egymást kizáró, de az összes esetet lefedő feltételű és egymás működésétől független kétágú elágazásnak a szekvenciájából egyetlen sokágú elágazást készítünk.



A baloldali algoritmusnak nem lehet olyan végrehajtása, amikor minden elágazásnak az „else” ága hajtódik vége, mert a feltételrendszer a feltevésünk szerint teljes. Ha ez nem állna fenn, akkor a jobboldali elágazást ki kell egészíteni egy „else” ággal.

Kiemelési szabály alkalmazására mutat speciális példát az alábbi átalakítás:



Az összevonásokkal illetve kiemelésekkel nemcsak hatékonyságot lehet javítani, hanem a program áttekinthetőségén, olvashatóságán is.

III. RÉSZ
TÍPUSKÖZPONTÚ PROGRAMTERVEZÉS

Az előző rész feladataiban és az azokat megoldó programokban olyan adatokat használtunk, amelyeket a legtöbb magasszintű programozási nyelvben megtalálunk, így eddig az adatok tervezésére és megvalósítására nem kellett sok energiát fordítanunk.

Az igazán nehéz feladatok adatai azonban nem ilyenek. Ezek az adatok jóval összetettebbek, a rajtuk elvégezhető műveletek bonyolultabbak. Ezen adatok konkrét tulajdonságaitól az egyszerűbb kezelés érdekében célszerű elvonatkoztatni; helyettük olyan adatokkal dolgozni, amelyeket úgy kapunk, hogy a lényeges tulajdonságokat kiemeljük a lényegtelenek közül, azaz általánosítjuk azokat. Annak következtében, hogy az adatok elvonatkoztatnak a feladattól, lehetővé válik a feladatnak egy jobban átlátható, egyszerűbb modelljét felállítani, amellyel a feladatot könnyebben fogalmazhatjuk és oldhatjuk meg.

Ugyanakkor az így kapott kitalált adatok többnyire elvonatkoztatnak azon programozási környezettől is, ahol majd a megoldó programnak működnie kell. A tervezés során ezért ki kell térni annak vizsgálatára, hogyan lehet a kitalált adatainkat a számítógépen megjeleníteni, azaz hogyan lehet az adat értékeit ábrázolni (reprezentálni); továbbá azokat a tevékenységeket (műveleteket), amelyeket az adaton el akarunk végezni, milyen programokkal lehet kiváltani (implementálni). Amikor egy adatot így jellemzünk: azaz megadjuk az értékeinek reprezentációját és az adattal végezhető műveletek implementációját, akkor az adat típusát adjuk meg. Sokszor előfordul, hogy egy adattípus bevezetésénél elsőre nem sikerül olyan reprezentációt és implementációt találni, amely a konkrét programozási környezetben is leírható. Ennek nem az ügyetlenség az oka, hanem az, hogy elsősorban a megoldandó feladatra koncentrálunk, és azt vizsgáljuk, milyen típusműveletek segíthetik a feladat megoldását. Az ilyen nem-megengedett, úgynevezett absztrakt típust ezért aztán egy megfelelő típussal kell majd a konkrét programozási környezetben helyettesítenünk, megvalósítanunk.

Egy feladat valamely adattípusának egy-egy típusművelete a feladatot megoldó program része lesz. A típusműveletek bevezetésével valójában részfeladatokat jelölünk ki a megoldandó feladatban. A típusműveletek implementálása az ilyen részfeladatok megoldását, összességében tehát a feladat finomítását jelenti. A feladat megoldása így két részre válik: egyrészt a bevezetett típus műveleteinek segítségével megfogalmazott megoldásra (főprogramra), másrészt a típusműveleteket – mint részfeladatokat megoldó – részprogramokra. A típusoknak az alkalmazása tehát az eredeti feladatot részfeladatokra felbontó speciális, típus központú feladatfinomítási technika.

7. Típus

Már a fenti bevezetőből is látszik, hogy egy adat típusát több nézőpontból lehet szemlélni. Amikor arra vagyunk kíváncsiak, hogy mire használhatjuk az adatot, azaz melyek a lehetséges értékei és ezekkel milyen műveleteket lehet végezni, akkor a típus felületét, kifelé megmutatkozó arcát nézzük; ezt szokták a típus interfészének, a ***típus specifikációjának*** nevezni. Amikor viszont az adat típusának egy adott programozási környezetbe való beágyazása a cél, akkor azt vizsgáljuk, hogy milyen elemekkel helyettesíthetőek, ábrázolhatóak a típusértékek, illetve melyek azok a programok, amelyek hatása a típusműveletekével azonos annak ellenére, hogy nem közvetlenül a típus értékeivel, hanem az azokat helyettesítő elemekkel dolgoznak. Ha vesszük a típusértékek helyettesítő elemeit, a helyettesítés módját, azaz a ***típusértékek reprezentációját*** (ábrázolását), valamint a típusműveleteket kiváltó programokat, azaz a ***típusműveletek implementációját***, akkor ezeket együtt a ***típus megvalósításának*** (realizálásának) nevezzük.[14] Egy ***adat típusa*** magába foglalja a típus-specifikációt és az ennek megfelelő, az ezt megvalósító típus-realizációt.

7.1. A típus fogalma

Most egy olyan példát fogunk látni, amelyik megoldása során pontosítjuk a típus korábban bevezetett fogalmát, és ezáltal eljutunk a korszerű típus definíciójához.

7.1. Példa. Képzeljük el, hogy olyan űrállomást állítanak Föld körüli pályára, amelynek az a feladata, hogy adott számú észlelt ismeretlen tárgy közül megszámolja, hány tartózkodik az űrállomás közelében (mondjuk *10000 km*-nél közelebb hozzá).

A feladat egyik adata az űrállomás; a másik az ismeretlen tárgyakat tartalmazó sorozat, pontosabban egy tömb; és természetesen az eredmény.

$A = (\mathit{Űrszonda}\ ,\ \mathit{UFO}^n\ ,\ \mathbb{N})$

Mindenek előtt válasszuk el a feladat adataiban a lényegest a lényegtelentől. Az űr ismeretlen tárgyait egy-egy térbeli pontként szemlélhetjük, hiszen mindössze a térben elfoglalt pozíciójuk lényeges a feladat megoldása szempontjából. Egy űrállomás megannyi tulajdonsága közül annak pozíciója és a védelmi körzete lényeges: az űrállomás így helyettesíthető egy térbeli gömbbel. Ennek a valóságos feladattól való elvonatkoztatásnak az eredménye az alábbi specifikáció.

[14] A műveletek implementációja a reprezentációra épül, ezért sokszor az implementáció fogalmába a reprezentációt is beleértik, azaz típus-implementáció alatt a teljes típus-megvalósításra gondolnak.

$A = (g{:}Gömb, v{:}Pont^n, s{:}\mathbb{N})$

$Ef = (g=g' \wedge v=v')$

$$Uf = \left(Ef \wedge s = \sum_{\substack{i=1 \\ v[i]\in g}}^{n} 1 \right)$$

A feladat megoldható egy számlálás segítségével. A megoldó programnak az a feltétele, hogy vajon benne van-e a $v[i]$ pont a g gömbben, nem megengedett, mert feltesszük, hogy programozási környezetünkben nem ismert sem a gömbnek, sem a pontnak a típusa. Emeljük ki egy l logikai változó segítségével a számlálás nem-megengedett feltételét egy $l := v[i] \in g$ értékadásba, és tekintsük ezt a továbbiakban egy gömb műveletének.

Egy *Gömb* típusú adat lehetséges értékei gömbök, amelyeken most egyetlen műveletet akarunk végezni: a „benne van-e egy pont egy gömbben” vizsgálatot.

Egy gömb azon pontok halmaza (mértani helye), amelyeknek egy kitüntetett ponttól (a középponttól) mért távolsága egy megadott értéknél (a sugárnál) kisebb vagy egyenlő. Formálisan: *Gömb*=$\{g \in \mathbf{2}^{Pont} |$ *van olyan* $c \in Pont$ *középpont és* $r \in \mathbb{R}$ *sugár, hogy bármely* $p \in g$ *pontra a távolság*$(c,p) \le r \}$.)

A típusműveletet, mint részfeladatot az alábbi módon specifikálhatjuk:

$A = (g{:}Gömb, p{:}Pont, l{:}\mathbb{L})$

$Ef = (g=g' \wedge p=p')$

$Uf = (Ef \wedge l = p \in g)$

Ezzel körvonalaztuk, más szóval specifikáltuk a *Gömb* típust.

Általában egy adattípus specifikációjáról (ez a típus-specifikáció) akkor beszélünk, amikor megadjuk az adat által felvehető lehetséges értékek halmazát, a típusérték-halmazt, és az ezen értelmezett típusműveleteknek, mint feladatoknak a specifikációit. Ezeknek a feladatoknak közös vonása, hogy állapotterüknek legalább egyik komponense a megadott típusérték-halmaz.

A „benne van-e egy pont egy gömbben” műveletét megoldja az $l := p \in g$ értékadás, amely nem-megengedett (végtelen elemű g halmaz esetén ez a részfeladat nem oldható meg véges lépésben). Ezért egy megengedett programmal kell helyettesíteni.

Ehhez először a gömböt kell másként, nem pontok halmazaként ábrázolni, hanem például a középpontjával és a sugarával. Természetesen egy térbeli pont és egy valós szám önmagában nem egy gömb, de helyettesíti, reprezentálja a gömböt. Ezt a reprezentációt matematikai értelemben egy $\rho{:}Pont \times \mathbb{R} \to 2^{Pont}$ függvény írja le, amely egy c középponthoz és egy r sugárhoz az annak megfelelő gömböt rendeli. Formálisan: $\rho(c,r) = \{q \in Pont \mid$ távolság$(c,q) \le r \} \subseteq 2^{Pont}$. Ez a függvény negatív sugárra is értelmes, de célszerű volna a negatív sugarú gömböket kizárni a vizsgálatainkból. Ezt megtehetjük úgy, hogy megszorítást adunk a gömböket helyettesítő $(c,r) \in Pont \times \mathbb{R}$ párokra: $r \ge 0$. Ezt a megszorítást a típus invariáns állításának hívják. Ez formálisan

egy I:Pont×$\mathbb{R}$ →$\mathbb{L}$ logikai függvény, ahol I(c,r) = r≥0. A típus invariánsa leszűkíti a ρ reprezentációs függvény értelmezési tartományát.

A gömbök itt bemutatott reprezentációja azért előnyös, mert a „benne van-e egy pont egy gömbben" típusművelet így már visszavezethető két pont távolságának kiszámolására, hiszen az l:=p∈g értékadás kiváltható az l:=távolság(c,p)≤r értékadással. Hangsúlyozzuk, hogy a két értékadás nem azonos hatású, hiszen – hogy mást ne mondjunk – eltér az állapotterük: az elsőé a (Gömb, Pont, $\mathbb{L}$) a másodiké a (Pont, $\mathbb{R}$, Pont, $\mathbb{L}$). A második értékadás állapotterében nem szerepel a gömb, hiszen ott azt egy pont és egy valós szám helyettesíti. Mégis a második értékadás abban az értelemben megoldja az első értékadás által megfogalmazott feladatot, amilyen értelemben egy gömb reprezentálható a középpontjával és a sugarával. Más szavakkal úgy mondjuk, hogy az l:=távolság(c,p)≤r értékadás a reprezentáció értelmében megoldja, azaz implementálja az l:=p∈g műveletet. Az l:=távolság(c,p)≤r értékadás közvetlenül az alábbi feladatot oldja meg

A = (c:*Pont*, r:$\mathbb{L}$, p:*Pont*, l:$\mathbb{L}$)
Ef = ($c=c' \wedge r=r' \wedge p=p'$)
Uf = *(Ef* $\wedge$ l = *távolság(c,p)*≤*r)*

de közvetve, a reprezentáció értelmében megoldja az eredeti feladatot is:

A = (g:*Gömb*, p:*Pont*, l:$\mathbb{L}$)
Ef = ($g=g' \wedge p=p'$)
Uf = ($Ef \wedge l = p \in g$)

Ezzel el is érkeztünk a korszerű típusfogalom definíciójához. Egy adat típusán azt értjük, amikor megadjuk, hogy az adat által felvehető lehetséges értékeket hogyan ábrázoljuk (reprezentáljuk), azaz milyen elemmel (értékkel vagy értékcsoporttal) helyettesítjük, továbbá leírjuk a típus műveleteit implementáló programokat, amelyek közös vonása, hogy állapotterükben komponensként szerepel a reprezentáló elemek halmaza. A típusértékeket helyettesítő (ábrázoló) reprezentáns elemeket gyakran egy bővebb halmazzal és egy azon értelmezett logikai állítással, a típus invariánsával jelöljük ki: ekkor az állítást kielégítő elemek a reprezentáns elemek. A reprezentáció több, mint a reprezentáns elemek halmaza: a reprezentáns elemek és a típus-értékek kapcsolatát is leírja. Ez tehát egy leképezés, amely gyakran egy típusértékhez több helyettesítő elemet is megad, sőt ritkán előfordul, hogy különböző típusértéket ugyanazon reprezentáns elemmel helyettesíti.

A Gömb típusleírása sokkal árnyaltabban és részletesebben mutatja be a gömböket, mint a Gömb korábban megadott típus-specifikációja. Igaz, hogy a típusleírásban explicit módon csak a gömbök ábrázolásáról (reprezentációjáról) és egy erre az ábrázolásra támaszkodó művelet programjáról (l:=távolság(c,p)≤r) esik szó, de ez implicit módon definiálja a gömbök halmazát (a típusérték-halmazt) és az azokon értelmezett („benne van-e egy pont a gömbben") típusműveletet is. Általában is igaz, hogy ha ismerjük a típus-invariánst és a reprezentációs

leképezést, akkor a típus-invariáns által kijelölt elemekhez ez a leképezés hozzárendeli a típusértékeket, így megkapjuk a típusérték-halmazt. A típusművelet programjából pedig (mint minden programból) kiolvasható az a feladat, amit a program megold, és ha ebben a feladatban a reprezentáns elemek helyére az azok által képviselt típusértékeket képzeljük, akkor megkapjuk a típusművelet feladatát. Így előáll a típus-specifikáció. Ezt a jelenséget úgy nevezzük, hogy ***a típus kijelöli önmaga specifikációját.***

Sajnos a Gömb típus műveletének programja nem megengedett, hiszen két térbeli pont távolságának (távolság(c,p)) kiszámítása sem az. Ezért ki kell dolgoznunk a Pont típust, amelynek művelete a d:=távolság(c,p) értékadás lesz.

A Pont típus értékei a térbeli pontok, amelyek például egy háromdimenziós derékszögű koordináta-rendszerben (ennek az origója tetszőleges rögzített pont lehet) képzelhetőek el, és ilyenkor azok a koordinátáikkal helyettesíthetők. ($\rho:\mathbb{R}\times\mathbb{R}\times\mathbb{R}\rightarrow$Pont) Hangsúlyozzuk, hogy három valós szám nem azonos egy térbeli ponttal, de egy rögzített koordináta-rendszerben egyértelműen reprezentálják azt. Mivel bármilyen koordináta-hármas egy térbeli pontot helyettesít, ezért a típus invariánsával ($I:\mathbb{R}\times\mathbb{R}\times\mathbb{R}\rightarrow\mathbb{L}$) nem kell megszorítást tennünk, az invariáns tehát az azonosan igaz állítás. Ebben a reprezentációban két pont távolságát az euklideszi képlet segítségével számolhatjuk ki. A képlet alapján felírt értékadás megoldja, implementálja a két pont távolságát meghatározó részfeladatot annak ellenére, hogy a művelet állapotterében pontok, az azt megoldó program állapotterében pedig valós számok szerepelnek. Mivel a képlet valós számokkal és a valós számok műveleteivel dolgozik, értéke már egy megengedett programmal kiszámolható.

$$A = (p.x{:}\mathbb{R},\ p.y{:}\mathbb{R},\ p.z{:}\mathbb{R},\ q.x{:}\mathbb{R},\ q.y{:}\mathbb{R},\ q.z{:}\mathbb{R},\ d{:}\mathbb{R})$$

$$d := \sqrt{(p.x-q.x)^2+(p.y-q.y)^2+(p.z-q.z)^2}$$

Felhasználva a *Gömb* és a *Pont* típus definícióját visszatérhetünk az eredeti feladat megoldásának elején említett számláláshoz, és az abban szereplő $l{:=}v[i]{\in}g$ értékadást az

$$l := \sqrt{(v[i].x-g.c.x)^2+(v[i].y-g.c.y)^2+(v[i].z-g.c.z)^2} \le g.r$$

programmal helyettesíthetjük. Ebben a $v[i].x$ a $v[i]$ pont x koordinátáját, $v[i].y$ az y koordinátáját és $v[i].z$ a z koordinátáját jelöli, továbbá a $g.c$ a g gömb középpontja, $g.r$ pedig a sugara. Az eredményül kapott program állapottere nem egyezik meg a feladat állapotterével: az eredeti állapottér gömbje és pontjai helyett a megoldó program valós számokkal dolgozik. Mégis, ez a program nyilvánvalóan megoldja a feladatot, még ha ezt nem is a megoldás korábban megadott definíciójának értelmében teszi.

A feladat megoldásában tetten érhető az adatabsztrakciós programozási szemléletmód, amely az adattípust állítja középpontba. Első lépésként elvonatkoztattuk a feladatot az űrszondák és tárgyak világából a térbeli gömbök és pontok világába. Itt megtaláltuk a feladat megoldásának „kulcsát”, azt a részfeladatot, amelyik el tudja dönteni, hogy egy pont benne van-e egy gömbben.

Ezt felhasználva már elkészíthettük az eredeti feladatot megoldó számlálást. Egyetlen, bár nem elhanyagolható probléma maradt csak hátra: a megoldás kulcsának nevezett részfeladatot, mint a *Gömb* típus egy műveletét, kellett megoldani. Ehhez azonban nem az absztrakt gömböket tartalmazó eredeti állapottéren kerestünk megoldó programot, hanem ott, ahol a gömböt a középpontjával és sugarával helyettesítettük. Más szavakkal, a *Gömb* típust meg kellett *valósítani*: először reprezentáltuk a gömböket, majd a reprezentáns elemek segítségével újrafogalmaztuk a részfeladatot. Az újrafogalmazott részfeladat megoldásához újabb adatabsztrakciót alkalmaztunk: bevezettük a *Pont* típust, amelyhez a két pont távolságának részfeladatát rendeltük típusműveletként. Végeredményben egy ismert típus, a valós számok típusának segítségével reprezentáltuk a pontokat és a gömböket, a műveleteket megoldó programok pedig ugyancsak a valós számok megengedett környezetében működnek.

Az adatabsztrakció alkalmazása intuíciót és sok gyakorlást igényel. Hogy mást ne említsünk, például nehezen magyarázható meg az, honnan láttuk az előző példában már a megoldás elején, hogy a "benne van-e a pont a gömbben" műveletet a *Gömb* típushoz és nem a *Pont* típushoz kell rendelni. Ez a döntés alapvetően meghatározta a megoldás irányát: először a *Gömb* típussal és utána a *Pont* típussal kellett foglalkozni; de kijelölte azt is, hogy a *Gömb* típus reprezentálásához felhasználjuk a *Pont* típust.

7.2. Típus-specifikációt megvalósító típus

Ha van egy típus-specifikációnk és külön egy típusunk, akkor joggal vetődik fel az a kérdés, vajon a típus megfelel-e az adott típus-specifikációnak: megvalósítja-e a típus-specifikációt. Ezt vizsgálhatnánk úgy is, hogy összehasonlítjuk a vizsgált típus-specifikációt a típus által kijelölt típus-specifikációval, de közvetlenül úgy is, hogy megvizsgáljuk vajon a típus reprezentáns elemei megfelelően helyettesíthetik-e a típus-specifikáció típusértékeit, és hogy a típus programjai rendre megoldják-e a típus-specifikáció feladatait (természetesen úgy értve, hogy a programok a típusértékek helyett azok reprezentáns elemeivel számolnak.)

Annak, hogy a típus megvalósítson egy típus-specifikációt két kritériuma van. Egyrészt minden típusérték helyettesíthető kell legyen reprezentáns elemmel, és minden reprezentáns elemhez tartoznia kell típusértéknek. Másrészt a típus-specifikáció minden feladatát a típus valamelyik programja a reprezentáció (alapján) értelmében megoldja (implementálja).

Egy típus-specifikációhoz több azt megvalósító típus is tartozhat. Például a *Gömb* típus-specifikációját nemcsak az előző alfejezetben bemutatott típus valósítja meg, hanem az is, ahol a gömböket közvetlenül négy valós számmal reprezentáljuk; az első három a középpont koordinátáit, az utolsó a sugarát adná meg. Ilyenkor nincs szükség a *Pont* típusra, az $l{:=}p{\in}g$ feladatot pedig közvetlenül egy valós számokkal felírt programmal kell implementálni. A *Pont* típusát is lehetne másként, például polár koordinátákkal reprezentálni, de ekkor természetesen más programot igényel két pont távolságának kiszámítása.

Nézzünk most egy olyan feladatot, amelynek megoldásához úgy vezetünk be egy új típust, hogy először a specifikációját adjuk meg, majd ehhez keresünk azt megvalósító típust.

7.2. Példa. Adott egy *n* hosszúságú *1* és *100* közé eső egész számokat tartalmazó tömb. Számoljuk meg, hogy ez a tömb hány különböző számot tartalmaz!

A feladatot számlálásra vezetjük vissza. A tömb egy eleménél akkor kell számolni, ha az még nem fordult elő korábban. Ezt egy lineáris kereséssel is eldönthetnénk, de ez nem lenne túl jó hatékonyságú megoldás, hiszen a tömböt újra és újra végig kellene vizsgálni. Ráadásul ilyenkor azt sem használnánk ki, hogy a tömb elemei *1* és *100* közé esnek. Számolhatnánk az *1* és *100* közé eső egész számokra, hogy melyik szerepel a tömbben, de hatékonyság szempontjából ez sem lenne lényegesen jobb az előzőnél (hosszú tömbök esetén egy kicsit jobb) hiszen a számlálás feltételének eldöntése itt is lineáris keresést igényel. Harmadik megoldás az, ha először előállítunk egy a tömb elemeiből képzett halmazt, és megnézzük, hogy ez a halmaz hány elemű. (Ebben a megoldásban a számlálás programozási tétele meg sem jelenik.)

$A = (a{:}\mathbb{N}^n, s{:}\mathbb{N})$

$Ef = (a=a')$

$$Uf = \left(Ef \wedge s = \left| \bigcup_{i=1}^{n} \{a[i]\} \right| \right)$$

A specifikáció szerint a megoldáshoz egy halmazra, pontosabban egy halmaz típusú adatra van szükségünk. Ebbe először sorban egymás után bele kell rakni a tömb elemeit, majd le kell kérdezni az elemeinek számát. A megoldó program ezért egy szekvencia lesz, amelynek első fele (az uniózás) az összegzés programozási tételére vezethető vissza, a második fele pedig a halmaz típus „elemszám” műveletére épül:

m..n	~	*1..n*
s	~	*h*
f(i)	~	$\{a[i]\}$
+,0	~	$\cup, \varnothing$

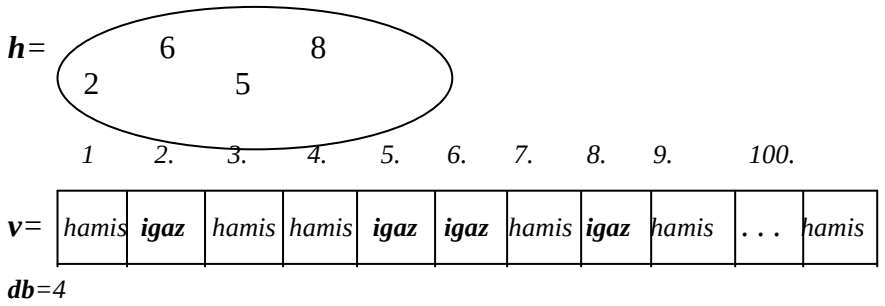
$h := \varnothing$
$i = 1 \,..\, n$
$h := h \cup \{a[i]\}$
$s := \lvert h \rvert$

A halmaztípus egy típusértéke egy olyan halmaz, amely elemei *1* és *100* közötti egész számok. Megengedett művelet egy halmazt üressé tenni, egy elemet hozzá uniózni, és az

elemeinek a számát lekérdezni. A halmaztípus típus-specifikációját formálisan az alábbi (értékhalmaz, műveletek) formában írhatjuk le:

$(\boldsymbol{2}^{\{1..100\}}, \{h{:=}\emptyset,\ h{:=}h\cup\{e\},\ \mathrm{s}{:=}|h|\})$.

Keressünk egy olyan konkrét típust, amely megvalósítja ezt a típus-specifikációt. Reprezentáljunk például egy *1* és *100* közé eső számokból álló halmazt *100* logikai értéket tartalmazó tömbbel.



7.1. Ábra
Egy halmaz és azt reprezentáló logikai tömb

A logikai tömbben az *e*-edik elem akkor és csak akkor legyen *igaz* értékű, ha a tömb által reprezentált halmaz tartalmazza az *e*-t. Vegyük még hozzá ehhez a tömbhöz külön azt az értéket, amely mindig azt mutatja, hogy hány igaz érték van a tömbben (ez éppen a reprezentált halmaz elemszáma). Ez a kapcsolat a logikai tömb és e darabszám között a típus-invariáns.

Itt a ρ reprezentációs függvény egy (logikai tömb, darabszám) párhoz rendeli azon egész számoknak a halmazát, amely a logikai tömb azon indexeit tartalmazza, ahol a tömbben igaz értéket tárolunk. Nyilvánvaló, hogy minden (tömb, darabszám) pár kölcsönösen egyértelműen reprezentál egy halmazt.

Térjünk most rá a típusműveletek implementálására. Az egyes műveleteket először átfogalmazzuk úgy, hogy azok ne közvetlenül a halmazra, hanem a halmazt reprezentáló tömb-darabszám párra legyenek megfogalmazva. Például a $h{:=}\emptyset$ feladatnak az a feladat feleltethető meg, hogy töltsük fel a logikai tömböt *hamis* értékekkel és nullázzuk le a darabszámot. Formálisan felírva az eredeti és a reprezentációra átfogalmazott feladatot:

$A = (h{:}\mathbf{2}^{\{1..100\}})$ $A = (v{:}\mathbb{L}^{100}, db{:}\mathbb{N})$

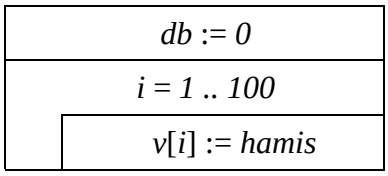
$Ef = (h=h') \rightarrow Ef = (\rho(v,db)=\rho(v',db')) =$

$= (v=v' \wedge db=db')$

$Uf = (h=\emptyset)$ $Uf = (\rho(v,db)=\emptyset) =$

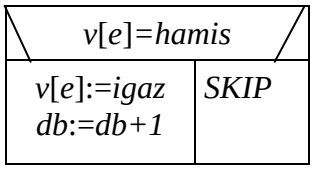
$= (\forall i \in [1..100]{:}v[i]=hamis \wedge db=0)$

Az átfogalmazott feladat elő- és utófeltételét kétféleképpen is felírtuk. Az első alak mechanikusan áll elő az eredeti specifikációból: a halmazt a tömb és a darabszám helyettesíti. A második alaknál már figyelembe vettük a reprezentáció sajátosságait. Könnyű belátni, hogy az átalakított feladatot az alábbi program megoldja:



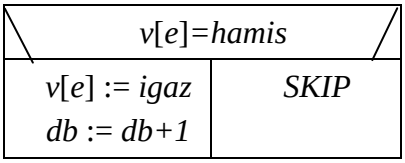
Ez a program bár közvetlenül nem halmazzal dolgozik, mégis abban az értelemben megoldja a $h{:=}\emptyset$ feladatot, amilyen értelemben egy halmazt egy logikai tömb és egy darabszám reprezentál. Ezt a fenti két specifikáció közötti kapcsolat szavatolja.

A másik két művelet esetében nem leszünk ennyire akkurátusak. (Meglepő módon ezek a műveletek jóval egyszerűbbek a fentinél, megvalósításuk nem igényel ciklust.) Könnyű belátni, hogy a $h{:=}h \cup \{e\}$ értékadást a $v[e]{:=}igaz$ értékadás váltja ki, hiszen amikor egy halmazba betesszük az e számot, akkor a halmazt reprezentáló tömb e-edik elemét i*gaz*-ra állítjuk. A típus-invariáns fenntartása érdekében azonban azt is meg kell nézni, hogy az e elem a halmaz szempontjából új-e, azaz $v[e]$ *hamis* volt-e. Ha igen, akkor a reprezentációt képező darabszámot is meg kell növelni. A megoldás egy elágazás.

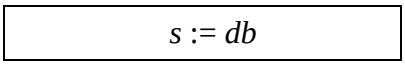
<table>
<tr><th></th><th>típusértékek</th><th>típusműveletek</th></tr>
<tr><td>típus-
specifikáció</td><td>$\boldsymbol{2}^{\{1..100\}}$
1 és 100 közé eső egész számokat tartalmazó halmazok</td><td>betesz
$h:=h\cup\{e\}$
legyen üres
$h := \varnothing$
elemszáma
$s:=\lvert h \rvert$</td></tr>
<tr><td></td><td>$\rho : \mathbb{L}^{100}\times\mathbb{N}\rightarrow\boldsymbol{2}^{\{1..100\}}$
$\rho(v,db) = \bigcup_{i=1}^{100}\{v[i]\}$
$I(v,db) = (db= \sum_{\substack{i=1\\ v[i]}}^{100} 1)$</td><td>$v$: $\mathbb{L}^{100}$, db:$\mathbb{N}$,
e: $\mathbb{N}$, l:$\mathbb{L}$, s:$\mathbb{N}$</td></tr>
<tr><td>típus</td><td>$\mathbb{L}^{100}\times\mathbb{N}$
logikai tömb és egy szám</td><td>betesz
 

legyen üres
$db := 0$
$\forall i\in[1..100]$: $v[i]:=hamis$</td></tr>
</table>

7.2. Ábra
Halmaz típusa



Ha a db-t nem vettük volna hozzá a reprezentációhoz, akkor az s:=|h| értékadást egy számlálással oldhatnánk meg. Így azonban elég helyette az s:=db értékadás.



A *7.2 ábrán* összefoglaltuk a halmaz típus specifikációjával és megvalósításával kapcsolatos megjegyzéseinket. A típus-specifikációban a műveletek feladatait értékadások formájában adtuk meg, a műveletek állapottere pedig kitalálható az értékadás változóinak típusából. A típus leírásban rövidített formában szerepel a „legyen üres” művelet programja.

7.3. Absztrakt típus

Adatabsztrakción azt a programtervezési technikát érjük, amikor egy feladat megoldása során olyan adatot vezetünk be, amely több szempontból is elvonatkoztat a valóságtól. Egyrészt elvonatkoztathat a feladattól, ha annak eredeti megfogalmazásában közvetlenül nem szerepel; csak a megoldás érdekében jelenik meg. Másrészt elvonatkoztathat a konkrét programozási környezettől, ahol majd az adat típusát le kell írni. Ezt a típust nevezzük ***absztrakt típusnak*.**

Absztrakt típus lehet egy típus-specifikációt, de egy olyan típus is,amelynek a reprezentációja és implementációja nem hatékony (ezért nem fogjuk ebben a formában alkalmazni) vagy nem-megengedett (lásd a megengedettség *3. fejezetben* bevezetett fogalmát) vagy hiányos (mert például nem ismert a műveleteinek programja, csak azok hatása, vagy még az sem).[15]. Egy absztrakt típustól mindössze azt várjuk, hogy világosan megmutassa az általa jellemzett adat lehetséges értékeit és azokon végezhető műveleteket, tehát az egyetlen szerepe az, hogy kijelöljön egy típus-specifikációt Ha ennek definiálásához nincs szükség reprezentációra,

[15] Ez magyarázza, hogy miért használják sokszor az absztrakt típus elnevezést magára a típus-specifikációra. Könyvünkben azonban az absztrakt típus tágabb fogalom, mint egy típus-specifikáció, hiszen tartalmazhat olyan elemeket is (például reprezentáció vagy típusművelet hatása, esetleg a részletes programja), amelyek nem részei a típus-specifikációnak.

akkor a típus-specifikációt *közvetlen*[16] módon adjuk meg, de ez a programozási gyakorlatban viszonylag ritka. Sokszor ugyanis nem tudjuk önmagában definiálni a típusérték-halmazt (úgy mint a gömbök vagy az *1* és *100* közötti elemekből képzett halmazok esetében tettük), csak annak egy reprezentációjával (példaként gondoljunk a pont fogalmára, amelyet mindannyian jól értünk, de definiálni nem lehet, csak valamilyen koordináta rendszer segítségével megadni). Általában a típusműveletek viselkedését is szemléletesebben lehet elmagyarázni, ha azok nem elvonatkoztatott típusértékekkel, hanem „megfogható" reprezentáns elemekkel dolgoznak. A „benne van-e egy pont a gömbben" művelet sokak számára érthetőbb, ha nem halmazelméleti alapon (benne van-e a pont a gömböt alkotó pontok halmazában), hanem geometriai szemlélettel (a pontnak a gömb középpontjától mért távolsága kisebb, mint a gömb sugara) magyarázzuk el. Ezért gyakoribb (lásd később *7.3, 7.4. példákat*) az, amikor egy típus-specifikációt *közvetett* módon írunk fel: adunk egy típust, ami kijelöli a típus-specifikációt. Ha ez a típus csak a típus-specifikáció kijelölésére szolgál, akkor tényleg nem fontos, hogy megengedett legyen, sőt az sem baj, ha hiányos. Az ilyen absztrakt típus csak abból a szempontból érdekes, hogy segíti-e megértetni a kijelölt típus-specifikációt, és nem számít, milyen ügyesen tudja reprezentálni a típusértékeket vagy mennyire hatékonyan implementálni az azokon értelmezett műveleteket.

Ha egy feladat megoldásánál absztrakt típust alkalmazunk, akkor később ezt olyan megengedett típussal (konkrét típussal) kell kiváltanunk, amely megvalósítja az absztrakt típus által kijelölt specifikációt, röviden fogalmazva ***megvalósítja az absztrakt típust***. Ha az absztrakt típus nemcsak egy típus-specifikációból áll, hanem rendelkezik reprezentációval és a típusműveleteket implementáló programokkal, vagy legalább azok hatásainak leírásával, akkor az absztrakt típust kiváltó konkrét típust úgy is elkészíthetjük, hogy közvetlenül az absztrakt típus elemeit (a reprezentációt és a programokat) valósítjuk meg. Ha a reprezentáció nem-megengedett vagy csak nem hatékony, akkor a reprezentáns elemeket próbáljuk meg helyettesíteni más konkrét elemekkel, azaz a reprezentációt reprezentáljuk. Ha a típusműveleteket megvalósító programok nem-megengedettek, akkor ezeket (és nem az eredeti típusműveleteket) az új reprezentáció alapján implementáljuk. Könnyű belátni, hogy az így kapott konkrét típus megvalósítja az absztrakt típus által leírt típus-specifikációt is.

Most mutatunk néhány, az itt vázolt folyamatot illusztráló úgynevezett adatabsztrakciós feladat-megoldást. A feladatok megoldása során olyan absztrakt típusokat vezetünk be, amelyek műveletei támogatják a feladat megoldását, annak ellenére, hogy a feladat eredeti megfogalmazásában semmi sem utal ezen adattípusok jelenlétére. Ezek a feladat szempontjából elvonatkoztatott típusok, ugyanakkor a programozási környezet számára is absztraktak. Emiatt gondoskodni kell a megvalósításukról is, azaz az absztrakt típusokat az őket megvalósító konkrét típusokkal kell helyettesíteni.

---

[16] Egy típus-specifikációt nemcsak a könyvünkben alkalmazott, úgynevezett elő- utófeltételes módszerrel adhatunk meg közvetlen módon. Más leírások is léteznek, ilyen például az úgynevezett algebrai specifikáció is. Ezek tárgyalásával itt nem foglalkozunk.

7.3. Példa. Adott egy *n* hosszúságú legfeljebb *100* különböző egész számot tartalmazó tömb. Melyik a tömbnek a leggyakrabban előforduló eleme!

A feladat megoldható egy maximum kiválasztásba ágyazott számlálással, ahol minden tömbelemre megszámoljuk, hogy hányszor fordult addig elő, és ezen előfordulás számok maximumát keressük. A *7.2. példa* nyomán azonban itt is bevezethetünk egy halmazszerű objektumot azzal a kiegészítéssel, hogy amikor egy elemet ismételten beleteszünk, akkor jegyezzük fel ennek az elemnek a halmazban való előfordulási számát (multiplicitását). Ezt a multiplicitásos halmazt zsáknak szokták nevezni. Miután betettük ebbe a zsákba a tömb összes elemét, akkor már csak arra van szükség, hogy megnevezzük a legnagyobb előfordulás számú elemét. Specifikáljuk a feladatot!

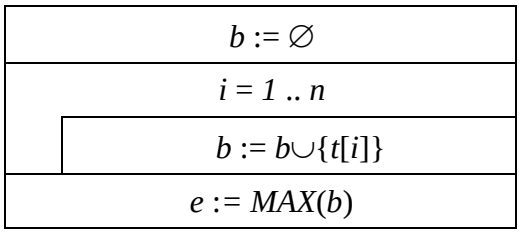
$$A = (t{:}\{1..100^{n}, b{:}Zsák, e{:}\{1..100\})$$

$$Ef = (t = t' \wedge n \geq 1)$$

$$Uf = (Ef \wedge b = \bigcup_{i=1}^{n} \{t[i]\} \wedge e = MAX(b))$$

A specifikációban használt unió és maximum számítás a zsák speciális műveletei lesznek. A feladat egy összegzésre vezethető vissza, amit a zsák maximális gyakoriságú elemének kiválasztása követ:

m..n	~	*1..n*
s	~	*b*
f(i)	~	$\{t[i]\}$
+,0	~	$\cup, \emptyset$



A zsáktípust absztrakt típussal definiáljuk. Egy számokat tartalmazó zsák ugyanis legegyszerűbben úgy fogható fel, mint számpárok halmaza, ahol a számpár első komponense a zsákba betett elemet, második komponense ennek az elemnek az előfordulási számát

tartalmazza. Reprezentáljuk tehát kölcsönösen egyértelmű módon egy zsákot a $\boldsymbol{2}^{\{1..100\}\times\mathbb{N}}$ hatványhalmaz egy (számpárokat tartalmazó) halmazával. Válasszuk a típus invariánsának azt az állítást, amely csak azokra a számpárokat tartalmazó halmazokra teljesül, amelyekben bármelyik két számpár első komponense különböző, azaz ugyanazon elem *1* és *100* közötti szám nem szerepelhet kétszer, akár két különböző előfordulás számmal a halmazban. A zsáktípus műveletei: a „legyen üres egy zsák" (b:=∅), a „tegyünk bele egy számot egy zsákba" (b:=b∪{e}), „válasszuk ki a zsák leggyakoribb elemét" (e:=*MAX*(b)), ahol b egy zsák típusú, az e egy *1* és *100* közötti szám típusú értéket jelöl.

A műveletek közül az „∪" művelet nem a szokásos halmazművelet, ezért ennek külön is felírjuk a specifikációját.

$$A = (b{:}\boldsymbol{2}^{\{1..100\}\times\mathbb{N}},\ e{:}\mathbb{N})$$
$$Ef = (\ b=b' \wedge e=e'\ )$$
$$Uf = (\ e=e' \wedge\ \exists(e,db)\in b' \rightarrow (b=b'-\{(e,db)\}\cup\{(e,db+1)\})$$
$$\wedge\ \neg\exists(e,db)\in b' \rightarrow (b=b'\cup\{(e,1)\})\ )$$

Ezzel a zsák absztrakt típusát megadtuk, kijelöltünk a zsák típus-specifikációját. Valósítsuk most meg az absztrakt zsáktípust!

Egy zsákot helyettesítő számpárok halmaza leírható egy *100* elemű természetes számokat tartalmazó tömbbel (v), ahol a v[e] azt mutatja, hogy az e szám hányszor van benne a zsákban. Ha v[e] nulla, akkor az e szám még nincs benne. Vegyük észre, hogy nem közvetlenül a zsákot próbáltuk meg másképp reprezentálni, hanem a zsákot az absztrakt típus által megadott korábbi reprezentációját.

Ezt a kétlépcsős megvalósítást alkalmazzuk a műveletek implementálásánál is. A $\boldsymbol{2}^{\{1..100\}\times\mathbb{N}}$-beli halmazokra megfogalmazott műveleteket próbáljuk megvalósítani, és ezzel közvetve az eredeti, a zsákokon megfogalmazott műveleteket is implementáljuk.

A zsákot reprezentáló halmaz akkor üres, ha minden *1* és *100* közötti szám egyszer sem (nullaszor) fordul benne elő, azaz a b:=∅ értékadást a $\forall e\in[1..100]$: v[e]:=*0* értékadás-sorozattal helyettesíthetjük. A b:=b∪{e} művelet implementálása még egyszerűbb: nem kell mást tenni, mint az e elem előfordulás számát megnövelni, azaz v[e]:=v[e]+*1*. A legnagyobb előfordulás számú elemet a reprezentáns v tömbre alkalmazott maximum kiválasztás határozza meg. Megjegyezzük, hogy ez még akkor is visszaad egy elemet, ha a zsák üres, mert ilyenkor minden elem nulla előfordulás számmal szerepel a tömbben.

A megoldásban jól elkülönül a típus megvalósítás három szintje (*7.3. ábra*). Legfelül van a zsák fogalma, amit az alatta levő absztrakt típus jelöl ki. Ezért nincs definiálva az absztrakt típus reprezentációs függvénye, csak a típus invariánsa. A típusműveletek állapottere kitalálható a műveletet leíró értékadás változóinak típusából. Az absztrakt típus alatt találjuk az azt megvalósító konkrét típust. Ennek invariáns állítása mindig igaz, ezért ezt külön nem tüntettük

fel. A műveleteket implementáló programok állapottere kitalálható a programok változóinak típusából.

	típusértékek	*típusműveletek*
típus specifikáció	*legfeljebb 100 különböző egész számot tartalmazó* ***Zsák***	***legyen üres***
		betesz
		maximum
	ρ:$\boldsymbol{2}^{\{1..100\}\times\mathbb{N}} \rightarrow$ ***Zsák*** $I(b)=\forall e_1, \quad e_2 \in b: \quad e_1.1 \neq e_2.1$	b:$\boldsymbol{2}^{\{1..100\}\times\mathbb{N}}$, e:{1..100}
absztrakt típus	$\boldsymbol{2}^{\{1..100\}\times\mathbb{N}}$*-beli* ***számpárok halmaza***	$b:=\emptyset$
		$b:= b\cup\{e\}$
		$e:=MAX(b)$
	$\rho : \mathbb{N}^{100} \rightarrow \boldsymbol{2}^{\{1..100\}\times\mathbb{N}}$ $\rho(v) = \bigcup_{i=1}^{100} \{(i, v[i])\}$	v: $\mathbb{N}^{100}$, e:{1..100}
konkrét típus	$\mathbb{N}^{100}$ ***előfordulás számok tömbje***	$\forall e \in [1..100]$: $v[e]:=0$
		$v[e]:=v[e]+1$
		$max, e := \boldsymbol{max}_{i=1}^{100} v[i]$

7.3. Ábra
Zsák absztrakt típusa és annak megvalósítása

7.4. Típusok közötti kapcsolatok

Egy bonyolultabb feladat megoldásában sok, egymással összefüggő típus jelenhet meg. A típusok közötti kapcsolatok igen sokrétűek lehetnek. A *7.1. példa* űrszondás feladatában például a *Pont* típus közvetlenül részt vett a *Gömb* típus reprezentációjában, azaz megjelent annak

szerkezetében, és ennek következtében a *Gömb* típus műveletét megvalósító program használhatja a *Pont* típus műveletét. Megtehetnénk azonban azt is, hogy a *Gömb* típust máshogy, például négy valós számmal reprezentáljuk. Ekkor a *Pont* típus nem vesz részt a *Gömb* típus reprezentációjában, ugyanakkor tagadhatatlanul ilyenkor is van valamilyen kapcsolat a két típus között: egyrészt minden gömb pontok mértani helyének tekinthető, másrészt a "benne van-e egy pont egy gömbben" művelet is összeköti ezt a két típust. A típusok közötti kapcsolatok, úgynevezett *társítások* (asszociációk) felfedése segíti a programtervezést, ugyanakkor egyszerűbbé, átláthatóbbá, hatékonyabbá teheti programjaink kódját is, különösen, ha a használt programozási nyelv támogatja az ilyen kapcsolatok leírását.

A típusok közötti társítási kapcsolatok lényegében különböző típusok közötti relációt jelent. Ezek lehetnek többes vagy bináris relációk. Bináris társítások esetén például azt lehet vizsgálni, hogy a reláció az egyik típus egy értékéhez hány típusértéket rendelhet a másik típusból. Ez *1-1* formájú, ha reláció kölcsönösen egyértelmű; *1-n* formájú, ha a reláció inverze egy függvény; és *m-n* formájú, ha tetszőleges nem-determinisztikus reláció. Az utóbbi két esetben további vizsgálat lehet az *n* és *m* értékének, vagy értékhatárainak meghatározása is. Ezek a vizsgálatok azt a célt szolgálják, hogy a valóságos dolgok közötti viszonyokat a feladatnak az adatabsztrakció után kapott modelljében is megjelenítsük.

Említettük már, hogy a *Gömb* típust másképpen is definiálhattuk volna, amikor közvetlenül négy valós számmal reprezentálunk egy gömböt. Ebben az esetben nem tartalmaz *Pont* típusú komponenst a *Gömb*, és ilyenkor annak az eldöntése, hogy a "benne van-e egy pont egy gömbben" művelet melyik típushoz tartozzon, nem magától értetődő. Ha a *Gömb* típus műveleteként valósítjuk meg, akkor két lehetőségünk is van. Vagy a gömb középpontját reprezentáló valós számokból kell egy pontot készíteni (ez nyilvánvalóan a *Pont* típus egy új művelete lesz), és ezután – a korábbi megoldáshoz hasonlóan – hivatkozunk a pontok távolságát meghatározó műveletre. Vagy közvetlenül az euklideszi távolság-képlettel dolgozunk, de ehhez előbb meg kell tudnunk határozni egy pont térbeli koordinátáit (ezek is a *Pont* típus egy új műveletei lesznek). Bizonyos programozási nyelvek úgy kerülik ki, hogy a *Pont* típushoz újabb műveleteket kelljen bevezetni és számolni kelljen azok meghívásával járó időveszteséggel, hogy bevezetik a *barátság* (friend) fogalmát, mint speciális társítási kapcsolatot. Ez lehetővé teszi, hogy egyik típus – mondjuk a *Gömb* típus – beleláthasson a másik – a *Pont* típus – reprezentációjába. Ilyenkor a "benne van-e egy pont egy gömbben" művelet az euklideszi távolság-képlet alapján közvetlenül megvalósítható; noha a művelet a *Gömb* típushoz tartozik, a vizsgált pont koordinátáit azonban a barátság miatt a Gömb típusban is közvetlenül látni. Még szebb az a megoldás, amely ezt a "benne van-e egy pont egy gömbben" műveletet kiveszi a *Gömb* típusból, azt egy független eljárásnak tekinti, de ezt az eljárást mind a *Pont*, mind a *Gömb* típus barátjává teszi, így azoktól közeli, de egyenlő távolságra helyezi el. A barátság kapcsolat segítségével lehetővé válik az eljárás számára mind egy gömb, mind egy pont reprezentációjára történő hivatkozás.

Típusok közötti kapcsolatnak tekinthetjük azt is, amikor egy típust (pontosabban az általa kijelölt típus-specifikációt) egy másik típussal megvalósítunk.

Ugyancsak típusok közötti kapcsolatot fogalmaz meg egy típus reprezentációja. Pontosabban, itt a típus és annak egy értéket helyettesítő elemi értékek típusai közötti reprezentációs kapcsolatról van szó.

A reprezentációs kapcsolat egyik vetülete az a tartalmazási kapcsolat, amelyik úgy tekint a reprezentált típusértékre, mint egy burokra, amely az őt reprezentáló elemi értékeket tartalmazza. Erre láttunk példát az űrszondás feladatban, ahol a *Pont* és a valós szám típusa építő eleme volt a *Gömb* típusnak: egy gömb reprezentációja egy pont és egy sugár volt. A tartalmazási kapcsolat tehát arra hívja fel a figyelmet, amikor egy típus részt vesz egy másik típus reprezentációjában.

A reprezentációs kapcsolat másik vetülete a reprezentáló típusok egymáshoz való viszonya, a típus-reprezentáció szerkezete. A reprezentáció szempontjából ugyanis lényeges az, hogy egy típusértéket helyettesítő elemi értékeknek mi az egymáshoz való viszonya: egy típusértéket különböző típusú elemi értékek együttese reprezentál-e, vagy különböző típusú értékek közül mindig az egyik, vagy azonos típusú értékek sokasága (halmaza, sorozata). Ez a reprezentáló típusok közötti szerkezeti kapcsolat fontos tulajdonsága a reprezentált típusnak.

Azokat a típusokat, ahol a helyettesítő elemek nem rendelkeznek belső szerkezettel, vagy az adott feladat megoldása szempontjából nem lényeges a belső szerkezetük, ***elemi típusoknak*** nevezzük. Ha egy típus nem ilyen, akkor ***összetett típusnak*** hívjuk. Egy *típus szerkezetén* a helyettesítő elemek gyakran igen összetett belső szerkezetét értjük, azaz azt, hogyan adható meg egy helyettesítő elem elemibb értékek együtteseként, és milyen kapcsolatok vannak ezen elemi értékek között.

Első hallásra ez nem tűnik korrekt meghatározásnak, hiszen a legegyszerűbb elemek is felbonthatók további részekre. A számítógépek világában célszerűnek látszik az elemi szintet a biteknél meghúzni, hiszen egy programozó számára az már igazán nem érdekes, hogy hány szabad elektron formájában ölt testet egy *1*-es értékű bit a számítógép memóriájában. A programkészítési gyakorlatban azonban még a bitek szintje is túlságosan alacsony szintűnek számít. A legtöbb feladat megoldása szempontjából közömbös az, hogy a programozási környezetben adott (tehát megengedett) típus értékei hogyan épülnek fel a bitekből. Például a feladatok túlnyomó többségében nem kell azzal foglalkoznunk, hogy a számítógépben egy természetes szám bináris alakban van jelen; nem fontos ismernünk az egész számok kódolására használt *2*-es komplemens kódot; nem kell tudnunk arról, hogy a számítógép egy valós számot *2* és *4* közé normált egyenes kódú mantisszájú és többletes kódú karakterisztikájú binárisan normalizált alakban tárol. (Legfeljebb csak a legnagyobb és a legkisebb ábrázolható szám, a fellépő kerekítési hiba vonatkozásában érdekelhet ez bennünket.) Ezért – hacsak a feladat nem követel mást – nyugodtan tekinthetjük a természetes számokat, az egész számokat, és a valós számokat is elemi típusoknak. Hasonló elbírálás alá esik a karakterek típusa, és a logikai típus is. Egy-egy feladat kapcsán természetesen más elemi típusok is megjelenhetnek. Ha egy típus

használata megengedett és nem kell hozzáférnünk a típusértékeket helyettesítő elemek egyes részeihez, akkor a típust elemi típusnak tekintjük. Hangsúlyozzuk, hogy ez a meghatározás relatív. Nem kell csodálkozni azon, ha egy típus az egyik feladatban elemi típusként, a másikban összetett típusként jelenik meg, sőt ugyanazon feladat megoldásának különböző fázisában lehet egyszer elemi, máskor összetett.

Az összetett típusok műveletei többnyire egy-egy típusérték helyettesítő elemének összetevő részeit kérdezik le, azokat módosítják, esetleg az összetevés módját változtatják meg. Nyilvánvaló, hogy az ilyen típusok műveleteinek elemzésénél, definiálásánál nagy segítséget jelent a helyettesítő elemek szerkezetének ismerete. Erre a típusoknak típus-specifikációval történő megadása nem alkalmas, mert az reprezentáció nélkül kevésbé szemléletes; ráadásul a programozó úgysem kerülheti el, hogy előbb utóbb gondoskodjon a típus reprezentálásáról, azaz döntsön a típusértékek belső logikai, majd fizikai szerkezetéről.[17]

Nézzünk meg most példaként néhány jellegzetes típusszerkezetet.

Egy típust *rekord* szerkezetűnek nevezünk, ha egy típusértékét megadott (típusérték-) halmazok egy-egy eleméből álló érték-együttes (rekord) reprezentálja. A $T = rec(m_1:T_1, \ldots , m_n:T_n)$ azt jelöli, hogy a T egy ***rekord (direktszorzat) típus***, azaz minden típusértékét egy $T_1 \times \ldots \times T_n$-beli elem (érték-együttes) reprezentálja. Ha t egy ilyen T típusnak egy értéke (rekord típusú objektum vagy röviden rekord), akkor a t reprezentánsának i-edik komponensére a $t.m_i$ kifejezéssel hivatkozhatunk (lekérdezhetjük, megváltoztathatjuk). Reprezentáljuk például a gömböket a középpontjukkal és a sugarukkal. Ekkor a *Gömb* típus rekord szerkezetű. Vezessünk be jól megjegyezhető neveket a reprezentáció komponenseinek azonosítására. Jelölje c a

[17] Az adatszerkezeteket középpontba állító vizsgálatoknál célszerű a típus szerkezetének definiálásánál egy típusértéket helyettesítő (reprezentáló) elemet olyan *irányított gráffal* ábrázolni, amelyben a csúcsokat elemi adatértékekkel címkézzük meg, az irányított élek pedig az ezek közti szomszédsági kapcsolatokat jelölik. Például egy sorozatszerkezetű típus egy értékét olyan gráffal írhatjuk le, ahol a csúcsokat sorba, láncba fűzzük, és mindegyik csúcsból (az utolsó kivételével) egyetlen irányított él vezet ki, amelyik a rákövetkező adatelemre mutat. A helyettesítő (reprezentáns) elemeknek irányított gráfokkal történő megjelenítése nem feltétlenül jelenti azt, hogy a típusnak konkrét programozási környezetben történő megvalósításakor az irányított gráfban rögzített szerkezetet hűen kell követni. A típus szerkezetének ilyen ábrázolása elsősorban a típus megadását, megértését szolgálja, ezért többnyire egy absztrakt típus definiálásához használjuk. Az absztrakt típus szerkezete, más szóval a típus absztrakt szerkezete lehetővé teszi a típus műveleteinek szemléletes bemutatását, és a műveletek hatékonyságának elemzését. Az irányított gráfok, mint absztrakt szerkezetek lehetőséget adnak a típusok különféle szerkezet szerinti rendszerezésére. Például az összes sorozatszerkezetű típus szerkezetét egy láncszerű irányított gráf, úgynevezett lineáris adatszerkezet jellemzi. Az absztrakt adatszerkezet jellemzésére meg szoktak még különböztetni ortogonális-, fa-, általános gráf-szerkezetű típusszerkezeteket is.

középpontot, *r* a sugarat. Ekkor a *g.c*–vel a *g* gömb középpontjára, *g.r*-rel pedig a sugarára hivatkozhatunk. Mindezen információt a *Gömb=rec*(*c:Pont,r:*$\mathbb{R}$) jelöléssel adhatjuk meg.

A típus *alternatív* szerkezetű, ha egy típusértékét megadott (típusérték-) halmazok valamelyikének egy eleme reprezentálja. A $T = alt(m_1{:}T_1; \ldots ; m_n{:}T_n)$ azt jelöli, hogy *T* egy ***alternatív (unió) típus***, azaz minden típusértékét egy $T_1 \cup \ldots \cup T_n$-beli elem reprezentál. Ha *t* egy ilyen *T* típusnak egy értéke (alternatív típusú objektum), akkor a $t.m_i$ logikai érték abban az esetben igaz, ha *t*-t éppen a T_i típus egyik értéke reprezentálja. Példaként tekintsük azt a típust, amely egy fogorvosi rendelőben egy páciensre jellemző adatokat foglalja össze. Mivel egy páciens lehet felnőtt és gyerek is, akiket fogorvosi szempontból meg kell különböztetnünk, hiszen például a gyerekeknél külön kell nyilván tartani a tejfogakat: *Páciens=alt*(*f:Felnőtt*;*g:Gyerek*). Természetesen egy ilyen példa akkor teljes, ha definiáljuk a *Felnőtt* és a Gyerek típusokat is. Ezek rekord típusok lesznek: *Felnőtt=rec*(*nev*:$\mathbb{K}^*$, *fog*:$\mathbb{N}$, *kor*:$\mathbb{N}$), *Gyerek=rec*(*nev*:$\mathbb{K}^*$, *fog*:$\mathbb{N}$, *tej*:$\mathbb{N}$, *kor*:$\mathbb{N}$). Ha *p* egy *Páciens* típusú objektum, akkor a $p.f$ állítással dönthetjük el, hogy a *p* páciens felnőtt-e; a *p.fog:=p.fog-1* értékadás pedig azt fejezi ki, hogy kihúzták a *p* egyik fogát.

Ha egy típus tetszőleges értékét sok azonos típusú elemi érték reprezentálja, akkor azt ***iterált (felsorolható, gyűjtemény)*** típusnak is szokták nevezni. Az elnevezés onnan származik, hogy a típusértéket reprezentáló elemi értékeket azok egymáshoz való viszonyától függő módon sorban egymás után fel lehet sorolni. Az elemi értékek kapcsolata a reprezentált típusérték belső szerkezetét határozza meg. Ez lehet olyan, amikor nincs kijelölve az elemi értékek között sorrendiség; a típusértéket elemi értékek közönséges halmaza reprezentálja. Ez a halmaz lehet multiplicitásos halmaz (zsák) is, amelyben megengedjük ugyanazon elem többszöri előfordulását. Lehet a reprezentáló elemsokaság egy sorozat is, amely egyértelmű rákövetkezési kapcsolatot jelöl ki az elemek között, de lehet egy olyan irányított gráf, amelynek csúcsai az elemi értékek, az azok közötti élek pedig sajátos, egyedi rákövetkezési relációt jelölnek. A *T=it*(*E*) azt a *T* iterált (szerkezetű) típust jelöli, amelynek típusértékeit az *E* elemi típus értékeiből építjük fel. A következő fejezetben számos nevezetes iterált szerkezetű típust fogunk bemutatni.

A típusszerkezethez kötődő típusok közötti kapcsolat a típusok közti szerkezeti rokonság. Két típust akkor nevezünk rokonnak, ha típusértékeik ugyanazon típus (esetleg típusok) elemeiből hasonló szerkezet alapján épülnek fel. Szerkezetileg rokon például egy egész számokat tartalmazó tömb és egy egész számokból álló sorozat. Rokon típusok műveletei természetesen különbözhetnek. Ilyenkor megkísérelhetjük az egyik típus műveleteit a rokon típus műveleteinek segítségével helyettesíteni. A rokonság gyakran egyoldalú, azaz a rokon típusoknak csak az egyike helyettesíthető a másikkal, fordítva nem; gyakran részleges, azaz nem az összes, csak néhány művelet helyettesíthető; esetenként névleges, mert a szerkezeti hasonlóság ellenére sem helyettesíthető egyik típus a másikkal.

Típusok között nemcsak szerkezeti rokonság, hanem műveleti rokonság is fennállhat. Erről akkor beszélhetünk, amikor két típusnak megegyeznek a típusműveletei. Például a *Gömb* típus

illetve egy *Kocka* típus ilyen típusműveleti rokonságot mutathat, ha mindkettőnél olyan műveleteket definiálunk, mint „add meg a test súlypontját”, „számítsd ki a térfogatot”, „benne van-e egy pont a testben”. Ezen műveletek specifikációja is azonos, csak a megvalósításuk tér el egymástól. Speciális típusműveleti rokonság a típus-altípus kapcsolat. Az altípus ugyanazokkal a műveletekkel rendelkezik, mint az általánosabb típus, csak azok értelmezési tartománya és értékkészlete szűkül. Például egész számok típusának egy altípusa az *1* és *10* közé eső egész számok típusa.

Az altípus fogalmának általánosításával juthatunk el az objektum orientált programozásban legtöbbet emlegetett típusok közötti kapcsolathoz, az öröklődéshez. Láttuk, hogy az altípus rendelkezik az általánosabb típus reprezentációjával és típus-műveleteivel, azaz „örökli” azokat az általánosabb típustól. Ezt az öröklést rugalmasabban is lehet érvényesíteni: meglevő típusból (esetleg típusokból) származtatunk egy új típust úgy, hogy egyenként megmondjuk, milyen tulajdonságokat (szerkezetet, szerkezet összetevőit, műveleteket) vesszünk át, más néven öröklünk a meglévő típus(ok)tól, és azokhoz milyen új tulajdonságokat teszünk hozzá. Az így létrejött új típust származtatott típusnak, a meglévő típusokat őstípusoknak nevezik.[18]

Az objektum elvű programtervezés például a *Gömb* és a *Pont* típus megadásánál észreveszi, hogy egy gömb általánosabb fogalom, mint a pont, hiszen a pont felfogható egy speciális, nulla sugarú gömbnek. Ilyenkor az úgynevezett megszorító öröklődést alkalmazva a *Pont* típust származtathatjuk a *Gömb* típusból, azaz azt mondjuk, hogy a *Pont* típus majdnem olyan, mint a *Gömb* típus, csak minden elemének a sugara nulla. A *Pont* típus tehát *Gömb* típus megszorításaként áll elő. Meg kell jegyezni, hogy ebben az esetben a gömb – a korábban bemutatott megvalósításával ellentétben – már nem egy pont és egy valós szám segítségével, hanem négy valós számmal van reprezentálva.

Ha a *Gömb* típuson definiálunk egy olyan műveletét, amely eldönti, hogy egy gömb tartalmaz-e egy másik gömböt, akkor ezt a műveletet speciálisan úgy is meg lehet majd használni, ha a másik gömböt egy pont helyettesíti (hiszen az is egy gömb). Ha a *Gömb* típus tartalmazná két gömb távolságát (pontosabban a gömbök középpontjának távolságát) kiszámoló műveletet, akkor ezt a művelet örökölné a *Pont* típus is, ahol azonban ez már megszorítva, két pont távolságának meghatározására szolgálna. Egyébként ugyanígy örökölhető a „benne van-e egy gömbben egy másik gömb” művelet is, amely a Pont típusra nézve a „benne van-e egy pontban egy másik gömb” vagy akár „benne van-e egy pontban egy másik pont” (ami lényegében az „azonos-e két pont” művelet) értelmet nyerne. A megszorító öröklődésnél elvben azt is el lehet érni, hogy az őstípusból származó bizonyos műveletek ne öröklődjenek a leszármazott típusra. Mivel a példában a *Pont* típus és a *Gömb* típus ugyanazon műveletekkel rendelkezik (hiszen a *Pont* örökli a *Gömb* egyetlen műveletét és nem vezet be mellé újat), ezért itt a *Pont* a *Gömb* altípusa is.

[18] Az objektum elvű programtervezés a típusok helyett az osztály elnevezést használja.

A megszorító öröklődés nemcsak az itt bemutatott specializáció mentén jöhet létre, hanem úgy, hogy az egymáshoz hasonló típusokat általánosítjuk, és ennek során elkészítjük azt az őstípust, amelyből a vizsgált típusok öröklődéssel származtathatók.

Létezik az objektum elvű programozás gyakorlatában egy másik, a fentiektől eltérő gondolkodás is, amely azért vezet be ősosztályokat, hogy az öröklődés révén a kód-újrafelhasználást támogassa. Ennek a gondolatnak jegyében mondhatjuk például azt, hogy egy gömböt leíró kód tartalmazza a pontot leíró kódot, ezért érdemes előbb a *Pont* típust megalkotni, és majd ebből származtathatjuk a *Gömb* típust. A származtatás során kiegészítjük egy pont reprezentációját még egy valós számmal (ez lesz a gömb sugara); átvesszük (örököljük) a két pont távolságát kiszámoló műveletet, amely most már két gömb középpontjainak távolságát számítja majd ki; és csak itt a gömbökre definiáljuk a "benne van-e egy pont egy gömbben" új műveletet. Az öröklődésnek ezt a fajtáját, amikor új tulajdonságokat veszünk hozzá az őstípushoz, analitikus öröklődésnek nevezik. Ebben a megközelítésben előbb a *Pont* típust kell megalkotni, és utána a *Gömb* típust, tehát nem a programtervezésnél eddig látott felülről lefelé haladó finomítási technikát, hanem ellenkezőleg, egy alulról felfelé haladó technikát alkalmazunk. Ez a szemlélet persze csak akkor kifizetődő, ha a programozási nyelv, amelyen implementáljuk majd a programunkat, rendelkezik az öröklődést támogató nyelvi elemekkel.

8. Programozási tételek felsoroló objektumokra

Ha egy adatot elemi értékek csoportja reprezentál, akkor az adat feldolgozása ezen értékek feldolgozásából áll. Az ilyen adat lényeges jellemzője, hogy az őt reprezentáló elemi értékeknek mi az egymáshoz való viszonya, a reprezentáció milyen jellegzetes belső szerkezettel rendelkezik. Számunkra különösen azok az adattípusok érdekesek, amelyek típusértékeit azonos típusú elemi értékek sokasága reprezentál. Ilyen például egy tömb, egy halmaz vagy egy sorozat. Ezeknek az úgynevezett gyűjteményeknek közös tulajdonsága, hogy a bennük tárolt elemi értékek egymás után felsorolhatók. Egy halmazból egymás után kivehetjük, egy sorozatnak vagy egy tömbnek egymás után végignézhetjük az elemeit. Éppen ezért az ilyen típusokat szokták felsorolhatónak (enumerable) vagy iteráltnak (iterált szerkezetűnek) is nevezni.

Felsorolni azonban nemcsak gyűjtemények elemeit lehet, hanem például egy egész szám valódi osztóit, vagy két egész szám által meghatározott zárt intervallum egész számait. Ennél fogva a felsorolható adat fogalma nem azonos a gyűjteménnyel, hanem annál általánosabb, az előbbi példákban szereplő úgynevezett virtuális gyűjteményeket is magába foglalja. A felsorolhatóság azt jelenti, hogy képesek vagyunk egy adatnak valamilyen értelemben vett első elemére ráállni, majd a soron következőre, az azt követőre, és így tovább, meg tudjuk kérdezni, van-e újabb soron következő elem (vagy van-e egyáltalán első elem), és lekérdezhetjük a felsorolás során érintett aktuális elemnek az értékét.

A felsorolást végző műveletek nem annak az adatnak a saját műveletei, amelynek elemeit felsoroljuk. Furcsa is lenne, ha egy egész szám (amelyiknek valódi osztóit kívánjuk felsorolni) alapból rendelkezne ilyen („vedd a következő valódi osztót” féle) műveletekkel. De egy egész számokat tartalmazó intervallumnak is csak olyan műveletei vannak, amivel az intervallum határait tudjuk lekérdezni, az intervallum egész számainak felsorolásához már egy speciális objektumra, egy indexre van szükség. Ráadásul egy intervallum felsorolása többféle lehet (egyesével vagy kettesével; növekvő, esetleg csökkenő sorrendben), ezeket mind nem lenne értelme az intervallum típusműveleteivel leírni. A felsorolást végző műveleteket mindig egy külön objektumhoz kötjük. Ha szükség van egy adat elemi érékeinek felsorolásra, akkor az adathoz hozzárendelünk egy ilyen felsoroló objektumot.

Egy felsoroló objektum feldolgozása azt jelenti, hogy az általa felsorolt elemi értékeket valamilyen tevékenységnek vetjük alá. Ilyen tevékenység lehet ezen értékek összegzése, adott tulajdonságú értékek megszámolása vagy a legnagyobb elemi érték megkeresése, stb. Ezek ugyanolyan feladatok, amelyek megoldására korábban programozási tételeket vezettünk be, csakhogy azokat a tételeket intervallumon értelmezett függvényekre fogalmaztuk meg, most viszont ennél általánosabb, felsorolóra kimondott változatukra lesz szükség. Ezt az általánosítást azonban könnyű megtenni, hiszen egy intervallumhoz nem nehéz felsorolót rendelni, így a korábban bevezetett intervallumos tételeket egyszerűen átfogalmazhatjuk felsoroló objektumra.

8.1. Gyűjtemények

Gyűjteményeknek (és itt nem foglalkozunk a virtuális gyűjteményekkel) az összetett szerkezetű adatokat nevezzük. Ezek legfőbb jellegzetessége, hogy reprezentációjuk elemi értékekből épül fel, azaz elemi értékeket tárol, amelyeket megfelelő műveletek segítségével be tudunk járni, fel tudunk sorolni. Leggyakrabban az iterált szerkezetű adatokat használjuk gyűjteményként, amelyek típusértékeit azonos típusú elemi értékek sokasága reprezentálja. Vizsgáljunk meg most néhány nevezetes gyűjteményt.

A ***halmaz (szerkezetű) típus*** típusértékeit egy-egy véges elemszámú 2^E-beli elem (E-beli elemekből képzett halmaz) reprezentálja. Egy halmaz típusú objektumnak a típusműveletei a halmazok szokásos műveletei lesznek. Értelmezzük: halmaz ürességének vizsgálatát ($h=\varnothing$, ahol h egy halmaz típusú értéket jelöl), halmaz egy elemének kiválasztását ($e:\in h$, ahol e egy elemi értéket hordozó segédváltozó), halmazból egy elem kivonását ($h:=h-\{e\}$), új elemnek a hozzáadását a halmazhoz ($h:=h\cup\{e\}$). A típus-megvalósítás szempontjából egyáltalán nem közömbös, hogy itt a szokásos unió illetve kivonás műveletével van-e dolgunk, vagy a megkülönböztetett (diszjunkt) unió illetve megkülönböztetett kivonással. Ez utóbbiak esetén feltételezzük, hogy a művelet megváltoztatja a halmazt, azaz unió esetén bővül (mert a hozzáadandó elem új, még nincs benne a halmazban), kivonás esetén fogy (mert a kivonandó elem benne van a halmazban). Ezeknek a műveleteknek az implementálása egyszerűbb, mert nem kell ezen feltételeket külön ellenőrizniük, igaz, a feltétel nem teljesülése estén abortálnak. A halmaz típust a *set*(E) jelöli.

Látjuk, hogy egy halmaz egy elemének kiválasztása ($e:\in h$) a nem-determinisztikus értékkiválasztással történik, amely ugyanazon halmazra egymás után alkalmazva nem ugyanazt az eredményt fogja adni. Be lehet vezetni azonban a determinisztikus elemkiválasztás műveletét ($e:=mem(h)$), amelyet ha ugyanarra a halmazra többször egymás után hajtjuk végre, akkor mindig ugyanazon elemét adja vissza a halmaznak. Az igazat megvallva, a halmaz szóba jöhető reprezentációi sokkal inkább támogatják ennek az elemkiválasztásnak a megvalósítását, mint a valóban véletlenszerű, nem-determinisztikus értékkiválasztást.

A ***sorozat (szerkezetű) típus*** típusértékeit egy-egy véges hosszú E^*-beli elem (E-beli elemekből képzett sorozat) reprezentálja, típusműveletei pedig a sorozatokon értelmezhető műveletek. Jelöljön a t egy sorozat típusú objektumot, amelyet egy sorozat reprezentál. Most a teljesség igénye nélkül felsorolunk néhány típusműveletet, amelyeket a sorozatra be szoktak vezetni. Ilyen egy sorozat hosszának lekérdezése ($|t|$), a sorozat valahányadik elemére indexeléssel történő hivatkozás (t_i ahol az i indexnek *1* és a sorozat hossza közé kell esni), egy elem törlése egy sorozatból (ha a sorozat nem üres) vagy egy új elem beszúrása. Magát a sorozat típust a *seq*(E) jelöli.

Speciális sorozattípusokhoz jutunk, ha csak bizonyos típusműveleteket engedünk meg. Ilyen például a ***verem*** és a ***sor***. A verem esetén csak a sorozat elejére szabad új elemet befűzni, a

sornál pedig csak a sorozat végére. Mindkettő esetén megengedett művelet annak eldöntése, hogy a reprezentáló sorozat üres-e. Mindkettőnél ki lehet olvasni a sorozat első elemét, és azt el is lehet hagyni a sorozatból. A ***szekvenciális outputfájl*** (jelölése: *outfile*(*E*)) két műveletet enged meg: üres sorozat létrehozását (*t*:=<>) és a sorozat végéhez új elem vagy elemekből képzett sorozat hozzáillesztését (jelölése: *t*:*write*(*e*) vagy *t*:*write*(<e_1, ... ,e_n>)). A ***szekvenciális inputfájlnak*** (jelölése: *infile*(*E*)) is egyetlen művelete van: a sorozat első elemének lefűzése, más szóval az olvasás művelete. Matematikai értelemben ezt egy olyan függvénnyel írhatjuk le, amely egy sorozat típusú objektumhoz (pontosabban az őt reprezentáló sorozathoz) három értéket rendel: az olvasás státuszát, a sorozat első elemét (ha van ilyen), és az első elemétől megfosztott sorozatot. Az olvasást az *st,e,t*:=*read*(*t*) értékadással, vagy rövidítve az *st,e,t*:*read* szimbólummal jelöljük. Az *st* az olvasás státusza. Ez egy speciális kételemű halmaznak (*Státusz* ={*abnorm, norm*}) az elemét veheti fel értékként. Ha a *t* egy üres sorozat, akkor az *st,e,t*:*read* művelet során az *st* változó az *abnorm* értéket veszi fel, a *t*-beli sorozat továbbra is üres marad, az *e* pedig definiálatlan. Ha a *t*-beli sorozat nem üres, akkor az *st,e,t*:*read* művelet végrehajtása után az *st* változó az *norm*-ot, az *e* a *t* sorozat első elemét, a *t* pedig az eggyel rövidebb sorozatot veszi fel.

Sorozat szerkezetűnek tekinthetjük a ***vektor típust***, vagy más néven egydimenziós tömböt. Ha eltekintünk egy pillanatra attól, hogy a vektorok tetszőlegesen indexelhetőek, akkor a vektor felfogható egy olyan sorozatnak, amelynek az elemeire a sorszámuk alapján lehet közvetlenül hivatkozni, de a sorozat hossza (törléssel vagy beszúrással) nem változtatható meg. Egy vektor típusához azonban a fent vázolt sorozaton kívül azt az indextartományt is meg kell adni, amely alapján a vektor elemeit indexeljük. A vektor típus tehát valójában egy rögzített hosszúságú sorozatból és egy egész számból álló rekord, amelyben a sorozat tartalmazza a vektor elemeit, az egész szám pedig a sorozat első elemének indexét adja meg. A vektor típust a *vec*(*E*) jelöli. A vektor alsó és felső indexének lekérdezése is éppen úgy típusműveletnek tekinthető, mint a vektor adott indexű elemére való hivatkozás. Ha *v* egy vektor, *i* pedig egy indexe, akkor *v*[*i*] a vektor *i* indexű eleme, amit lekérdezhetünk vagy megváltoztathatunk, azaz állhat értékadás jobb vagy baloldalán. Általánosan a *v* vektor indextartományának elejét a *v.lob*, végét a *v.hib* kifejezések adják meg. Ha azonban a vektor típusra az általános *vec(E)* jelölés helyett továbbra is a korábban bevezetett $E^{m..n}$ jelölést alkalmazzuk, akkor az indextartományra egyszerűen az *m* és *n* segítségével hivatkozhatunk. Világosan kell azonban látni, hogy itt az *m* és az *n* a vektor egyedi tulajdonságai, és nem attól független adatok.

A kétdimenziós tömbtípusra, azaz a ***mátrix típusra*** vektorok vektoraként gondolunk. Ennek megfelelően egy *t* mátrix *i*-edik sorának *j*-edik elemére a *t*[*i*][*j*] hivatkozik (ezt rövidebben *t*[*i,j*]-vel is jelölhetjük), az *i*-edik sorra a *t*[*i*], az első sor indexére a *t.lob*, az utolsóéra a *t.hib*, az *i*-edik sor első elemének indexére a *t*[*i*].*lob*, az utolsó elem indexére a *t*[*i*].*hib*. A mátrix típust a *matrix*(*E*) jelöli. (Ezek a műveletek általánosíthatóak a kettőnél több dimenziós tömbtípusokra is.) Speciális, de a leggyakrabban alkalmazott mátrix (téglalap-mátrix) az, amelynek sorai egyforma hosszúak és ugyanazzal az indextartománnyal indexeltek. Ezt jelöli az $E^{l..n \times k..m}$, ahol a

sorokat az [*l..n*] intervallum, egy sor elemeit pedig a [*k..m*] intervallum indexeli. Ha *k=1* és *l=1*, akkor az $E^{n\times m}$-jelölést is használjuk, és ilyenkor *n*m*-es mátrixokról (sorok száma *n*, oszlopok száma *m*) beszélünk.

Könyvünkben megengedettnek tekintjük mindazokat a típusokat, amelyeket megengedett típusokból az itt bemutatott típusszerkezetek segítségével készíthetünk.

8.2. Felsoroló típus specifikációja

A felsorolható adatoknak (vektornak, halmaznak, szekvenciális inputfájlnak, valódi osztóit felkínáló természetes számnak; tehát a valódi vagy virtuális gyűjteményeknek) az elemi értékeit egy külön objektum segítségével szoktuk felsorolni. Természetesen egy felsoroló objektumnak hivatkoznia kell tudni az általa felsorolt adatra, amelyen keresztül támaszkodik a felsorolt adat műveleteire, de ezen kívül egyéb segédadatokat is használhat.

Egy ***felsoroló objektum***[19] (enumerator) véges sok elemi érték felsorolását teszi lehetővé azáltal, hogy rendelkezik a felsorolást végző műveletekkel: rá tud állni a felsorolandó értékek közül az elsőre vagy a soron következőre, meg tudja mutatni, hogy tart-e még a felsorolás és vissza tudja adni a felsorolás során érintett aktuális értéket. Azt a típust, amely biztosítja ezeket a műveleteket, és ezáltal felsoroló objektumok leírására képes, ***felsoroló típusnak*** nevezzük.

Egy *t* felsoroló objektumra tehát definíció szerint négy műveletet vezetünk be. A felsorolást mindig azzal kezdjük, hogy a felsorolót a felsorolás során először érintett elemi értékre – feltéve, hogy van ilyen – állítjuk. Ezt általánosan a *t*:=*First*(*t*) értékadás valósítja meg, amit a továbbiakban *t.First()*[20]-tel jelölünk. Minden további, tehát soron következő elemre a *t.Next()* művelet (amely a *t*:=*Next*(*t*) rövidített jelölése) segítségével tudunk ráállni. Vegyük észre, hogy mindkettő művelet megváltoztatja a *t* felsoroló állapotát. A *t.Current()* művelet a felsorolás alatt kijelölt aktuális elem értéket adja meg. A *t.End()* a felsorolás során mindaddig hamis értéket ad vissza, amíg van kijelölt aktuális elem, a felsorolás végét viszont igaz visszaadott értékkel jelzi. Nem szükségszerű, de ajánlott e két utóbbi műveletet úgy definiálni, hogy ne változtassa meg a felsoroló állapotát. Mi a továbbiakban ezt következetesen be is tartjuk. Fontos kritérium, hogy a felsorolás vége véges lépésben (a *t.Next()* véges sok végrehajtása után) bekövetkezzék. A felsoroló műveletek hatását általában nem definiáljuk minden esetre. Például nem-definiált az, hogy a *t.First*() végrehajtása előtt (tehát a felsorolás kezdete előtt) illetve a *t.End()* igazra váltása után (azaz a felsorolás befejezése után) mi a hatása a

[19] Amikor a felsorolható adat egy gyűjtemény (iterált), akkor a felsoroló objektumot szokták bejárónak vagy iterátornak is nevezni, míg maga a felsorolható gyűjtemény a bejárható, azaz iterálható adat.

[20] A műveletek jelölésére az objektum orientált stílust használjuk: *t.First()* a *t* felsorolóra vonatkozó *First()* műveletet jelöli.

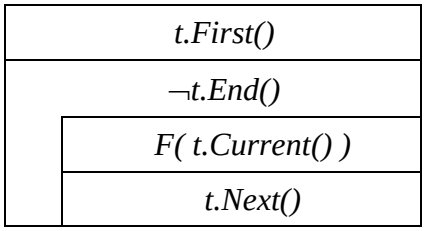
t.Next(), a *t.Current()* és a *t.End()* műveleteknek. Általában nem definiált az sem, hogy mi történjen akkor, ha a *t.First()* műveletet a felsorolás közben ismételten végrehajtjuk.

Minden olyan típust felsorolónak nevezünk, amely megfelel a felsoroló típus-specifikációnak, azaz implementálja a First(), Next(), End() és Current() műveleteket.

A felsoroló típust enor(*E*)-vel jelöljük, ahol az *E* a felsorolt elemi értékek típusa. Ezt a jelölés alkalmazhatjuk a típus értékhalmazára is. Egy felsoroló objektum mögé mindig elemi értékeknek azon véges hosszú sorozatát képzeljük, amelynek elemeit sorban, egymás után be tudjuk járni, fel tudjuk sorolni. Ezért specifikációs jelölésként megengedjük, hogy egy *t* felsoroló által felsorolható elemi értékre úgy hivatkozzunk, mint egy véges sorozat elemeire: a t_i a felsorolás során *i*-edikként felsorolt elemi érték, ahol i az *1* és a felsorolt elemek száma (jelöljük ezt |*t*|-vel) közé eső egész szám. Hangsúlyozzuk, hogy a felsorolható elemek száma definíció szerint véges. Ezzel tulajdonképpen egy absztrakt megvalósítást is adtunk a felsoroló típusnak: a felsorolókat a felsorolandó elemi értékek sorozata reprezentálja, ahol a felsorolás műveleteit ezen a sorozat bejárásával implementáljuk.

Egy felsoroló típus konkrét reprezentációjában természetesen nem szerepel ez a sorozat (kivéve, ha éppen sorozat bejárására definiálunk felsorolót), de mindig megjelenik valamilyen hivatkozás arra az adatra, amelyet fel akarunk sorolni. Ez az adat lehet egyetlen természetes szám, ha annak az osztóit kell előállítani, lehet egy vektor, szekvenciális inputfájl, halmaz, esetleg multiplicitásos halmaz (zsák), ha ezek elemeinek felsorolása a cél, vagy akár egy gráf, amelynek a csúcsait valamilyen stratégiával be kell járni, hogy az ott tárolt értékekhez hozzájussunk. A reprezentáció ezen az érték-szolgáltató adaton kívül még tartalmazhat egyéb, a felsorolást segítő komponenseket is. A felsorolás során mindig van egy aktuális elemi érték, amelyet az adott pillanatban lekérdezhetünk. Egy vektor elemeinek felsorolásánál ehhez elég egy indexváltozó, egy szekvenciális inputfájl esetében a legutoljára kiolvasott elemet kell tárolni illetve, azt, hogy sikeres volt-e a legutolsó olvasás, az egész szám osztóinak felsorolásakor például a legutoljára megadott osztót.

Egy felsoroló által visszaadott értékeket rendszerint valahogyan feldolgozzuk. Ez a feldolgozás igen változatos lehet; jelöljük ezt most általánosan az *F(e)*-vel, amely egy *e* elemi értéken végzett tetszőleges tevékenységet fejez ki. Nem szorul különösebb magyarázatra, hogy a felsorolásra épülő feldolgozást az alábbi algoritmus-séma végzi el. Megjegyezzük, hogy mivel a felsorolható elemek száma véges, ezért ez a feldolgozás véges lépésben garantáltan befejeződik.



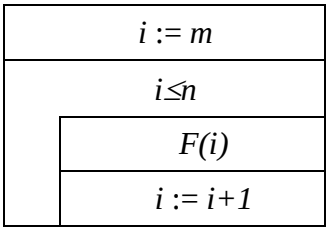
8.3. Nevezetes felsorolók

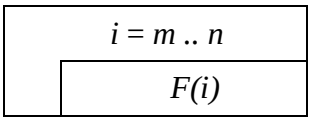
Az alábbiakban megvizsgálunk néhány fontos felsoroló típust, olyat, amelynek reprezentációja valamilyen nevezetes – egy kivétellel – iterált típusra épül. Vizsgálatainknak fontos része lesz, hogy megmutatjuk, hogyan specializálódik a fenti általános feldolgozást végző algoritmus-séma egy-egy konkrét felsoroló típusú objektum esetén. Természetesen minden esetben ki fogunk térni arra, hogy a vizsgált típus hogyan feleltethető meg a felsoroló típusspecifikációnak, azaz mi a felsorolást biztosító *First(), Next(), End()* és *Current()* műveletek implementációja.

Tekintsük először az ***egész-intervallumot felsoroló típust***. Itt egy [*m..n*] intervallum elemeinek klasszikus, *m*-től *n*-ig egyesével történő felsorolására gondolunk. Természetesen ennek mintájára lehet definiálni a fordított sorrendű vagy a kettesével növekedő felsorolót is. Az egész számok intervallumát nem tekintjük iterált szerkezetűnek, hiszen a reprezentációjához elég az intervallum két végpontját megadni, műveletei pedig ezeket az intervallumhatárokat kérdezik le. Ugyanakkor mindig fel lehet sorolni az intervallumba eső számokat. Az egész-intervallumra épülő felsoroló típus attól különleges számunkra, hogy a korábban bevezetett programozási tételeinket is az egész számok egy intervallumára fogalmaztuk meg, amelyeket valójában ezen intervallum elemeinek felsorolását végezték. Ennél fogva a korábban mutatott programozási tételeket könnyen tudjuk majd felsorolókra általánosítani.

Az egész-intervallumot felsoroló típus egy típusértékét egy [*m..n*] intervallum két végpontja (*m* és *n*) és az intervallum elemeinek felsorolását segítő egész értékű indexváltozó (*i*) reprezentálja. Az *i* változó az [*m..n*] intervallum aktuálisan kijelölt elemét tartalmazza, azaz implementálja a *Current()* függvényt. A *First()* műveletet az *i*:=*m* értékadás, a *Next()* műveletet az *i*:=*i+1* értékadás váltja ki. Az *i*>*n* helyettesíti az *End()* függvényt. (A *First()* művelet itt ismételten is kiadható, és mindig újraindítja a felsorolást, mind a négy művelet bármikor, a felsoroláson kívül is terminál.)

Ezek alapján a felsoroló objektum feldolgozását végző általános algoritmus-sémából előállítható az egész-intervallum klasszikus sorrendű feldolgozását végző algoritmus, amelyet számlálós ciklus formájában is megadhatunk.





Könnyű végiggondolni, hogy miként lehetne az intervallumot fordított sorrendben (*i:=n, i:=i-1, i<m*), vagy kettesével növekedően (*i:=m, i:=i+2, i>n*) felsorolni.

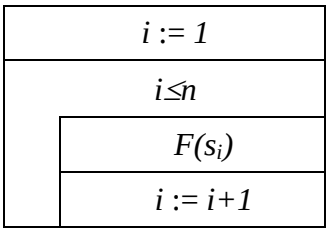
Magától értetődően lehet ***sorozatot felsoroló típust*** elkészíteni. (Itt is a klasszikus, elejétől a végéig tartó bejárásra gondolunk, megjegyezve, hogy más bejárások is vannak.) A reprezentáció ilyenkor egy sorozat típusú adat (*s*) mellett még egy indexváltozót (*i*) is tartalmaz. A sorozat bejárása során az *i* egész típusú indexváltozót az *1 .. |s|* intervallumon kell végigvezetni éppen úgy, ahogy ezt az előbb láttuk. Ekkor az s_i-t tekinthetjük a *Current()* által visszaadott értéknek, az *i:=1* a *First()* művelet lesz, az *i:=i+1* a *Next()* művelet megfelelője, az *i>|s|* pedig az *End()* kifejezéssel egyenértékű. (A *First()* művelet ismételten is kiadható, amely újraindítja a felsorolást, a *Next()* és *End()* bármikor végrehajtható, de a *Current()*-nek csak a felsorolás alatt van értelme.) Ezek alapján az *s* sorozat elemeinek elejétől végéig történő felsorolását végző programot az alábbi két alakban írhatjuk fel.

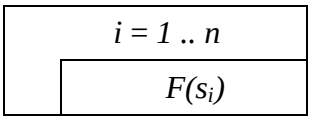
```
i := 1
i≤|s|
    F(si)
    i := i+1
```

```
i = 1 .. |s|
    F(si)
```

A ***vektort*** (klasszikusan) ***felsoroló típus*** a sorozatokétól csak abban különbözik, hogy itt nem egy sorozat, hanem egy *v* vektor bejárása a cél.[21] A bejáráshoz használt indexváltozót (jelöljük ezt most is *i*-vel) a bejárandó *v* vektor indextartományán (jelöljük ezt [*m..n*]-nel) kell végigvezetnünk, és az aktuális értékre a *v*[*i*] formában hivatkoznunk. Ennek értelmében az *v*[*i*]-t tekinthetjük a *Current()* műveletnek, az *i:=m* a *First()* művelet lesz, az *i:=i+1* a *Next()* művelet megfelelője, az *i>n* pedig az *End()* kifejezéssel egyenértékű. (A műveletek felsoroláson kívüli viselkedése megegyezik a sorozat felsorolásánál mondottakkal.) Egy vektor elemeinek feldolgozását az intervallumhoz hasonlóan kétféleképpen írjuk fel:

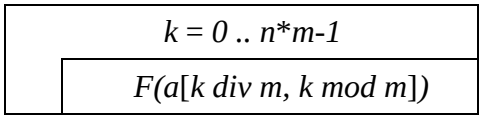
[21] Könyvünkben úgy tekintünk a vektor indextartományára, mint egy egész intervallumra. A vektor típus fogalma azonban általánosítható úgy, hogy indextartományát egy felsoroló objektum írja le. Ilyenkor a vektort felsoroló objektum műveletei megegyeznek az ő indextartományát felsoroló objektum műveleteivel egyet kivéve: a *Current()* műveletet a *v*[*i.Current()*] implementálja, ahol *i* az indextartomány elemeit felsoroló objektumot jelöli.



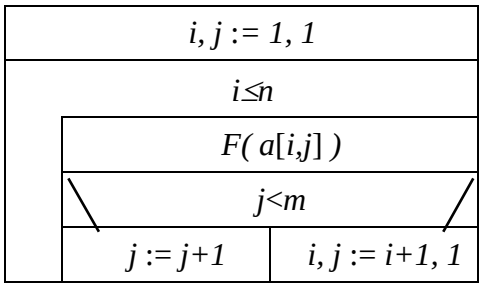


Mivel a mátrix vektorok vektora, ezért nem meglepő, hogy ***mátrixot felsoroló típust*** is lehet készíteni. A mátrix esetén az egyik leggyakrabban alkalmazott úgynevezett sorfolytonos felsorolás az, amikor először az első sor elemeit, azt követően a másodikat, és így tovább, végül az utolsó sort járjuk be.

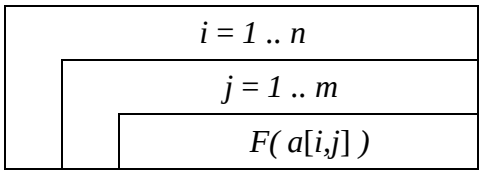
Egy *a* jelű *n*m*-es mátrix (azaz téglalap-mátrix) sorfolytonos bejárásánál a mátrixot felfoghatjuk egy *n*m* elemű *v* vektornak, ahol minden *1* és *n*m* közé eső *k* egész számra a *v*[*k*] = *a*[((*k-1*) *div m*) *+1*, ((*k-1*) *mod m*) *+1*]. Ezek után a bejárást a vektoroknál bemutatott módon végezhetjük. Egyszerűsödik a képlet, ha a vektor és a mátrix indextartományait egyaránt *0*-tól kezdődően indexeljük. Ilyenkor *v*[*k*] = *a*[*k div m, k mod m*], ahol a *k* a *0..n*m-1* intervallumot futja be. A mátrix elemeinek sorfolytonos bejárása igen egyszerű lesz, bár nem ez az általánosan ismert módszer.



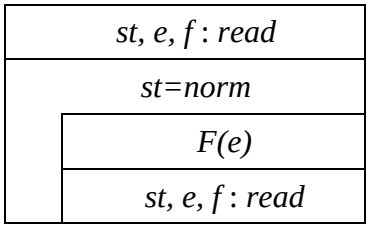
Mivel a mátrix egy kétdimenziós szerkezetű típus, ezért a bejárásához az előbb bemutatott módszerrel szemben két indexváltozót szoktak használni. (Más szóval a mátrix felsorolóját a mátrix és két indexváltozó reprezentálja.) Sorfolytonos bejárásnál az egyiket a mátrix sorai közötti bejárásra, a másikat az aktuális sor elemeinek a bejárására használjuk. A bejárás során *a*[*i,j*] lesz a *Current()*. Először a *a*[*1,1*]-re kell lépni, így a *First()* műveletet az *i,j*:*=1,1* implementálja. A soron következő mátrixelemre egy elágazással léphetünk rá. Ha a *j* bejáró még nem érte el az aktuális sor végét, akkor azt kell eggyel megnövelni. Ellenkező esetben az *i* bejárót növeljük meg eggyel, hogy a következő sorra lépjünk, a *j* bejárót pedig a sor elejére állítjuk. Összességében tehát az *IF*(*j<m*: *j*:=*j+1*; *j=m*: *i,j*:=*i+1,1)* elágazás implementálja a *Next()* műveletet. Az *End()* kifejezést az *i>n* helyettesíti. (Ez a megoldás könnyen általánosítható nem téglalap-mátrixra is, ahol a sorok eltérő hosszúak és eltérő indexelésűek.)



Ennek a megoldásnak a gyakorlatban jobban ismert változata az alábbi kétciklusos algoritmus. Mindkét változat ugyanabban a sorrendben sorolja fel az (*i,j*) indexpárokat az (*1,1*)-től indulva az (*n,m*)-ig. Az egyetlen eltérés a két változat között az, hogy leálláskor (amikor *i=n+1*) a fenti változatban a *j* értéke *1* lesz, míg a lenti változatban *m+1*.



A ***szekvenciális inputfájl felsoroló típusa*** egy szekvenciális inputfájllal (*f*), az abból legutoljára kiolvasott elemi értékkel (*e*), és az utolsó olvasás státuszával (*st*) reprezentál egy bejárót. A szekvenciális inputfájl felsorolása csak az elejétől végéig történő olvasással lehetséges.



A *First()* műveletet az először kiadott *st,e,f:read* művelet váltja ki. Az ismételten kiadott *st,e,f:read* művelet az *f.Next()* művelettel egyenértékű. Az *st,e,f:read* művelet az aktuális elemet az *e* segédváltozóba teszi, így a *Current()* helyett közvetlenül az *e* értékét lehet használni. Az *f.End()* az olvasás sikerességét mutató *st* státuszváltozó vizsgálatával helyettesíthető: *st=abnorm*. Mindegyik művelet bármikor alkalmazható, mert a *read* művelet mindig értelmezett, de a bejárást nem lehet ismételten elindítani vele, hiszen egy szekvenciális inputfájl logikai

értelemben csak egyszer járható be, minden olvasás elhagy belőle egy elemet. Mind a négy művelet minden esetben terminál.

A ***halmazt felsoroló típus*** reprezentációjában egy a felsorolandó elemeket tartalmazó *h* halmaz szerepel. Ha a $h=\varnothing$, akkor a halmaz bejárása nem lehetséges vagy nem folytatható – ez lesz tehát az *End()* művelet. Ha a $h\neq\varnothing$, akkor könnyen kiválaszthatjuk felsorolás számára a halmaznak akár az első, akár soron következő elemét. Természetesen az elemek halmazbeli sorrendjéről nem beszélhetünk, csak a felsorolás sorrendjéről. Ez az elemkiválasztás elvégezhető a nem-determinisztikus $e:\in h$ értékkiválasztással éppen úgy, mint a halmazokra korábban bevezetett determinisztikus elemkiválasztás ($e:=mem(h)$). Mi ez utóbbit fogjuk a *Current()* művelet megvalósítására felhasználni azért, hogy amikor a halmaz bejárása során tovább akarunk majd lépni, akkor éppen ezt, a felsorolás során előbb kiválasztott elemet tudjuk majd kivenni a halmazból. Ehhez pedig pontosan ugyanúgy kell tudnunk megismételni az elemkiválasztást. A *Next()* művelet ugyanis nem lesz más, mint a *mem(h)* elemnek a *h* halmazból való eltávolítása, azaz a $h:=h-\{mem(h)\}$. Ennek az elemkivonásnak az implementációja egyszerűbb, mint egy tetszőleges elem kivonásáé, mert itt mindig csak olyan elemet veszünk el a *h* halmazból, amely biztosan szerepel benne, ezért ezt külön nem kell vizsgálni. A *Next()* művelet is (akárcsak a *Current()*) csak a bejárás alatt – azaz amíg az *End()* hamis – alkalmazható. Láthatjuk azonban, hogy sem az *End()*, sem a *Current()*, sem a *Next()* művelet alkalmazásához nem kell semmilyen előkészületet tenni, azaz a felsorolást elindító *First()* művelet halmazok bejárása esetén az üres (semmit sem csináló) program lesz.

Ezen megjegyzéseknek megfelelően a halmaz elemeinek feldolgozását az alábbi algoritmus végzi el:

$h\neq\varnothing$
F(mem(h))
$h := h - \{mem(h)\}$

8.4. Programozási tételek általánosítása

A programozási tételeinket korábban az egész számok intervallumára fogalmaztuk meg, ahol egy intervallumon értelmezett függvénynek értékeit kellett feldolgozni. Ennek során bejártuk, felsoroltuk az intervallum elemeit, és mindig az aktuális egész számhoz tartozó függvényértéket vizsgáltuk meg. Az egész számok intervallumához – mint azt láttuk – elkészíthető egy klasszikus sorrendű felsoroló, amely éppen úgy képes az intervallumot bejárni, ahogy azt az intervallumos programozási tételekben láttuk.

Ezek alapján általánosíthatjuk az intervallumos programozási tételeinket bármilyen más felsoroló típusra is. A feladatokat ilyenkor nem intervallumon értelmezett függvényekre, hanem egy felsorolóra, pontosabban a felsoroló által felsorolt értékeken értelmezett függvényekre mondjuk ki. A megoldó programokban az intervallumot befutó *i* helyett a *Current()*, az *i*:=*m* értékadás helyett a *First()*, az *i*:=*i+1* értékadás helyett a *Next()*, és az *i*>*n* feltétel helyett az *End()* műveletet használjuk. Az így kapott általános tételekre aztán bármelyik konkrét felsorolóra megfogalmazott összegzés, számlálás, maximum vagy adott tulajdonságú elem keresése visszavezethető. Ezek között természetesen az intervallumon értelmezett függvényekkel megfogalmazott feladatok is, tehát a korábban megszerzett feladat-megoldási tapasztalataink sem vesznek el.

A programozási tételek ehhez hasonló általánosításaival egyszer-egyszer már korábban is találkoztunk. Már korábban is oldottunk meg olyan feladatokat, ahol nem intervallumon értelmezett függvényre, hanem vektorra alkalmaztuk a tételeinket. Ezt eddig azzal magyaráztuk, hogy a vektor felfogható egy táblázattal megadott egész intervallumon értelmezett függvénynek. Most már egy másik magyarázatunk is van. Az intervallum és a vektor rokon típusok: mindkettőhöz készíthető felsoroló. Egy felsorolóra megfogalmazott összegzés, számlálás, maximum vagy adott tulajdonságú elem keresés során pedig mindegy, hogy egy intervallumon értelmezett függvény értékeit kell-e sorban megvizsgálni vagy egy vektornak az elemeit, hiszen ezeket az értékeket úgyis a felsorolás állítja elő.

Hangsúlyozzuk, hogy az általános programozási tételekben nem közvetlenül a felsorolt elemeket dolgozzuk fel (adjuk össze, számoljuk meg, stb.), hanem az elemekhez hozzárendelt értékeket. Ezeket az értékeket bizonyos tételeknél (pl. összegzés, maximum kiválasztás) egy $f:E\rightarrow H$ függvény (ez sokszor lehet az identitás), másoknál (pl. számlálás, keresés) egy $\beta:E\rightarrow\mathbb{L}$ logikai függvény jelöli ki. Ennek következtében a feldolgozás során általában nem a *t.Current()* értékeket, hanem az *f(t.Current())* vagy β*(t.Current())* értékeket kell vizsgálni.

A specifikációk utófeltételében egy felsorolás elemeire hivatkozhatunk indexeléssel (lévén egy absztrakt sorozat a felsoroló hátterében), de a visszavezetés szempontjából ez sok lényegtelen elemet tartalmaz. Ezért egy új specifikációs jelölést is bevezetünk: az $e\in t$ kifejezéssel (amelyik nem halmazműveletet jelöl, hiszen a *t* nem egy halmaz, hanem egy felsoroló) azt a szándékot fejezzük ki, hogy az *e* változóban sorban egymás után jelenjenek meg a *t* felsorolás elemei.

1. Összegzés

Feladat: Adott egy *E*-beli elemeket felsoroló *t* objektum és egy $f:E\to H$ függvény. A *H* halmazon értelmezzük az összeadás asszociatív, baloldali nullelemes műveletét. Határozzuk meg a függvénynek a *t* elemeihez rendelt értékeinek összegét! (Üres felsorolás esetén az összeg értéke definíció szerint a nullelem: *0*).

Specifikáció:

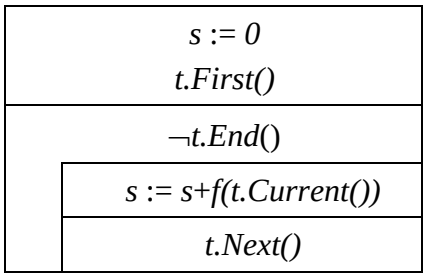
$A = (t{:}enor(E), s{:}H)$

$Ef = (t=t')$

$$Uf = \left(s = \sum_{i=1}^{|t'|} f(t'_i) \right) =$$

$$= \left(s = \sum_{e \in t'} f(e) \right)$$

Algoritmus:



2. Számlálás

Feladat: Adott egy *E*-beli elemeket felsoroló *t* objektum és egy $\beta:E\to\mathbb{L}$ feltétel. A felsoroló objektum hány elemére teljesül a feltétel?

Specifikáció:

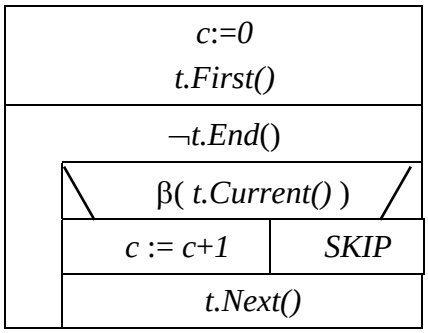
$A = (t{:}enor(E), c{:}\mathbb{N})$

$Ef = (t=t')$

$$Uf = \left(c = \sum_{\substack{i=1 \\ \beta(t'_i)}}^{|t'|} 1 \right) =$$

$$= \left(c = \sum_{\substack{e \in t' \\ \beta(e)}} 1 \right)$$

Algoritmus:



3. Maximum kiválasztás

Feladat: Adott egy *E*-beli elemeket felsoroló *t* objektum és egy $f:E \rightarrow H$ függvény. A *H* halmazon definiáltunk egy teljes rendezési relációt. Feltesszük, hogy *t* nem üres. Hol veszi fel az *f* függvény a *t* elemein a maximális értékét?

Specifikáció:

$A = (t:enor(E), max:H, elem:E)$

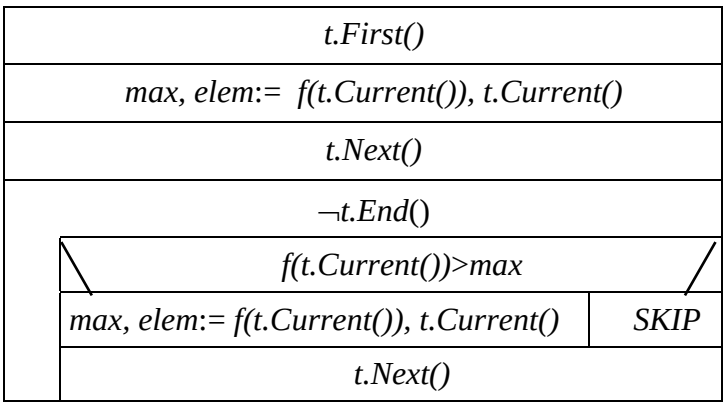
$Ef = (t=t' \wedge |t|>0)$

$$Uf = (\exists ind \in [1..|t'|]: elem = t'_{ind} \wedge max = f(t'_{ind}) = \max_{i=1}^{|t'|} f(t'_i)) =$$

$$= ((max, ind) = \max_{i=1}^{|t'|} f(t'_i) \wedge elem = t'_{ind}) =$$

$$= (max, elem = \max_{e \in t'} f(e))$$

Algoritmus:



4. Kiválasztás

Feladat: Adott egy *E*-beli elemeket felsoroló *t* objektum és egy β:$E \to \mathbb{L}$ feltétel. Keressük a *t* bejárása során az első olyan elemi értéket, amely kielégíti a β:$E \to \mathbb{L}$ feltételt, ha tudjuk, hogy biztosan van ilyen.

Specifikáció:

$A = (t{:}enor(E), elem{:}E)$

$Ef = (t=t' \wedge \exists i \in [1..|t|]: \beta(t_i))$

$Uf = (\exists ind \in [1..|t'|]: elem = t'_{ind} \wedge$

$\wedge \beta(elem) \wedge \forall j \in [1..ind-1]: \neg\beta(t'_j)$

$\wedge\ t = t'[ind+1..|t'|]) =$

Algoritmus:

t.First()
$\neg\beta$*(t.Current())*
t.Next()
elem:=*t.Current()*

$$= \left((ind,t) = \underset{ind \geq 1}{\textbf{select}}\ \beta(t'_{ind}) \wedge elem = t'_{ind} \right) = \left((elem,t) = \underset{elem \in t'}{\textbf{select}}\ \beta(elem) \right)$$

5. Lineáris keresés

Feladat: Adott egy *E*-beli elemeket felsoroló *t* objektum és egy β:$E \to \mathbb{L}$ feltétel. Keressük a *t* bejárása során az első olyan elemi értéket, amely kielégíti a β feltételt.

Specifikáció:

$A = (t{:}enor(E), l{:}\mathbb{L}, elem{:}E)$

$Ef = (t=t')$

$Uf = (l = \exists i \in [1..|t'|]: \beta(t'_i)$

$\wedge\ l \to \exists ind \in [1..|t'|]: elem = t'_{ind}$

$\wedge\ \beta(elem) \wedge$

$\wedge\ \forall j \in [1..ind-1]: \neg\beta(t'_j)$

$\wedge\ \ t = t'[ind+1..|t'|]) =$

Algoritmus:

l := hamis; t.First()
$\neg l \wedge \neg$*t.End()*
elem := t.Current()
l := β*(elem)*
t.Next()

$$= \left((l,ind,t) = \underset{i=1}{\overset{|t'|}{\textbf{search}}}\ \beta(t'_i) \wedge elem = t'_{ind} \right) = \left((l,elem,t) = \underset{e \in t'}{\textbf{search}}\ \beta(e) \right)$$

6. Feltételes maximumkeresés

Feladat: Adott egy *E*-beli elemeket felsoroló *t* objektum, egy β:$E \to \mathbb{L}$ feltétel és egy f:$E \to H$ függvény. A *H* halmazon definiáltunk egy teljes rendezési relációt. Határozzuk meg *t* azon elemeihez rendelt *f* szerinti értékek között a legnagyobbat, amelyek kielégítik a β feltételt.

Specifikáció:

$A = (t{:}enor(E), l{:}\mathbb{L}, max{:}H, elem{:}E)$

$Ef = (t=t')$

$Uf = (l = \exists i \in [1..|t'|]: \beta(t'_i) \wedge l \to \exists ind \in [1..|t'|]: elem = t'_{ind} \wedge \beta(elem) \wedge max = f(elem) \wedge$

$\forall i \in [1..|t'|]: (\beta(t'_i) \to f(t'_i) \leq max)) =$

$$= ((l, max, ind) = \max_{\substack{i=1 \\ \beta(t'_i)}}^{|t'|} f(t'_i) \wedge elem = t'_{ind}) =$$

$$= ((l, max, elem) = \max_{\substack{e \in t' \\ \beta(e)}} f(e))$$

Algoritmus:

<table>
<tr><td colspan="4">l:= hamis; t.First()</td></tr>
<tr><td colspan="4">¬t.End()</td></tr>
<tr><td>¬β(t.Current())</td><td colspan="2">β(t.Current()) ∧ l</td><td>β(t.Current()) ∧ ¬l</td></tr>
<tr><td>SKIP</td><td colspan="2">f(t.Current())>max</td><td rowspan="2">l, max, elem := igaz, f(t.Current()), t.Current()</td></tr>
<tr><td>max, elem:= f(t.Current()), t.Current()</td><td>SKIP</td></tr>
<tr><td colspan="4">t.Next()</td></tr>
</table>

IRODALOM JEGYZÉK

1. Dijkstra E.W., „A Discipline of Programming”,Prentice-Hall, Englewood Cliffs, 1973.
2. Dijkstra E.W. and C.S. Scholten, „Predicate Calculus and Program Semantics”, Springer-Verlag, 1989.
3. Fóthi Ákos: „Bevezetés a programozáshoz” ELTE Eötvös Kiadó. 2005.
4. Fowler M., „Analysis Patterns: Reusable Object Models”, Addison-Wesley, 1997.
5. Gries D., „The Science of Programming”, Springer Verlag, Berlin, 1981.
6. Gregorics, T., Sike, S.: „Generic algorithm patterns”, Proceedings of Formal Methods in Computer Science Education FORMED 2008, Satellite workshop of ETAPS 2008, Budapest March 29, 2008, p. 141-150.
7. Gregorics, T.: „Programming theorems on enumerator”, Teaching Mathematics and Computer Science, Debrecen, 8/1 (2010), 89-108
8. Hoare C.A., „Proof of Correctness of Data Representations}, Acta Informatica” (1972), 271-281.
9. Kondorosi K.,László Z.,Szirmay-Kalos L.: Objektum orientált szoftverfejlesztés, ComputerBooks, Budapest, 1997.
10. Sommerville I., „Software Engineering”, Addison-Wesley, 1983.
11. „Workgroup on Relation Models of Programming: Some Concepts of a Relational Model of Programming,Proceedings of the Fourth Symposium on Programming Languages and Software Tools, Ed. prof. Varga, L., Visegrád, Hungary, June 9-10, 1995. 434-44.”