A specifikáció hatékonyabb tanítása a felsőoktatás bevezető programozáskurzusain

Horváth Győző

horvath.gyozo@inf.elte.hu
ELTE IK

Absztrakt. A felsőoktatás kezdő programozáskurzusain a módszeres feladatmegoldás három nagy részből áll: 1) a feladat specifikálásából, 2) a megoldó program algoritmusából, és 3) az algoritmus implementációjából. A specifikálás során számos olyan döntést rögzítünk, amely segít a későbbi lépések leírásához, így a specifikáció a tervezés egy kiemelten fontos szakasza. A leíráshoz használt nyelv azonban elég nagy absztrakciót igényel, így a gyakorlatban a feladatmegoldásnak ezen része sokszor hiányos vagy hibás. Ebben a cikkben azt szeretnénk körüljárni, hogy milyen problémák merülnek fel a felsőoktatás kezdő programozáskurzusain használt specifikációs lépéssel kapcsolatban, és szeretnénk bemutatni egy olyan eszközt, amely segíthet az előbbi problémák enyhítésében, és a specifikációt egy szintre emelheti a feladatmegoldás során az algoritmizálási és kódolási lépésekkel.

Kulcsszavak: specifikáció, programozás, programozásoktatás, felsőoktatás

1. A specifikáció szerepe és formája

A felsőoktatásban számos megközelítést alkalmaznak a programozás oktatására a bevezető kurzusokon. Az Eötvös Loránd Tudományegyetem Informatikai Karán hagyományosan az algoritmusokkal ismerkednek meg először a hallgatók a procedurális paradigma keretében. A programozásoktatás során igyekszünk olyan eszközöket adni a hallgatók kezébe, amellyel *helyes* programokat, helyes algoritmusokat tudnak készíteni egy számítógép segítségét igénybe vevő probléma megoldására. Annak eldöntésére, hogy egy algoritmus helyesen oldja-e meg a feladatot, többféle módszer is létezik. Ehhez használhatunk informális eszközöket, mint pl. az elvárásaink szöveges megfogalmazása, vagy a fekete doboz tesztelési módszerből ismert bemeneti-kimeneti tesztesetek megadása, de ezeknek közös ismertetőjegye az, hogy nem tudják teljes mértékben garantálni az algoritmus működésének helyességét. Ez utóbbit csak formális eszközökkel, matematikai modellekkel tudjuk garantálni.

Az ELTE Informatika Karán Fóthi Ákos dolgozta ki a programozás *formális* leírását Dijkstra és Hoare munkáira alapozva [5], az így megszületett reláció modell halmazelmélettel és az elsőrendű logika eszközkészletével bizonyította, hogy adott algoritmikus szerkezetek valóban az elvárásoknak megfelelő megoldást adnak. A leírás középpontjában a feladat adat-, illetve állapottere áll, a vezérlési szerkezetek pedig ezen állapottérben átmeneteket hajtanak végre. Egy feladat leírása során tehát meg kell adni a feladatot leíró adatok, illetve változók állapotterét, az elsőrendű logika nyelvén állításokat kell megfogalmaznunk a kezdőállapotra, illetve ugyanígy meg kell adnunk, hogy adott kezdőállapothoz milyen végállapotot tekintünk elfogadottnak. Az ilyen módon megadott feladatleírásból pedig levezethetővé válnak az azt megoldó algoritmusok.

Az 1. ábrán arra láthatunk példát, hogy pl. a maximumkiválasztás tétel leírása hogyan néz ki ebben a formalizmusban:

$$A = \underset{m}{\mathbb{Z}} \times \underset{n}{\mathbb{Z}} \times \underset{i}{\mathbb{Z}} \times \underset{max}{\mathcal{H}}$$

$$B = \underset{m'}{\mathbb{Z}} \times \underset{n'}{\mathbb{Z}}$$

$$Q: (m = m' \wedge n = n' \wedge m \leq n)$$

$$R: \big(Q \wedge i \in [m..n] \wedge max = f(i) \wedge \forall j \in [m..n]: f(j) \leq f(i)\big)$$

1. ábra: A maximumkiválasztás tétel specifikációja a relációs modellben

Ezt a feladatleírást hívjuk *specifikációnak*, és szerepe a feladat tömör, precíz, formális leírása [1][2] abból a célból, hogy a megoldó program helyességét ellenőrizhesse vagy éppen biztosítsa (attól függ, melyik irányból nézünk a problémára). A specifikáció és algoritmus tehát nagyon szorosan összetartozik. A specifikációból levezethető az algoritmus, vagy fordítva, az algoritmus helyességét a specifikáció biztosítja. Elemei:

- az adat-, illetve *állapottér szerkezetének* a leírása, egyes leírásokban megkülönböztetve a
  - a *bemeneti adatok* szerkezetének leírását, és
  - a *kimeneti adatok* szerkezetének leírását;
- a bemeneti adatokra vonatkozó megszorítások logikai feltétele, az ún. *előfeltétel*;
- a helyesen összetartozó bemeneti-kimeneti adatok kapcsolatát leíró azonosan igaz logikai feltétel, az ún. *utófeltétel*.

A bevezető programozáskurzusokon azonban nem várhatjuk el a középiskolából frissen érkező hallgatóktól, hogy ennyire mély matematikai eszközökkel operáló formális eszközkészlettel minden esetben maguk lássák be a megalkotott algoritmusok helyességét. A helyes algoritmuskészítést úgy próbáljuk elérni, hogy a gyakran előforduló feladattípusok esetében elkészítjük azok általános specifikációit, levezetjük azok algoritmusait, minek következtében előállnak az adott feladattípus bizonyítottan helyes specifikáció-algoritmus párjai, majd a programozásoktatás keretében az aktuális feladatokat ezek analógiájára készítjük el. Ezt hívjuk *analóg programozásnak* [4].

Ezen a ponton két út kínálkozik. Az egyik az előállt algoritmusok helyességére fókuszál, és feladatmegoldásai során az ottani lépéseket veszi mintául, és adaptálja az aktuális feladatra. Kiindulási pontja tehát a típusfeladat algoritmusa, és a hozzá hasonló feladatokat az ott alkalmazott gondolatsorral és ötleteivel oldja meg, ún. *algoritmikus gondolkodás* útján. Mivel ötleteket használ, ezért az így előállt algoritmusok nagyobb változatossággal bírhatnak, és az ötletek feladatok széles körén lehet alkalmazni. Ezt az utat választotta az ELTE informatika tanár képzése, ahol középpontban az algoritmikus gondolkodás kialakítása, elsajátítása és átadása állt, hiszen az általános és középiskolai informatikaoktatásban erre van szükség. Ott a specifikáció formája így néz:

Be: $n \in \mathbb{N}, x_{1..n} \in \mathbb{H}^n, \leq: \mathbb{H} \times \mathbb{H} \to \mathbb{L}$

Ki: $max \in \mathbb{N}, maxért \in \mathbb{H}$

Ef: $n \geq 1$

Uf: $1 \leq max \leq n$ *és* $\forall i(1 \leq i \leq n): x_{max} \geq x_i$ *és* $maxért = x_{max}$

2. ábra: A maximumkiválasztási tétel az informatikatanár-képzésen.

A másik út a feladat formális leírását, a specifikációt tekinti kiindulási pontjának, és egy aktuális feladat leírását egy általános feladat leírásához illeszti abból a célból, hogy ahogy az általános feladat specifikációjából levezethető az általános algoritmus, úgy a konkrét feladat ahhoz hasonló

specifikációjából is hasonló lépéseken keresztül levezethető lenne a konkrét algoritmus, így elég megkeresni a konkrét és az általános feladat eltéréseit, és a megfelelő átnevezéseket kell csak megtenni. Az analóg programozásnak ezt a formálisabb, kötöttebb fajtáját hívjuk *visszavezetésnek*. Az előző maximumkiválasztás ebben így néz ki:

$$A = (m{:}\,\mathbb{Z}, n{:}\,\mathbb{Z}, ind{:}\,\mathbb{Z}, max{:}\,H)$$
$$Ef = (m = m' \wedge n = n' \wedge n \geq m)$$
$$Uf = \big(Ef \wedge ind \in [m..n] \wedge max = f(ind) \wedge \forall i \in [m..n]{:}\, max \geq f(i)\big)$$

3. ábra: A maximumkiválasztási tétel a programtervező informatikus alapképzésen.

Láthatjuk tehát, hogy mind a két út a maga módján a helyes megoldás elkészítésére törekszik, az első algoritmus-vezérelt módon, a második specifikáció-vezérelt módon teszi ezt. Ugyanakkor mind a két út megtartja a feladat megoldásának mindkét lépését, a specifikációt és algoritmust, csupán a hangsúly kerül máshova. Ez a hangsúlyeltolódás természetesen a specifikáció oktatására és számonkérésére is hatással van. Ott, ahol inkább az algoritmikus gondolkodás van középpontban, a specifikációnak a probléma megoldásában játszott szerepe könnyebben súlyt veszít, így az oktatás során is először annak bemutatása, közös elkészítése, értelmezése történik meg, és a hallgatóknak csak a szemeszter végi számonkérésekben kell ezek megalkotásáról számot adniuk. Ezzel ellentétben viszont ott, ahol a problémamegoldás a specifikáció megalkotásán áll vagy bukik, sokkal korábban előjön ennek nemcsak olvasása, hanem önálló létrehozása.

Ha a két analogikus út leírási módját szemléljük (ld. 2. és 3. ábra) és összehasonlítjuk az 1. ábrával, akkor azt láthatjuk, hogy a specifikáció jelölésmódja alapvetően megmaradt az egyes megközelítésekben, de az adaptációk során ezek a maguk céljára alakították ezt, ezért is találkozhatunk eltérő jelölésmódokkal és oly sokféle dialektussal a bevezető programozáskurzusunk által ajánlott szakirodalmakban [1][2][3][4][5]. Itt a formális elmélet éppen a kezdők számára való érthetőség miatt mára praktikummá szelídült, ahol sokkal fontosabb az adatok megfelelő szerkezetének átgondolása és az általános feladattípusok felismerése. A látható különbségek ellenére azonban nyilvánvaló az is, hogy közös tőről fakadnak, és ha ugyan a helyességbizonyítás a kezdő kurzusokban nem is jelenik meg, de a specifikáció jelen formája egyrészt jelzi ezt az örökséget, másrészt bármikor lehetőséget ad erre.

2. Tapasztalatok, problémák, célok a specifikáció oktatása során

A bevezető programozási kurzus praktikusan tekint a specifikációra, és a módszeres feladatmegoldás lépései [3] egyikének tekinti. Ennek során először meghatározzák, 1) mi a feladat (*specifikáció*), 2) megtervezik a megoldó absztrakt program lépéseit (*algoritmus*), 3) ezt egy választott programozási nyelv segítségével implementálják (*kódolás*), 4) ellenőrzik, hogy a feladat helyesen és hatékonyan oldja-e meg a feladatot (*tesztelés*), amennyiben nem, akkor 5) *megkeresik* és 6) *javítják* a hibát, 7) a program működéséről és felépítéséről pedig leírást készítenek (*dokumentáció*). Ugyan a feladatmegoldás minden lépése szükségszerű, mégis programozásoktatásunk kiemelt szerepet tulajdonít az első három lépésnek, a specifikációnak, az algoritmusnak és a kódolásnak, amely során a feladat meghatározásától a lépésekre bontott megoldáson át jut el a hallgató a működő kódhoz. Mindhárom lépésnek egyedi leíró nyelve van: a specifikáció a matematika formális eszközeit használja, az algoritmus leírásához egy újabb absztrakt nyelvre vagy jelölésrendszerre van szükség (nálunk pl. struktogramra), a kódoláshoz pedig az adott programozási nyelv eszközkészletét, működési elvét kell megérteni. Ez a hallgatóktól végeredményképpen három absztrakt nyelv megismerését és alkalmazását várja el a feladatmegoldás során.

Ezek közül is különösen a specifikáció bizonyul nehezen érthetőnek, megtanulhatónak és alkalmazhatónak a hallgatók számára. Az algoritmus lépésekre bontó jellege ugyanis közel áll a mindennapi gondolkodásunkhoz, és viszonylag kevés elemből építkezik, ezért ez könnyen megtanulható, a

programozási nyelv használata pedig vagy már eleve ismert számukra, vagy pedig az teszi először félelmetessé ugyan, de aztán később vonzóvá, hogy a kézzel fogható alkotás folyamata ott valósul meg. Ezzel szemben a specifikáció magas absztrakciós szintet igénylő formalizmusa sok, gyakran általuk nem ismert jelöléssel dolgozik, így eleve idegen számukra ez a leíró nyelv. Azért is különösen fájó pont ez a hallgatók számára, mert alapképzésünk átalakításával a bevezető programozáskurzuson a visszavezetés technikájával oldjuk meg a feladatokat, amelynek lényeges pontja a specifikáció értése és készítése. Ezzel a tervezés első fázisa, a specifikáció szerepe kulcsfontosságúvá vált, de ezáltal még kontrasztosabban tapasztalható a hallgatóktól elvárt tudás és a bennük lévő értetlenség közötti különbség.

A problémát elsősorban az alkalmazott formalizmus jelenti. Az elsőrendű logikai nyelvének és jelölésmódjának rutinszerű használata nem várható el a frissen érkező hallgatóktól. Sőt, sokszor maguk az oktatók sincsenek tisztában azzal, hogy egész pontosan milyen szabályrendszerrel dolgoznak. A logikai alapműveletek (és, vagy, nem), a kvantorok használata még egységes, de bizonyos esetekben, pl. sorozat elemeinek egyediségének meghatározásakor, ad-hoc jelölések kerülnek alkalmazásra.

Az elmúlt évek oktatói tapasztalata alapján elmondható, hogy a hallgatók többsége nem érti a specifikáció célját. Nem látja, hogy egy specifikáció utófeltétele egy olyan „szerződés", amelyekre a helyesen összetartozó be- és kimeneti adatok azonosan igaz logikai értéket adnak. További probléma, hogy egyszerű feladatoknál is, de különösen olyan esetekben, ahol több programozási mintát alkalmazó komplex problémákat kell megoldani, a hallgatói megoldások hemzsegnek a hibáktól. Ezek egy része „szintaktikai" hiba, amely során a fenti leírási forma sérül, például hiányoznak vesszők, hibásan kerülnek meghatározásra sorozatok, rekordok, a logikai kifejezések zárójelezése nem egyértelmű. Ezeket a hiányosságokat az értelmező oktató „kegyesen" kijavítja, hiszen alapvetően érthető a hallgató szándéka, de érezhető az is, hogy nem beszéli még ezt a nyelvet folyékonyan. Ennél nagyobb gond, hogy a felírt specifikációk szemantikusan is rosszak lehetnek, amiről a hallgató az alkotási folyamat során visszajelzés hiányában nem értesül, így nincs meg benne a tanulási folyamat során oly fontos visszacsatolási-megerősítési fázis. Oktatói szempontból is teher ezeknek a specifikációknak értelmezése, értékelése. A visszás helyzetet úgy is jellemezhetjük, hogy miközben a hallgatók még modellszinten sem tudják, hogy hogyan ábrázoljanak adatot, aközben olyan eszközök használatát várjuk el tőlük, amelyek a formális helyességbizonyításhoz valók.

Jogos tehát a hallgatók részéről az az igény, hogy ha már a specifikáció ennyire fontos részét képezi egy programozási feladat megoldásának, akkor legyen lehetőségük ezt a részterületet gyakorolni. Ennek a cikknek ez az igény volt a kiindulópontja, és az a felismerés, hogy erre egyelőre semmilyen eszköz nem áll rendelkezésre. Arra a kérdésre keresem a választ, hogy hogyan tudjuk a hallgatókat segíteni a helyes specifikáció írásában úgy, hogy az gyakorolható is legyen. Ebben a cikkben javaslatot teszek arra, hogyan lehet a specifikáció tanítását hatékonyabbá tenni, és hogyan biztosíthatjuk a hallgatók számára a gyakorlási lehetőséget.

3. Tervezési szempontok a specifikáció gyakoroltatásához

A módszeres feladatmegoldás első lépéseként megjelenő specifikációnak a benne rejtetten jelen lévő helyességbizonyítási potenciálon túl az alábbi fontos és *praktikus haszna* van az adott probléma feldolgozásának szempontjából:

- *modellezés*: a feladat adatainak absztrakt struktúrákba rendezése, adatábrázolás;
- *megoldási terv készítése*: az esetek többségében az állapottér adatai közötti összefüggés megadása egyben támpontot ad a megoldás módjára is (végrehajtható specifikáció);
- *tesztelés*: az előző két pont előkészítéseként érdemes konkrét bemeneti-kimeneti adatokat megadni, melyek később az elkészült implementáció tesztelésére is használhatóak.

A következőkben egy olyan eszközt szeretnék definiálni, amely a feladatmegoldás fenti praktikus szempontjait támogatja. Emellett fő cél az is, hogy a hallgatók *szintaktikailag* és *szemantikailag* is helyes specifikációkat tudjanak írni, miközben *értik* is, miről szól a specifikáció. Fontos szempont ez út során az, hogy a *specifikáció leírási módjának* meg kell felelnie azoknak a formáknak, amelyek a bevezető programozáskurzus szakirodalmában megtalálhatók. Mindenképpen egy olyan jelölőrendszer használata a cél, amely viszonylag könnyedén visszavezethető, visszaalakítható olyan formátumra, amely lehetőséget ad a formális helyességbizonyításra. Ameddig ez a kapcsolat megvan, addig használható az analóg programozás modellje.

Ezt követően azt kell átgondolnunk, hogy hogyan tudjuk a hallgatók számára *gyakoroltathatóvá* tenni a specifikációt. A gyakorlásnak az a célja, hogy a hallgató a megszerzett tudását alkalmazza egy ismeretlen feladaton. Ebből úgy tanul, ha a megoldására visszajelzést kap, lehetőleg minél hamarabb. Szerencsére feladatokat nagy számban találunk a különböző online feladatgyűjteményekben. Egy feladat kiadásakor a hallgató elkészítheti megoldását, amit elküld az oktatójának, aki azt kiértékeli és visszajelzést ad. Ennek a módszernek több hátránya van: egyrészt a hallgató nagyon lassan kap visszajelzést a feladatára, másrészt az oktatóra jut a kiértékelés és visszajelzésnek a terhe. Egy másik megoldás lehet, ha készítünk egy feladatgyűjteményt, amiben a megoldások is elolvashatóak. Ebben az esetben az oktató nincs az értékelési folyamatba bevonva, ezt a hallgatónak kell megtennie.

Sokkal jobb lenne, ha a specifikációk gyakoroltatását, kiértékelését egy *automata rendszerben* tudnánk elvégezni. Az első kérdés ezzel kapcsolatban, hogy a specifikációt milyen formában kellene megadni ennek a rendszernek. Lehetőségek:

- *Kézi rajz*: hagyományosan a specifikációk „papír-alapúak" éppen a bennük használt sok speciális jelölés miatt: különböző logikai karakterek, alsó és felső indexek, nagy szummák, alá-fölé pozícionált elemekkel. Egy kézi rajzot azonban automatikusan kielemezni túlságosan bonyolult feladat.
- *Egyenletszerkesztők*: szerencsére szövegszerkesztőkben egyenletszerkesztők segítségével jól megadhatók ezek a formulák. Viszont az ilyenek szerkesztése vagy túl körülményes, vagy egy újabb speciális nyelv megtanulását igényli.
- *LaTeX, AsciiMath*: egyszerű szöveges formátumok, ahol kódszerűen, „laposan" lehet megadni a kifejezéseket, viszont továbbra is gond velük, hogy egy újabb nyelv elsajátítása szükséges hozzájuk.

A fentiek közül az utolsó kettő ad lehetőséget automatikus kiértékelésre, mivel kötött formátumuk van, és csak az utóbbi olyan, ami könnyedén beadható a gép számára a kódszerű formája miatt, viszont új nyelv megtanulását teszi szükségessé. Ezek alapján az tűnik jó megoldásnak, ha *saját feladat-specifikus nyelv*et (DSL) készítünk, amely gyakorlatilag megfelel a „kézi" megoldásoknak, csak az egyes elemek egy sorban, laposan, kódszerűen jelennek meg. Ilyetén megvan a szabadság, hogy azt minél inkább az előzményekhez és az igényekhez alakítsam, megőrizve a kapcsolatot a specifikáció formális hátterével és szándékával. Az ebben a nyelvben leírt specifikáció szintaktikusan ellenőrizhető, és a hallgató számára azonnali visszajelzés adható erről.

A szabad szöveges megadás azért is fontos, mert ugyanannak a feladatnak nagyon sokféle megoldása lehet. Mindenképpen támogatni kell, hogy az eszköz szabadságot adjon a gondolatoknak és a kreatív folyamatoknak. Ez nagy terhet ró a *szemantikus ellenőrzés*re. Honnan tudjuk, hogy egy kiadott feladatra jó-e a megoldás? A kérdés érdekessége, hogy egy logikai bukfencet tartalmaz: elvileg a specifikáció mondja meg, mikor helyes a feladat, de mi most éppen arra vagyunk kíváncsiak, hogy adott feladathoz mikor helyes a specifikáció. A specifikáció helyességének ellenőrzésére formális és informális módszerek állnak rendelkezésre:

- *Hivatalos specifikáció*: a feladathoz készíthetünk hivatalos megoldást. Ennek szövegszerű összehasonlítását azonnal elvethetjük. A gond az, hogy két eltérő specifikáció azonosságának belátása nagyon nehéz. Ehhez a feladat szimbolikus végrehajtására lenne szükség formális módszerekkel.

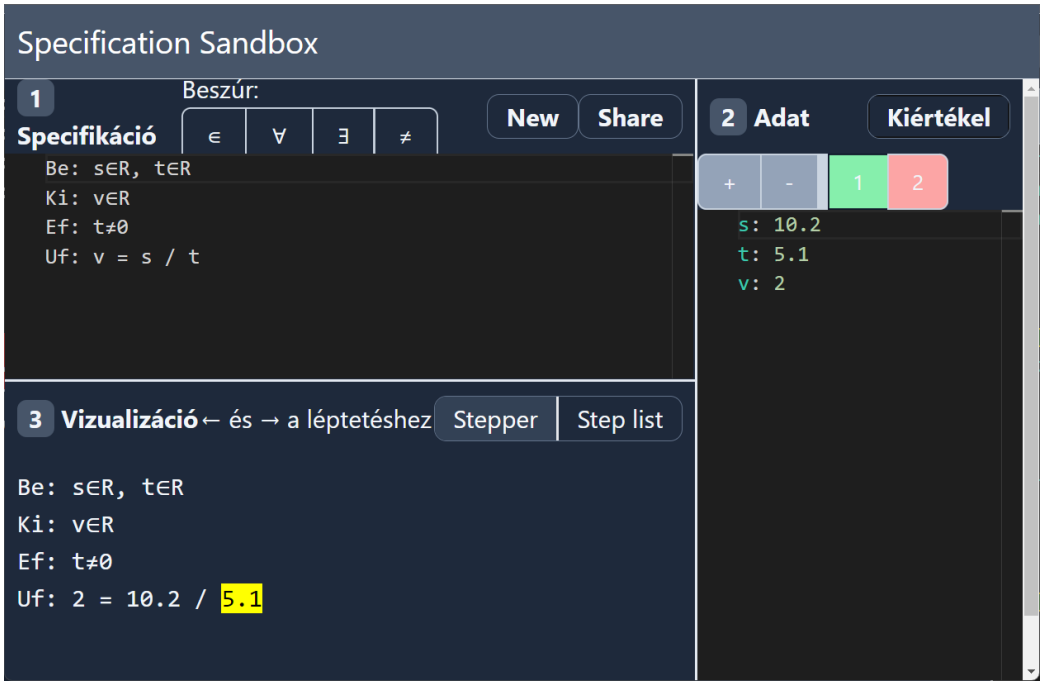
- *Automatikus bizonyító rendszerek*: formális megoldásként igénybe vehetnénk az elsőrendű logikai nyelvén értő automatikus bizonyító rendszereket. Azon túl, hogy ezek nagyon komplex rendszerek, a specifikációkat át kellene fordítani ezen rendszerek nyelvére, ami megint nem kis feladat lenne.
- *Fekete doboz tesztelés*: informális módszer, tulajdonképpen az alapján döntjük el, hogy a specifikáció helyes-e, hogy az általunk helyesnek talált bemeneti-kimeneti adatpárokra az utófeltétel igaz értéket ad. Ha mindegyikre azt adja, akkor valamekkora valószínűséggel helyes az utófeltétel. A módszer általánosan alkalmazott kódok tesztelésére, így itt is érvényes a használata. Az elméleti hátterét pedig az elsőrendű logika behelyettesítési szabálya adja: azaz, ha egy rendszer általánosan igaz, akkor konkrét értékeket behelyettesítve is igazat fog adni.

Nagyon fontos, hogy az így megalkotott nyelv *támogassa a bevezető kurzusban használatos elemeket*. A teljesség igénye nélkül néhány példa:

- adatok megadása, eleme, halmazok, sorozatok (tömbök), direkt szorzatok (rekordok) akár egymásba ágyazott megadása
- matematikai operátorok, =, ≠, <, >, +, -, *, /, div, mod, zárójelezés
- alapvető matematikai függvények
- egzisztenciális és univerzális kvantorok
- állapottér bővítés
- programozási minták (tételek) rövidített formájának használata
- függvények definiálása, meghívása, rekurzív függvények

Végezetül fontos elvárás, hogy az eszköz lehetővé tegye a specifikáció „működésének” megértését a kiértékelés *lépésekre bontott vizualizációjával*. A specifikációt egy szerződésnek tekintjük a megrendelő és a programozó között, amely a matematika formális nyelvén írja le, hogy mit várunk a feladattól, másképpen, mikor jó egy feladat. Ezt praktikusan úgy tehetjük meg, hogy a programba bemeneti és az általa visszaadott kimeneti adatokat behelyettesítjük az utófeltételbe, és megnézzük igaz lesz-e a logikai kifejezés eredménye. A behelyettesítés utáni kiértékelési fázisokat lehetne lépésenként bemutatni.

4. A specifikációt gyakoroltató eszköz bemutatása, megvalósítás részletezése



4. ábra: A specifikációs eszköz (feladat).

A fenti szempontoknak megfelelő eszközt készítettem el egy kliensoldali webes alkalmazás formájában [7]. Az eszköz fő területe a bal oldalon található szerkesztő. Ide kell beírni az általam definiált specifikációs nyelvet. A nyelv főbb funkciói az alábbi példákban láthatók:

```
Be: n∈N, névsor∈Tanuló[1..n],
    Tanuló=Név x Mag, Név=S, Mag=N
Ki: mon∈L
Ef: n>=1 és ∀i∈[1..n-1]:(névsor[i].név<=névsor[i+1].név)
Uf: n=1 -> mon=igaz és
    n>1 -> mon=MIND(i=1..n-1, névsor[i].mag<=névsor[i+1].mag)
```

1. példa: Specifikáció annak eldöntésére, hogy az egymást követő tanulók magasságai monoton növekednek-e(feladat).

```
Be: n∈N, mag∈N[1..n]
Ki: van∈L, k∈N, v∈N
Fv: szigetkezdet:N->L,
    szigetkezdet(i)={hamis, ha i=1;
                     mag[i-1]=0 és mag[i]>0 egyébként}
Fv: szigetvég:N->L,
    szigetvég(i)={hamis, ha i=n;
                  mag[i+1]=0 és mag[i]>0 egyébként}
Fv: keresvége:N->L x N,
    keresvége(i)=KERES(j=i..n, szigetvég(j))
Fv: táv:N->N, táv(i)=keresvége(i).ind - i
Ef: -
Uf: (van,k,)=MAX(i=1..n, táv(i), szigetkezdet(i) és keresvége(i).van)
 és van -> v=keresvége(k).ind
```

2. példa: Specifikáció a leghosszabb sziget megállapítására **(**feladat**)**.

Ezt a nyelvet PEG (Parsing Expression Grammars) [6] formátumban adtam meg. A forráskód bejárása során egy absztrakt szintaxis fa épül fel, melyet később a kiértékelés során járok be az aktuális adatokkal. A bejárás során a bemeneten, kimeneten és a segédadatokban található adatok típusosan létrejönnek és elérhetővé válnak a bemeneten és kimeneten kiértékelődő logikai kifejezés számára. Kétféle típust definiálhatunk: tömböt és rekordot, ezeket a program rekurzívan járja be és építi fel belőle a típus-fát. Ha a bejárás során hibát találunk, akkor azt a szerkesztő alatt lehet látni.

A szemantikai helyesség eldöntéséhez szükség van tesztadatokra. Ezeket a jobb oldali részben lehet megadni YAML[1] formátumban. A bejárás során felépült típus-fából JSON[2] sémát építünk, így a jobb oldali szerkesztő a specifikációnk elvárásainak megfelelő hibaüzeneteket tud adni, ha a megadott adat nem megfelelő szerkezetű. A megadott adatokkal automatikusan ellenőrzésre kerül a specifikáció helyessége. Ha ezek szemantikai hibát okoznak, mint pl. számot szöveggel hasonlítunk össze stb., akkor ugyancsak hibaüzentet kapunk. Ha a behelyettesítés sikerrel járt, még akkor is kaphatunk hibaüzenetet, ha az előfeltétel vagy az utófeltétel kiértékelése nem lett igaz. Így akár érvénytelen teszteket is meg tudunk adni és ellenőrizni. Ha a szintaktikai és szemantikai ellenőrzés is sikerrel járt, akkor a teszt számának háttere zöldre vált.

Ekkor megnézhetjük az utófeltétel kiértékelését a „Kiértékel” gombra kattintva. Bal oldalt alul jelenik meg ekkor a specifikáció, sárgával jelezve az aktuális kiértékelési lépést. A léptetést a bal-jobb billentyűvel érhetjük el. A kiértékelés végén az utófeltételnek igaz vagy hamis értéket kell mutatnia.

A megírt specifikáció megosztható. A „Share” gombra kattintva az eszköz a szerkesztőmezők tartalmából egy tömörített szöveges reprezentánst készít, amit az URL-be is belegenerál, másrészt a vágólapra is odamásol. Ezzel nagyon egyszerűvé válik az előkészített vagy elkészített megoldások megosztása, ellenőrzése.

---

[1] A YAML egy felhasználóbarát adatleíró nyelv (YAML Ain't Markup Language, https://yaml.org/)

[2] A JSON a JavaScript nyelv tömb és objektumliterál formátumára épülő adatleíró nyelv (JavaScript Object Notation, https://www.json.org/)

5. Módszertani elemzés, elvárások

Az így elkészített specifikációs eszköz segítségével különböző típusú feladatokat adhatunk ki, amelyek a feladatmegoldás és specifikálás különböző aspektusaira terelhetik a figyelmet. Ezeket a feladattípusokat órán és házi feladatként is alkalmazhatjuk.

- *Üres feladat*: a leggyakoribb feladattípus, mely során a feladat szövegesen kerül kiadásra, és a hallgatónak helyes megoldást kell hozzá készítenie. Ennek során a hallgató elkészíti a specifikációt, mikor az szintaktikusan helyes, felvesz tesztadatokat, és addig javítja a specifikációt, amíg a teszt zöld nem lesz. Ebben az esetben egyébként ajánlott a tesztekkel kezdeni. A mintaadatok ugyanis segítenek képet alkotni az adatok szerkezetéről, így a bemenet és kimenet, sőt az előfeltétel leírása is könnyebben adódik. Érdemes több tesztet előkészíteni, így a specifikáció helyességéről gyorsabban meg tudunk győződni.
- *Tesztvezérelt feladatmegoldás*: klasszikusan feladatgyűjteménybe való feladattípus. A szöveges megadás mellett az általunk helyesnek vélt tesztadatokat is előkészítjük, miközben a specifikáció üresen marad. Ezt az állapotot mentjük el egy linkben, és adjuk ki feladatként a hallgatóknak, hogy írjanak olyan specifikációt, amely a feladatot megoldva a teszteket kielégíti. Ez a procedúra ráadásul jó példa akár a tesztvezérelt fejlesztésre.
- *Tesztelés*: ritkább feladattípus, mely során adott egy megoldás, és a hallgatónak kell hozzá teszteket írnia. Fejleszti a specifikáció olvasását, ezen belül is az adatmodellezést, annak értelmezését.
- *Adatábrázolás*: a tesztekben megadott adatszerkezethez készítsenek a hallgatók bemenet és kimenet leírást. Előfeltétel, utófeltétel nem kell. Egyelőre elvi lehetőség, mert a szintaktikai ellenőrző elvárja azok helyes megadását, de ezzel egy nagyon fontos készséget lehetne gyakoroltatni.

Specifikációs eszközömnek a főbb módszertani előnyei vannak, illetve az alábbiakat várjuk:

- *Interakció*: Talán a legfontosabb előnye az eszköznek az, hogy a hallgatók interakcióba léphetnek a specifikációval. Nem győzöm hangsúlyozni, hogy mekkora minőségi különbség ez a papíros változathoz képest! Mindkét esetben az írás előtt már kialakul egy elképzelés az alkotóban a modellről, és ezt próbálja formálisan leírni. Papíron nincs visszajelzés, ha valaki bizonytalan, hogy vessző, kettőspont, zárójel kirakása szükséges-e, vagy milyen sorrendben kell megadni az adatokat, akkor nem kap segítséget. A fejében lévő modellre nincs visszajelzése az alkotási folyamatnak. A specifikációs eszközben azonnali visszajelzést kap hibaüzenet, piros aláhúzás formájában, tudja, hogy ott baj van, és ott valamit korrigálni kell. Azonnali visszajelzést kap a gondolati modell, gyorsabb a visszacsatolási, tanulási folyamat. De ennél sokkal fontosabb, hogy mindez megtörténik szemantikai szinten is. Az előzőnél ugyanis csak egy szintaktikai modell kerül megerősítésre, ez utóbbinál viszont a megoldás szemantikai modelljére érkezik azonnali visszajelzés. Ez lehetővé teszi a specifikáció tetszőleges formálását, fejlesztését, refaktorálását.
- *Kreativitás*: az előzőekből következik, hogy az eszköz lehetőséget ad a gondolatok szabad megformálására. Ehhez pedig az adja a biztonságot, hogy az eszköz azonnali visszajelzést ad akkor is, ha valami jól vagy rosszul sikerül.
- *Oktatási folyamat támogatása*: a hivatkozásokon keresztül könnyen előkészíthetőek, kioszthatóak és beadhatók a feladatok. Szabványos és széles körben támogatott webes eszköztárral lehet dolgozni, a linkeket dokumentációba, tanulássegítő rendszerekbe, emailbe, üzenetekbe lehet ágyazni. A helyességet a tesztek zöld-piros állapota azonnal jelzi. Mind az órai, mind az órán kívüli tevékenységek támogatása könnyebbé, az oktatói értékelés gyorsabbá válik.
- *Feladatok eredményesebb megoldása*: hosszú távú célként elvárható, hogy a hallgatók jobban fognak tudni koncentrálni a feladat gondolati modelljére, így gyorsabban haladhatnak. Egy feladatsor létrehozásával pedig akár többet is gyakorolhatnak, ami szintén segítheti az összeredményességüket.

6. Összefoglalás

A feladatmegoldás módszeres lépései közül a specifikáció emelkedik ki a többi közül olyan tekintetben, hogy a legnagyobb nehézséget ez okozza a középiskolából frissen érkező hallgatóknak a felsőoktatás bevezető programozás kurzusain. Ennek oka elsősorban az erős formalizmus és a szokásos algoritmikus gondolkodástól eltérő mód. Mivel ezeket elhagyni nem lehet, így készítettem egy olyan eszközt, amellyel a hallgatók könnyebben gyakorolhatják a specifikációk írását. Ehhez egy mindenki által elérhető és használható webes alkalmazást készítettem, amelyben egy általam definiált, a szakirodalomnak megfelelő teljes, precíz és formális nyelvet használok a specifikációk megadására. A specifikációk helyességét fekete doboz módszerrel ellenőrzöm tesztadatok megadásával. Az eszköz azonnali visszajelzést ad mind a szintaktikai, mind a szemantikai helyességre. Ezen túl tartalmaz egy vizualizációs eszközt, mely az utófeltétel kiértékelését lépésekre bontva mutatja meg, hogy hogyan működik a specifikáció. Az eszköznek már most lehet látni számos módszertani előnyét, de ennek igazolására további mérések elvégzését tervezem. Továbbfejlesztési lehetőségek:

- feladatok szövegének megadási lehetősége,
- egy online feladatgyűjtemény készítése,
- különböző specifikációs nyelvi dialektusok támogatása.

Irodalom

1. Szlávi Péter, Zsakó László: Módszeres programozás: Programozási bevezető. µlógia 18, 8., javított kiadás (2004)
2. Szlávi Péter, Zsakó László: Módszeres programozás: Programozási tételek. µlógia 19, 6., bővített kiadás (2004)
3. Zsakó László, Szlávi Péter, Heizlerné Bakonyi Viktória, Horváth Győző, Menyhárt László, Pap Gáborné, Papp-Varga Zsuzsanna, Gregorics Tibor: *Programozási alapismeretek*. Budapest, Magyarország : ELTE Informatikai Kar (2012)
   https://progalap.elte.hu/downloads/seged/eTananyag/ (utoljára megtekintve: 2023.11.23.)
4. Gregorics Tibor: *Programozás 1. kötet Tervezés*. Budapest, Magyarország : ELTE Eötvös Kiadó (2013)
5. Fóthi Ákos: *Bevezetés a programozásba*. Harmadik javított kiadás. (2012)
   https://bzsr.web.elte.hu/progmod/konyv.pdf (utoljára megtekintve: 2023.11.23.)
6. Bryan Ford: Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. ACM SIGPLAN Notices, Volume 39, Issue 1 (2004)
7. https://progalap.elte.hu/specifikacio/ (utoljára megtekintve: 2023.11.23.)