Webböngészőben futó programozási környezet megvalósíthatósági vizsgálata

Horváth Győző[1], Menyhárt László[1]

[1] {gyozke, menyhart}@inf.elte.hu
ELTE IK

Absztrakt. Cikkünkben egy új, webböngészőben futtatható programozási környezetet mutatunk be, mely segítségével hatékonyabbak lehetnek a programozás oktatásának bizonyos részei. Elsősorban a kódolás, vagyis a szintaxis oktatására fókuszáltunk, ennek során a jelenleg használt konzolos alkalmazásoknál előforduló munkafelületek számát redukáltuk és építettük egybe magával a fordító programmal és a szerkesztő alkalmazással (IDE). Először az oktatási célokat, elvárásainkat és a megszokott környezetet vizsgáljuk, majd az általunk kitalált és létrehozott programozási környezetet írjuk le. A javasolt eszköz jelenleg TypeScript-ben írt forráskódot kezel, támogatja a hallgatókat az adatszerkezetek megértésében és a manuális, illetve nagyszámú automatikus tesztelésben is.

Kulcsszavak: programozási környezet, webböngésző, TypeScript

1. Bevezetés

A programozási környezet jellege erősen befolyásolhatja az oktatás, a tanulás és feladatmegoldás hatékonyságát. A programozással ismerkedők számára számos kiváló környezet érhető el. Bevált gyakorlat, hogy az elemi algoritmikus elemeket először blokk-alapú környezetekben ismerik meg a tanulók, majd fokozatosan térnek át a kód-alapú környezetekre. Jól látszik ez a folyamat az alap- és középfokú oktatásban, ahol az általános iskolás diákok játékos formában blokk-alapú rendszerekben ismerkednek meg a szekvencia, elágazás, ciklus, az események és eseménykezelők, változók fogalmával, majd az általános iskola és középiskola határán térnek át (ha egyáltalán jut erre idő) a kód alapú programozásra. A Scratch környezet jó kiindulási alapot biztosít erre, de számos más rendszer is átvette a blokk alapú programozást (pl. böngészőkben a Blockly). A code.org oldalon például mostanra számos lecke érhető el, különböző korcsoportokba és tudásszintekbe sorolva. Az oldalon bármelyik leckénél megtekinthetjük a blokkoknak megfelelő JavaScript kódot, felhívva a diákok figyelmét arra, hogy a blokk-programozás csak egy egyszerűsítő interfész a valódi kód felé. Ugyanezen az oldalon nemrég óta érhető el egy App Lab: JavaScript eszközök középiskolásoknak nevű lecke, amely már alcímével is mutatja, hogy ebben a leckében már egy konkrét nyelv kódolásával, a JavaScripttel ismerkedhetnek meg. A leckében grafikus felületi elemeket helyeznek el és vezérelnek, amelyeket aztán akár mobiltelefonokon is meg tudnak tekinteni. A felületen egy hatásos animációval mutatják meg, hogy a blokkokból hogyan lesz programkód, illetve ugyanezt visszafele. Ebben a leckében a diák szabad kezet kap, hogy blokkokkal vagy programkóddal dolgozik.

Kód alapú programozást másképp is be lehet vezetni. Ezeknek népszerű formái ugyancsak játékos környezetbe, de azért a nagyobb korosztályt megcélozva csomagolják be ezt a fajta új ismeretet. Ezeknél apróbb játékos feladatok megoldását kell kód írásával elvégezni, a környezetek pedig a legelemibb lépésektől vezetik végig a diákokat a bonyolultabb logikát igénylő feladatokig. Ilyen környezet például a codecombat.org oldal, amely kifejezetten oktatási céllal jött létre, és a kód alapú programozásra épít, miközben a diákok egy szerepjátékot „játszanak”.

Sajnos sokszor a blokk-alapú programozásra úgy tekintenek, mint amit az ifjabb korosztálynak találtak ki, és nem azt a tulajdonságát veszik észre, hogy a kódolás kettős arculatát, a szintaxist és a szemantikát szétválasztja, és kétlépcsőssé teszi: a blokkok használatával és formai egyezésen alapuló összeilleszthetőségével a szintaxis eleve adott, a tanulást az egyes elemek jelentésének és hatásának megértésére lehet koncentrálni. Ha ezek megvannak, akkor lehet áttérni arra, hogy egy-egy ilyen szemantikai blokkot hogyan lehet a fordító- vagy futtatókörnyezet számára érthetővé tenni. Kicsit olyan ez, mintha a blokk-alapú programozással elég lenne csak az algoritmusra fókuszálni a tényleges programkód nélkül.

Az egyetemi oktatásban ritkán jelenik meg a blokk-alapú programozás. Vannak kivételek, így például Berkeley egyetem CSPrinciple kurzusa a Snap! nyelvet használja [1], ami Scratch alapokon saját blokkok definiálását is lehetővé teszi, és számos adatszerkezethez való hozzáférést biztosít.

Mégis az egyetemi kezdő kurzusok legtöbbje azonnal kód alapon oktat programozást. Annak érdekében, hogy ez minél egyszerűbb módon történhessen, minél inkább a lényegre lehessen koncentrálni a lehető legkevesebb „zaj” nélkül, a legegyszerűbb fajta alkalmazásokat, parancssori programokat készíttetnek a hallgatókkal. Ezekben az alkalmazásokban nem kell foglalkozni felületi elemekkel, különböző helyről érkező eseményekkel és azok kezelésével, nem kell bonyolult interfészeken keresztül kommunikációt végezni, csupán a standard inputról beolvasni és a standard outputra kiírni, a kettő között pedig elvégezni az adott feladatot.

A kódolás azonban csak egy része a programozásoktatásnak. Általánosabban feladatmegoldásról kell beszélnünk, amelynek a kódolás csupán csak az egyik lépését képviseli. A feladatmegoldás a feladatleírás értelmezésétől indul, a benne lévő adatok absztrahálásával és leírásával, valamint a köztük lévő kapcsolat leírásával folytatódik (ezt nevezzük specifikációnak), majd egy absztrakt nyelven történő megoldási lépéssorozat megadásából áll (algoritmus), amelyet a kódolás során az adott vagy választott programozási nyelven valósítunk meg. A feladat megoldása még szűkebb körben sem ér véget, hiszen az elkészült program helyességéről teszteléssel kell meggyőződnünk, a feltárt hibákat pedig ki kell javítani. Kisebb programok és feladatok esetén itt véget is érhet a feladatmegoldás.

A programozásoktatásnak van egy másik dimenziója is a feladatmegoldás lépésein túl. Az adott kurzus elvégzésével szeretnénk, ha a hallgatók bizonyos készségekre tennének szert. Egy programozási kurzus kimeneti követelményeinek megfelelően elvárható, hogy a hallgató képes legyen az adatot a feladatleírásból kinyerni és megfelelő módon leírni (adatábrázolás), a feladatot megfelelő és nagy valószínűséggel helyes lépésekre bontani (algoritmikus gondolkodás), kódot nemcsak írni, hanem azt értően olvasni is, az adott feladat helyességéről a be- és kimenő adatok vizsgálatával meggyőződni (tesztelés), a hibaüzeneteket értelmezni és megfelelő eszközökkel a hiba helyét megkeresni és a hibát kijavítani (hibakeresés és -javítás).

Ebben a cikkben azt vizsgáljuk, hogy a jelenlegi parancssori programok írását lehetővé tevő programozási környezetek mennyire képesek a fenti feladatmegoldási folyamatot támogatni és a kimeneti elvárásokban szereplő készségeket fejleszteni. Javaslatot teszünk egy olyan webes programozási környezet kialakítására, mely a fenti célokat megfelelően támogatni igyekszik.

2. Hagyományos programozási környezetek vizsgálata

Mivel eltérő körülményekhez eltérő megoldások szükségesek, így fontos, hogy meghatározzuk azt a környezetet, amelyben vizsgálódásainkat végeztük. A következőkben tett megállapítások és elemzések a felsőoktatás programozásba bevezető kurzusát helyezik fókuszba, amely meghatározza a korosztályt, az elérendő célokat és a szükséges eszközöket.

Az ilyen kurzusokat viszonylag sokféle előismerettel veszik fel a diákok: valaki korábban egyáltalán nem foglalkozott programozással, a többség valamilyen szinten találkozott már valamilyen programozási nyelvvel, és vannak olyanok is, akik már összetettebb feladatokat is oldottak meg programozással. Egy ilyen kurzus tehát egyszerre vezet be mindent az alapoktól, és építi fel az ismereteket egységes formában, hogy a kurzus végére mindenkinek közel egységes rendszerezett ismerete legyen. Ennek megfelelően a feladatok viszonylag kisméretűek, még a kurzus végére sem kerülnek elő olyan feladatok, hogy azokat pl. több állományba kelljen szétszedni.

A rendszerezett ismeretek része a feladatmegoldás módszeres lépésekre bontása is, amelyet fentebb már ismertettünk. Nézzük meg ezeket a lépéseket az eszközszükséglet és a fejlesztői környezet oldaláról. Első lépésként feladat megismerése, elolvasása a cél. Ez történhet szóban elmondva, táblára felírva, vetítőn kivetítve; formája lehet papír alapú vagy digitális; digitális esetében forrása lehet egy általános honlap vagy egy speciális célú feladattár; formátuma valamilyen dokumentumformátum (HTML, PDF, docx, stb). Általában a feladat kiírása másképpen érhető el, mint ahol az implementálás fog történni.

A következő két lépés, a specifikálás és az algoritmizálás a tervezést szolgálja. A tervezés hagyományosan papíron vagy táblán történik. Újabb jelenségként egyre gyakrabban fordul elő azonban, hogy a hallgatók digitális eszközeiken végzik a jegyzetelést, és egyrészt nincs náluk füzet és toll sem, másrészt viszont a fenti két tervezési lépés elvégzése hagyományos szerkesztőkkel (szöveg, kép) nagyságrendileg nagyobb feladat, mint kézzel leírva. Így ez a lépés az órákon a hallgatók részéről egyre többször elmarad. Ezzel a problémával azonban jelenlegi cikkünk nem kíván foglalkozni. A feladatmegoldás ezen részének az az üzenete most számunkra, hogy a tervezési meggondolások egy másik környezetben vagy eszközzel valósulnak meg.

A feladatmegoldás látványos, alkotó része az implementálás, a terv kóddá alakítása. Ez a lépés sokszor valamilyen fejlesztői környezetben valósul meg. Azért választjuk ezeket, mert ezek a kényelmes fejlesztéshez szükséges elemeket egyben tartalmazzák, szemben egy általános célú kódszerkesztésre alkalmas szerkesztővel, ahol az alap munkamenet beállítása időigényes feladat, és sokszor a felület sem támogatja az egyszerű használatot. Nézzük meg ezt kicsit konkrétabban!

A kód szerkesztésére elegendőek lennének a kódoláshoz szükséges paraméterekkel rendelkező egyszerű szerkesztők (pl. Notepad++, Sublime Text, Atom, Visual Studio Code). A fordítás, futtatás történhetne parancssorból, de ennek használata számos egyéb ismeretet is megkíván a programozni vágyótól: parancssor használata, fordítóprogram paraméterezése, stb. Másik lehetőség, hogy egyes szerkesztőkben be lehet állítani, billentyűzetkombinációkhoz lehet rendelni a fordítás parancsát, viszont ennek a beállítása nehézkes (bár egyszeri munka és nem feltétlenül a hallgatónak kell elvégeznie), a futtatáshoz továbbra is parancssorra van szükség (bár sokszor ez be tud épülni a szerkesztőbe), nincs lehetőség felületi ikonhoz rendelni a műveleteket, azaz a felület nem testreszabható. Ha mégis ilyen környezet kerül felhasználásra, akkor annak meg lehet az az előnye a kezdeti nehézségek áthidalása után, hogy a hallgató egyszerűbb környezetben, kevesebb zavaró felületi komponens között tud dolgozni, jobban megértheti a háttérben zajló műveleteket (fordítás, futtatás).

Az integrált fejlesztői környezetek (IDE) az előzőekkel szemben előre fel vannak készítve bizonyos típusú programok szerkesztéséhez, fordításához, futtatásához szükséges eszközökkel (pl. CodeBlocks, Visual Studio, Eclipse). Használatuk kényelmes, az alapbeállítások sokszor elegendőek. Nagy hátrányuk azonban, hogy sokkal összetettebb programok írására alkalmasak, mint amire egy kezdő kurzusnak szüksége van. Egy egyszerűbb feladat megoldásához elegendő egyetlen állomány is, ezek a környezetek azonban projektekben gondolkodnak, amihez több állományt is felvehetünk. A felületük sokszor nagyon bonyolult, hiszen számos funkció elérését

lehetővé teszik a menükben, eszköztárakban, panelekben, beállításokban. Sokkal többre valók, mint amire szükség lenne, és ez felesleges "zajjal" jár.

Ráadásul bármelyik környezetet is használjuk, azt előzetesen telepíteni kell, minden további függőségével együtt. Ráadásul a telepítést a hallgatónak is el kell végeznie saját számítógépén, ha otthon is szeretne dolgozni. Mindig van egy-két olyan eset, amikor ez akadályokba ütközik.

A kódolás után egyik utolsó lépésként következik a tesztelés (amit a hibakeresés és hibajavítás követ), ami két lépésből áll: a szintaktikai és a szemantikai ellenőrzésből. A szintaktikai hibák (ha van, akkor) a fordítás során kiderülnek: a hibalista általában külön panelben jelenik meg, de jobb esetben a környezet jelzi a programkódban is a hiba helyét és jellegét.

A szemantikai tesztelésnek két lényeges része van: a tesztesetek előállítása és a tesztek végrehajtása. A tesztesetek rosszabb esetben csak szóban hangzanak el, jobb esetben felkerülnek a táblára vagy a füzetbe. A tesztelést eleinte kézzel végezzük: a parancssori ablakban kézzel írjuk be a bemeneti adatokat és ott figyeljük a program válaszát. A parancssor általában a fejlesztői környezettől független külön ablakban jelenik meg. Hosszabb input esetén a teszteket fájlba írjuk, és ezeket irányítjuk a futó program standard bemenetére. A tesztek fájlba írását vagy a fejlesztőkörnyezetben, vagy attól függetlenül készíthetjük el. A standard bemenetre irányítás sokszor az integrált fejlesztőkörnyezetekben nem is valósítható meg, így a teszteléshez mindenféleképpen parancssor nyitása szükséges, ami plusz komplexitást ad ehhez a lépéshez.

A hagyományos környezetekben történő fejlesztés gyakran kiegészül online ellenőrző rendszerek (pl. Bíró, Mester [2, 3]) használatával, ahol a kódot objektíven le lehet ellenőrizni. Ezek a rendszerek általában csak a feladatleírást és a kötegelt, automatizált ellenőrzést végzik el, a fejlesztés továbbra is a korábban részletesen tárgyalt környezetben történik.

A fentiek alapján a feladat elvégzéséhez a hallgatónak nyolc különböző helyre kell figyelnie hét különböző felületen:

1. Feladatleírás (valahol külön ablakban: böngésző, PDF olvasó, stb)
2. Tervezés (tábla vagy füzet)
3. Kódolás (fejlesztőkörnyezet)
4. Fordítás (fejlesztőkörnyezet, hibaüzenetek külön panelben)
5. Futtatás (sokszor külön felbukkanó parancsablak)
6. Tesztelési fájlok megadása (külön ablakban)
7. Tesztelés fájlokkal (külön parancsablak)
8. Automatizált tesztelés (külön webes felület, sokféle funkcióval)

3. Javasolt programozási környezet felülete

A fentiekből látható, hogy a feladat elvégzéséhez szükséges információk viszonylag sok helyre vannak szétszórva, ami - amíg a rutin nem alakul ki - feleslegesen von el energiát a hallgató figyelméből, illetve az oktató feladatmegoldásra szánt magyarázati idejéből. Alapozó tárgynál sokkal jobb lenne kevesebb helyre figyelni.

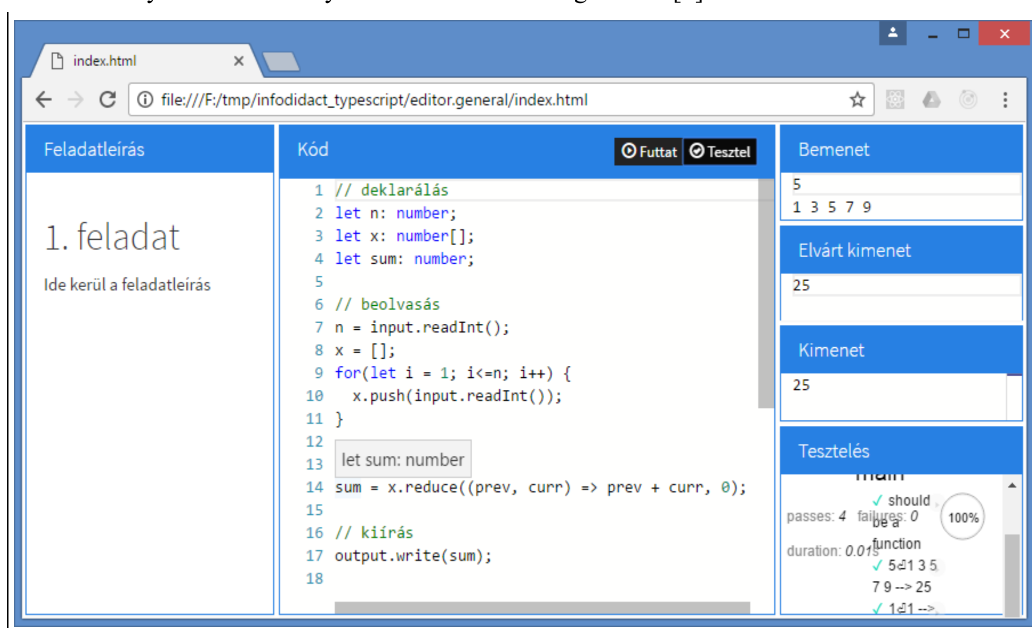
A következőkben egy lehetséges programozói környezet kialakítására teszünk javaslatot, ami a fentebb említett problémák egy részére javasol egy lehetséges megoldást. A környezettel kapcsolatos elvárásaink és céljaink a következők:

- a feladatmegoldás lépéseit mennél inkább támogassa;
- lehetőleg minél több minden egy helyen elérhető legyen;
- a kényelmes kódszerkesztési funkciók továbbra is elérhetőek maradjanak;

- ha lehet, akkor telepítés nélkül használható legyen;
- támogassa az izolált rendszereket is (pl. nincs hálózat).

Az általános célú integrált fejlesztői környezetekkel ellentétben egy olyan programozási környezet létrehozása a célunk, amely a feladatmegoldás lépéseit támogatja, és a figyelmet megpróbálja egy helyen tartani. Ennek megfelelően a környezet felületének kialakításánál ezt kell figyelembe venni.

Mivel jelenleg nem célunk a tervezési fázis során tapasztalt problémákra megoldást találni, így azt a részt továbbra is a programozási környezeten kívüli hatókörbe helyezzük, praktikusan füzetbe vagy táblán elvégezve ezt a lépést. Az oktatás során megvannak ennek az előnyei: a markáns eszköz-elkülönülés (számítógép és tábla/füzet) jelzi, hogy ez jellegében is más folyamat; azt is üzeni, hogy nem rögtön “géphez ülünk”, kódolunk, hanem átgondoljuk a feladatot; végül pedig az írással együtt járó motorikus mozgásoknak is hatékony visszahatása lehet a gondolkodási folyamatokra bizonyos tanulási stílusú hallgatóknál [4].



1. ábra: A javasolt eszköz felülete

A tervezésen kívül viszont a feladatmegoldáshoz tartozó elemeket igyekeztünk egy felületen elhelyezni. A legnagyobb részt maga a kódszerkesztő foglalja el, hiszen az érdemi munka időbeli és markáns része itt valósul meg, ráadásul viszonylag nagy felületet kell biztosítani ahhoz, hogy a kód sorok számát tekintve átlátható legyen. Ha az oktató kódszerkesztést mutat, akkor fontos, hogy a kódszerkesztő rész minél nagyobb méretben és minél nagyobb betűméretben kivetíthető legyen, így a betűméret beállítását és a teljes képernyős szerkesztést is elérhetővé tettük. Fontos, hogy a feladat leírása is ezen a felületen elérhető legyen: a pontos értelmezéshez vissza-visszanyúlhatunk a feladatszöveghez egyrészt, másrészt sokszor példa kódrészletek vagy tesztesetek jelenhetnek meg a feladatleírásban, amit kódolás és tesztelés során felhasználhatunk. A szintaktikai hibák jelenjenek meg a kódszerkesztő ablakban figyelemfelhívó módon, vagy akár hibalistát is adhatunk a kódszerkesztő közelében (tipikusan alatta). A teszteléshez elengedhetetlen a bemeneti adatok megadása, illetve a helyes tesztelés tanításához módszertanilag ildo-

mos az elvárt kimenetet is megfogalmazni egy-egy futtatás előtt. A megírt kódot általában kipróbálni, futtatni szeretnénk, így ez is legyen elérhető a kódszerkesztő közelében. A környezet a megadott bemenettel futtassa le a kódot, és ugyancsak a felület megfelelő helyére írja ki a feladat kimenetét. A környezet jelezze, ha az elvárt és aktuális kimenet nem egyezik meg. A bemenet és az elvárt kimenet megadása a hallgatói teszteléseket támogatja, de közel sem biztos, hogy azok teljes körűek. Ezért a felület biztosítja azt a lehetőséget is, hogy az oktató által megadott tesztek is kötegelten lefussanak a hallgató kódjára, és ennek eredményéről is tájékoztatja a hallgatót. Ennek segítségével a hallgató még egy visszajelzést kaphat, hogy a feladatot jól végezte el, ezzel is segítve az oktató közbeavatkozással járó segítségkérés elkerülését.

4. Az eszközök

4.1. Webböngésző

A programozási környezet megvalósításához szükséges konkrét eszközök választását meghatározza az a platform, amelyen a környezet implementálása megvalósul. Mi ehhez a webböngészőt választottuk több okból. Szinte nincs manapság olyan felhasználóknak kiadott operációs rendszer, amelyen ne lenne alapértelmezetten valamilyen webböngésző feltelepítve. Ezzel biztosíthatjuk részben azt, hogy telepítés nélkül használható legyen a programozási környezetünk. Ez mind az oktatási intézményben, mind otthoni felhasználás során előny lehet. Ráadásul ennek az az előnye is van, hogy platform-, azaz operációsrendszer-független megoldást adunk. A webes technológiákkal elkészített alkalmazások ráadásul felhasználhatóak mind online, mind pedig offline formában (kell felkészítés mellett). Ez utóbbival biztosíthatjuk azt is, hogy izolált környezetben is felhasználható legyen, pl. akkor, ha nincsen hálózati kapcsolat. Ez utóbbi esetben természetesen valamilyen módon gondoskodni kell a tananyag offline formában (pl. tömörített állományként) történő eljuttatásában a célgépekre.

A webes platformra esett választásunk azért is, mert ennek számos előnye lehet a hagyományos környezetekhez képest, ahogy azt korábbi cikkeinkben kifejtettük. Lehetőséget ad, időtől tértől független elérhetőségre, kellően automatizáltság mellett az órán kívüli tanulásra is [5, 6].

A webböngészőben való futtatás és a többi cél elérése érdekében az eszközöket is ezeknek megfelelően kellett megválasztani. A kinézet és elrendezés megvalósítását HTML és CSS segítségével értük el. A rugalmas kinézethez CSS3 flexbox-okat, a tipográfiához Twitter Bootstrap-et használtunk. A felület viselkedését JavaScript nyelv biztosítja.

4.2. Kódszerkesztő

A programozási környezet legfontosabb része a szerkesztő, és ezzel kapcsolatban a kényelmes szerkesztés miatt többek között a következő elvárásokat fogalmazhatjuk meg:

- kódszínezés
- automatikus kódformázás
- szerkesztői funkciók: keresés/csere
- behúzások kezelése
- kódkiegészítés, ajánlások

A webes világban számos kódszerkesztő érhető el (Ace, CodeMirror), amelyek valamilyen módon támogatják ezeket, azonban szerencsére az utóbbi időben jelent meg a Microsoft Monaco nevű szerkesztője, amely ezeket szinte alapbeállításként tartalmazza. További előnye, hogy a nyelvi támogatás is viszonylag könnyen megvalósítható benne (ld. később), és hogy az egyszerűbb kódszerkesztők legmodernebb újításait is tartalmazza: több kurzor, változó hivatkozási

helyeinek keresése, gyorsbillentyűk, gyorsparancs panel, csak hogy a legfontosabbak kiemeljük. A szerkesztő a nyelvi hibákat képes a kódon belül piros aláhúzásként jelezni, a hibaüzenetet pedig megjeleníteni.

4.3. TypeScript

Az offline elvárás erősen behatárolja az elérhető nyelvek körét. Míg azok a rendszerek, amelyeket szerver szolgál ki, képesek a beírt kódot szerveroldalon fordítani és futtatni [2-3, 7-11], addig mi a programozási környezetünkben első körben ezt mind a böngészőn belül szerettük volna megoldani. Ez jelentősen lekorlátozza a használható programozási nyelvek körét. A webböngészők elsődleges programozási nyelve a JavaScript. A JavaScript oktatási felhasználásáról bevezető programozási nyelvként egy korábbi cikkünkben már írtunk [12], és ott azt találtuk, hogy ez a nyelv képes megfelelni az alapvető elvárásoknak, csupán egyetlen hátrányaként említettük, hogy típusok megadása a nyelvben expliciten nem lehetséges. Szerencsére már az akkori cikkünkben is jeleztük, hogy erre megoldást nyújthat a TypeScript nyelv, amely a JavaScript típusos kiegészítője. A TypeScript nyelv fordítója, amely JavaScriptet állít elő, TypeScriptben íródott, így lehetőség nyílik a fordító böngészőbeli használatára. Szerencsére a Monaco szerkesztővel is nagyon jól együtt tud működni, amellyel együtt nagyon jó fejlesztői élmény nyújtható böngészőn belül.

Érdemes néhány szót szólni a TypeScript nyelvről, hiszen korábbi cikkünk [hivatkozás] óta a nyelv további jellemzőkkel fejlődött, amelyek az oktatásban is hasznosak lehetnek. A nyelv szintaktikáját illetően a C alapú nyelvekhez tartozik, így ahol eleve ilyen nyelvet oktatnak, az áttérés nem lehet nehéz. Támogatja az alapvető típusokat: boolean, number, string. Ezek közül a number lehet módszertanilag érdekes, hiszen általában mind a formalizálás során (matematika), mind pedig a kódolás során a legtöbb nyelvben megkülönböztetünk egész és valós típust. Az oktatás során azonban gyakran tapasztalható, hogy a hallgatók eljutnak annak felismeréséig, hogy egy adat számként ábrázolandó, de sokszor csak további figyelmeztetés hatására pontosítják a típust. A bevezető kurzuson előforduló feladatok során pedig előfordul, hogy hibára is vezet az egész és lebegőpontos megkülönböztetés, vegyük például a C++ nyelvet, ahol két egész szám osztásakor az osztás egészrészét kapjuk vissza. Biztos, hogy kezdő kurzuson nyelvi-szintaktikai finomságokra kell koncentrálni a feladat helyes lépésekre osztása helyett?

A TypeScript nyelv egyik hasznos újítása a típusdefiníciók bevezetése. Ezek segítségével az algoritmizálás során (bizonyos algoritmus-leíró eszközök esetén) használt típusdefiníciókhoz közel álló formában tudjuk az összetett (és az elemi) szerkezeteket típusnévvel illetni, és később ezen keresztül hivatkozni rájuk. Nézzünk egy példát, ahol pontok tömbjével szeretnénk dolgozni, és meghatározni az origótól legtávolabb eső pontot:

Specifikáció

```
Be: n∈ℕ, pontok∈Pontⁿ, Pont=X×Y, X,Y=ℝ
Ki: legtávolabbi∈Pont
```

Algoritmus (opcionálisan jelenik meg és az elemi típus elnevezése is opcionális):

```
Típus TPontok = Tömb(1..n: TPont)
TPont = Rekord(x,y: Valós)
TEgész = Egész
Függvény feldolgozás(Konst n: TEgész, Konst pontok: TPontok): TPont
...
Függvény Vége
```

Kód (TypeScript, opcionálisan az elemi típust is elnevezve)

```
type TPontok = TPont[];
type TPont = { x: number, y: number };
type TEgész = number;

let n: TEgész;
let pontok: TPontok;

function feldolgozás(n: TEgész, pontok: TPontok): TPont {
  return //...
}
```

Kód (C++)

```
typedef struct { double a; double b; } TPont;
typedef TPont TPontok[100];
typedef int TEgesz;

TEgesz n;
TPontok pontok;

TPont feldolgozas(const int n, const TPontok pontok) {
    return //...
}
```

Kód (FreePascal)

```
type TPont = record x: real; y: real; end;
type TPontok = array of TPont;
type TEgesz = integer;

var n: TEgesz;
var pontok: TPontok;

function feldolgozas(const n: TEgesz; const pontok: TPontok): TPont;
begin
  feldolgozas := //...
end;
```

Figyeljük meg, hogy mennyire hasonlít a TypeScript a régebben oktatási nyelvként széles körben elterjedt FreePascalhoz. Szemben vele viszont a típusdefiníciók tetszőleges sorrendben megadhatók, kevésbé bőbeszédű a típusmegadás, használható ékezetes karakter. A hasonlóság csak növelhető, ha TypeScriptben is `var`-t használunk `let` helyett.

4.4. Feladatleírás

Annak érdekében, hogy minél egyszerűbben és gyorsabban tudjuk leírni a feladatokat, a feladatleírás formátumának a Markdown nyelvet választottuk. Ezt a nyelvet kifejezetten azzal a céllal hozták létre, hogy általa egyszerűbb legyen egy dokumentum szerkezetének meghatározása, és könnyen lehessen belőle többek között HTML-t generálni.

4.5. Bemenet és kimenet

A JavaScript (és így a TypeScript) nyelv nem rendelkezik beolvasó és kiíró utasításokkal. Ezeket a funkciókat a futtatókörnyezet, jelen esetben a webböngésző biztosítja. Éppen ezért a böngészőbeli TypeScriptben nem tudunk a standard bemenetről olvasni és a standard kimenetre írni. Ezeket valamilyen módon szimulálni kell. A böngésző alapértelmezetten biztosít felugró ablakokat, de ezek használata nehézkes és csúnya. Sokkal egyszerűbb felületi elemeken keresztül

megadni a bemenetet úgy, mintha a hagyományos megoldás során rögtön fájlban határoztuk volna meg a beolvasandó adatokat. A programnak ezt a felületi elemet kell standard inputként tekintenie, és egy másik elemet standard kimenetként. Ezzel kapcsolatban két dolgot kellett meghatározni: a bemenet formátumát és azt a segédletet, amelyen keresztül a bemeneti adatok mint folyam érhetők el.

A bemenet formátumára két koncepciót is kidolgoztunk. Egyrészt támogatjuk azt a formátumot, amely strukturálatlan szövegként határozza meg a bemenetet, és oly elterjedt a különböző oktatói és versenyzői platformokon. A beolvasás megkönnyítése érdekében azonban strukturált formátum is megadható JSON formátumban. Ebben az esetben a beolvasás és kiírás leegyszerűsödik.

Az I/O műveletek elősegítésére formátumtól függő segédosztályokat írtunk, amelyek példányait (`input` és `output`) a kódszerkesztőben natívan elérhetővé tettünk, azaz a szerkesztő ismeri és kódkiegészítéskor felajánlja metódusaival együtt (pl. `input.readInt()`).

Strukturálatlan szöveg és beolvasása

```
5
1 3 5 7 9
```

```
n = input.readInt();
x = [];
for(let i = 1; i<=n; i++) {
  x.push(input.readInt());
}
```

Strukturált bemenet és beolvasása

```
{
  ″x″: [1, 3, 5, 7, 9]
}
```

```
x = input.read('x')
```

4.6. Kód futtatása

A kód futtatását megelőzi a fordítási fázis, amikor TypeScriptből JavaScript lesz. Az így előállt kódot egy megfelelően elzárt környezetben futtatjuk. Szintaktikai hibás kód esetén a kimenetre hibaüzenet érkezik, egyébként pedig ezeket és a szemantikai hibákat a kódszerkesztő piros aláhúzással jelzi.

4.7. Kötegelt tesztelés

Lehetősége van az oktatónak input-output párokat meghatározni, vagy akár teljesen önálló tesztelőt írni, és ezeket kötegelt formátumban a hallgató kódján végrehajtani. A végrehajtást egyelőre a *mocha* nevű tesztfuttató környezet végzi *chai* ellenőrző könyvtár segítségével. A kötegelt tesztelés egy beépülő iframe-ben fut le.

4.8. Nyomkövetés

Böngészőben nyomkövetést a böngésző fejlesztői eszköztárával lehet megtenni. A kódban egy `debugger;` utasítást kell elhelyezni és meg kell nyitni a fejlesztői eszköztárat. A böngészők széleskörű eszközkészletet biztosítanak a megfelelő nyomkövetéshez: lépésenkénti végrehajtás, töréspont elhelyezése, változók értékeinek kiírása, tetszőleges kifejezés kiértékelése, stb. Sajnos böngészőtől függhet az elzárt környezetben való nyomkövetés lehetősége.

5. Feladattípusok

Az alábbi feladattípusok támogatottak a jelenlegi rendszerben:

- “parancssori” programok írása: adott bemenetből az adott kimenet előállítása (kétféle formátummal: szabad szöveges és JSON)

- függvény implementálása: a függvény szignatúrája adott, feladat a bemeneti paraméterek alapján a megfelelő kimenet előállítani. Ebben a feladattípusban nem kell a beolvasással és a kiírással vesződni, elég a feldolgozó függvényt megvalósítani. Mind a szabad szöveges, mind a strukturált bemenet és kimenet támogatott.
- Hibajavítás: adott egy rosszul működő kód, feladat ennek kijavítása. Használatával a hallgató a kód olvasásában is gyakorlatot szerez.

6. Értékelés

A javasolt környezet egy helyen biztosítja a kezdő programozásoktatáshoz szükséges környezeti elemeket. Segítségével idő takarítható meg, a figyelem jobban összpontosítható a megoldandó feladatra, és az automatikus teszteknek köszönhetően nagyobb önálló munka valósítható meg.

A hagyományos környezetekkel (pl. CodeBlocks) és az online ellenőrző rendszerekkel (pl. Bíró, Mester) szemben az általunk javasolt környezetben sokkal kevesebb felé kell a hallgatónak figyelnie a feladatmegoldás során:

1. Feladatleírás/Kódolás/Fordítás/Futtatás/Tesztelés/Automatikus tesztelés (javasolt programozási környezetünk)
2. Tervezés (tábla vagy füzet)
3. Tesztelési fájlok megadása (egyelőre még külön alkalmazásban)

Az egységes és leegyszerűsített környezet időt és energiát spórolhat meg mind hallgatói, mind pedig oktatói oldalon. A feladattípusokkal és a bemeneti formátumok megadásával különböző nehézségű feladatok határozhatók meg. Ha kell teljesen elhagyható a beolvasás és kiírás, és csak a feldolgozó függvényre kell koncentrálni. Ha kell, akkor a beolvasás is megkövetelhető, de itt is lehet egyszerűsíteni a strukturált forma használatával, vagy éppen nehezíteni a szabad forma alkalmazásával. A tesztek támogatásával hibajavítási feladatok is kiadhatók, amelyek fejleszthetik a kód készítése mellett meglévő kód olvasását és értelmezését.

Mindezek együttesen járulnak hozzá a bevezetőben kifejtett kimeneti célok elérésére, és sok esetben azok az energiák és idők, amelyek a hallgatói oldalról a figyelem megosztására, a munkafolyamat követésére, oktatói oldalról pedig a magyarázatra, a hibajavításra fordítódnának, a minőségi és mennyiségi feladatmegoldásra fordíthatóak. A környezettel az egyéni és a differenciált munka is jobban támogatható.

A felület számos irányban fejleszthető tovább: szerveroldali fordítás több nyelv támogatásával, a tervezési lépések ellenőrzésével, tesztesetek kezelésével, további feladattípusok támogatásával, hallgatói munkát követő és elemző rendszerrel.

Hivatkozások

1. http://snap.berkeley.edu/ (utoljára megtekintve: 2016.11.10.)
2. http://biro.inf.elte.hu/ (utoljára megtekintve: 2016.11.10.)
3. http://mester.inf.elte.hu/ (utoljára megtekintve: 2016.11.10.)
4. http://www.iprernyo.hu/Tanulasmodszertan.ppt (utoljára megtekintve: 2016.11.10.)
5. Horváth Győző, Menyhárt László, Zsakó László: Egy webes játék készítésének programozás-didaktikai szempontjai. In: Szlávi Péter, Zsakó László (szerk.) INFODIDACT 2015. Budapest: Webdidaktika Alapítvány, 2015. Paper 2. 10 p. (ISBN:978-963-12-3892-1)

6. Horváth Győző, Menyhárt László: Oktatási környezetek vizsgálata a programozás tanításához. In: Szlávi Péter, Zsakó László (szerk.) INFODIDACT 2014: Informatika Szakmódszertani Konferencia. Budapest: Webdidaktika Alapítvány, 2014. Paper 7. (ISBN: 978-963-12-0627-2)
7. ACM verseny értékelője https://uva.onlinejudge.org/ (utoljára megtekintve: 2016.11.10.)
8. http://cpp.sh/ (utoljára megtekintve: 2016.11.10.)
9. https://www.tutorialspoint.com/compile_cpp11_online.php (utoljára megtekintve: 2016.11.10.)
10. https://www.codechef.com/ide (utoljára megtekintve: 2016.11.10.)
11. http://codepad.org/ (utoljára megtekintve: 2016.11.10.)
12. Horváth Győző, Menyhárt László Gábor: Teaching introductory programming with JavaScript in higher education. The 9th International Conference on Applied Informatics. Eger, Hungary, 2014.