Számonkérési formák a Deklaratív programozás c. tárgyban a BME-n

Benkő Tamás* Hanák Dávid* Hanák Péter†
Szeredi Péter*
Budapesti Műszaki és Gazdaságtudományi Egyetem
{benko,dhanak,hanak,szeredi}@inf.bme.hu

2002. szeptember 24.

Kivonat

A cikk a Budapesti Műszaki Egyetem Műszaki Informatika szakos hallgatói számára tartott Deklaratív programozás című tárgyban alkalmazott számonkérési módszereket írja le. A tárgy jelenlegi formájában nyolc évvel ezelőtt indult és ezalatt az idő alatt a tárgyat hallgató diákok száma rendkívül gyorsan nőtt. 1996-os illetve 2002-es példák segítségével bemutatjuk, hogyan fejlődtek ennek hatására a számonkérési módszerek.

1. Bevezetés

A Deklaratív programozás című tárgyban Standard ML és Prolog nyelvet oktatunk másod- illetve harmadéves informatikus hallgatóknak.

A kurzus előtt a diákok tapasztalatot szereznek tradicionális, imperatív programozási nyelvekben, mint a Pascal és a C. Emiatt mi mind a tanítás, mind a számonkérés során a nyelvek deklaratív tulajdonságaira helyezzük a hangsúlyt.

A tárgy két fő problémája a laboratóriumi gyakorlatok hiánya valamint a hallgatók számának gyors növekedése. A gyakorlatokat önállóan elvégzendő feladatok kiadásával pótoljuk, de ezek összeállítása, megszervezése és kiértékelése nagyon sok munkával jár, ha sokakkal végeztetjük el.

A cikk felépítése a következő. A 2. szakasz áttekintést ad a tárgy történetéről és a diákok tudásának mérésére szolgáló módszerekről. A 3. szakaszban példákat mutatunk zárthelyi feladatokra. A 4. szakasz bemutatja a kis házi feladatokat, és röviden ismerteti a beadott megoldások automatikus tesztelését végző rendszerünket. Az 5. szakaszban két példát mutatunk nagy házi feladatokra, míg a 6. szakaszban a félév végi vizsgáztatás menetét ismertetjük. Végül a 7. szakasz összefoglalja az elmondottakat.

*Számítástudományi és Információelméleti Tanszék
†Irányítástechnika és Informatika Tanszék

2. A tárgy története és háttere

A tárgy jelenlegi formájában 1994-ben 120 hallgatóval indult. 2002-ben már 400 hallgató jelentkezett a tárgyra. Mivel ez a tárgy az alapképzés része, előbb–utóbb az informatika szak minden hallgatójának át kell mennie a vizsgán. Bár a hallgatók száma jelentősen növekedett, a tárgy oktatóinak száma lényegében nem változott. Az oktatói csapat jelenleg két előadóból (egyikük SML-t, másikuk Prologot tanít) és két doktoranduszból áll. Bizonyos feladatokban időnkét magasabb évfolyamos diákok is segédkeznek, akik korábban megszerették a tárgyat.

A diákok tudását többféle módon mérjük: zárthelyivel, kis és nagy házi feladatokkal és félév végi vizsgával. A zárthelyit nagyjából a félév közepén íratjuk meg, kis házi feladatokat több alkalommal adunk ki a második–harmadik héttől kezdve. A zárthelyi és a kis házi feladatok célja az, hogy a hallgatókat minél korábban gyakorlásra bírja. A nagy házi feladat általában úgy tűzzük ki, hogy a hallgatóknak 6–8 hete legyen a kidolgozásra. Ez a feladat a legbonyolultabb egész félév során. A félév végi vizsga arra szolgál, hogy felmérjük a diákok deklaratív programozási képességét valamint a két nyelv alapvető elveinek ismeretét.

Mivel mindig azt gondoltuk, hogy a diákok minél korábbi aktivizálása létfontosságú, és mivel nincs gyakorlat, a zárthelyi és a házi feladatok eleinte mind kötelezőek voltak. A hallgatói létszám gyors növekedése miatt azonban a feladatokra adott megoldások „kézi” ellenőrzése lehetetlenné vált. Ez vezetett minket arra, hogy megpróbáljuk mind a megoldások kiértékelését, mind a másolások kiszűrését [1] automatizálni. Mivel a másolásokat nem lehet 100%-os pontossággal felderíteni és mert a hallgatók erősen terheltek más tárgyak miatt, a feladatok beadása jelenleg nem kötelező. Szintén a diákok nagy száma miatt, a vizsga szóbeli. Ez a forma megbízhatóbbnak bizonyult, és kevesebb erőfeszítést is igényel a vizsgáztatóktól.

A zárthelyire és a vizsgára való felkészülést egy

saját fejlesztésű, web-alapú gyakorlórendszer támogatja [2, 3]. Szeretnénk ennek a rendszernek a használatát kötelezővé tenni és feljegyezni a diákok tévedéseit, de ez problematikus, mivel nincs módunk biztonsággal azonosítani a gyakorlatokat megoldó személyt.

3. Zárthelyi

A zárthelyi fő célja, hogy ellenőrizzük, hogy a hallgatók megértették az oktatott nyelvek alapvető fogalmait és mechanizmusait. A 90 perc alatt megoldandó feladatsor tartalmaz „elméleti" kérdéseket – melyek főleg analitikai képességeket igenyelnek – és egyszerű programozási gyakorlatokat. Az alábbiakban mindkét típusra adunk példát.

Egy tipikus SML „elméleti" feladat, amikor a hallgatóknak meg kell határozniuk a típusát olyan kifejezéseknek, mint

```
fun f x (y, z) = x (y, z)
```

vagy meg kell adniuk a típusát és értékét olyan kifejezéseknek, mint

```
map List.drop [([9,8,7,6,5],4), ([3,2],1)]
```

Gyakori feladat Prologban, egy adott Prolog hívás eredményének meghatározása (siker, meghiúsulás vagy hiba), és siker esetén a keletkező változóbehelyettesítések megadása.

```
| ?- 4+2+3 < X+Y.
```

Egy másik feladat, amikor két Prolog kifejezés kanonikus (azaz szintaktikus édesítőszer nélküli) alakját kell meghatározni, és el kell végezni az egyesítésüket megadva a változóbehelyettesítéseket.

```
| ?- f(U*V, [3*2+b|W]) = f(S, [S+T,T]).
```

Hogy csökkentsük a feladatok megoldásához szükséges időt és kikényszerítsük a két nyelv összehasonlítását, a programozási feladatok gyakran megegyeznek a két nyelven. Az alábbiakban egy példát mutatunk egy ilyen programozási feladatra.

„Adott egy pozitív egész és egy számrendszer alapja, határozza meg a számjegyek listáját!" A specifikációt az eljárás fejkommentárjával és néhány példával pontosítjuk.

```
% számjegyek(Szám, Számrendszer, Számjegyek):
% Számjegyek a Szám szám Számrendszer
% számrendszerbeli jegyeinek listája.
% :- pred számjegyek(int::in, int::in,
%                    list(int)::out).

| ?- számjegyek(45, 10, JL).
                JL = [4,5]
| ?- számjegyek(45,  2, JL).
                JL = [1,0,1,1,0,1]
| ?- számjegyek(45, 16, JL).
                JL = [2,13]
```

Néha további követelményeket is támasztunk, például megtiltjuk segédeljárások/segédfüggvények definiálását vagy megköveteljük, hogy a megoldás jobbrekurzív legyen.

A programozási feladatok javítása nagyon időigényes, különösen azért, mert a hallgatók hajlamosak a szükségesnél komplikáltabb megoldásokat adni. Ha valaki meg akarja találni a legkevesebb változtatást, amivel a megoldás kijavítható (ez ugyanis egy viszonylag jó mérőszám), akkor még egy egyszerű feladat javítása is beletelhet akár fél órába is. Mivel annyi időnk nincs, hogy ezt 400 dolgozatra elvégezzük, az évek során mind több és több pusztán analízist kívánó feladatot vezettünk be. 2002-ben csak egy programozási feladat volt a zárthelyin. A kizárólag analízist kívánó feladatok hátulütője, hogy nem túl érdekesek és elég nehéz olyanokat találni, amelyek releváns képességet vagy tudást tesztelnek. (A lexikai tudást nem tartjuk igazán fontosnak.)

4. Kis házi feladatok

Hogy a hallgatók a lehető leghamarabb programozni kezdjenek, minden szemeszterben több kis házi feladatot is kitűzünk. Általában ezek megoldása nem igényel 15–20 sornál hosszabb programot.

Az első ilyen feladat általában könnyebb, mint a későbbiek, mivel a diákok ekkor még nem ismerik a polimorfizmust vagy egyéb fejlettebb fogalmakat. Az ilyen egyszerű példák egyike a holland zászló probléma. Ebben a feladatban adott értékek egy listája, mindegyik „befestve" a holland zászló egy színével. A program dolga, hogy az értékeket sorba rakja a holland zászló elrendezésének (piros, fehér, kék) megfelelően.

A példa SML specifikációja a következő.

```
(* zaszlo ls = ls lista elemeinek listája
  a holland zászló színeinek sorrendjében
   zaszlo : {num : int, col : string} list
         -> {num : int, col : string} list
*)

- zaszlo [{num=1, col="fehér"},
          {num=2, col="piros"},
          {num=3, col="kék"},
          {num=4, col="piros"}];
> val it = [{num = 2, col = "piros"},
            {num = 4, col = "piros"},
            {num = 1, col = "fehér"},
            {num = 3, col = "kék"}]
     : {num : int, col : string} list
```

Később kicsit nehezebb feladatokat tűzünk ki, mint például a racionális kifejezések egyszerűsítése.

„Írjon egy olyan `történték/2` Prolog-eljárást, amely eleget tesz az alábbi specifikációnak:

- Az első (input) argumentum egy aritmetikai kifejezés, a második (output) argumentum ennek racionális számértéke.
- Az eredmény tovább nem egyszerűsíthető.
- A nevező mindig pozitív.
- Nullával való osztás esetén az eljárás meghiúsul."

```
% :- kif      == integer       \/
%               { - kif     } \/
%               { kif + kif } \/
%               { kif - kif } \/
%               { kif * kif } \/
%               { kif / kif }.
% :- tört     == { számláló #
%                  nevező }.
% :- számláló == integer.
% :- nevező   == integer.
%
% történték(+Kif, -Érték): a Kif aritmetikai
% kifejezés legegyszerűbb értéke Érték.
% :- pred történték(kif::in, tört::out).

| ?- történték(12, E).
     E = 12#1 ? ; no
| ?- történték(4/(-6), E).
     E = -2#3 ? ; no
| ?- történték(3/2 - 1/3, E).
     E = 7#6 ? ; no
| ?- történték(- (4/5) + 1/4 * 6 - 3, E).
     E = -23#10 ? ; no
| ?- történték(1 / (3 - 6/2), E).
     no
```

A házi feladatok feldolgozása teljesen automatikus [2, 3]. A hallgatók elektronikus levélben küldik be megoldásaikat, melyek egy várakozási sorba kerülnek (ha az adott hallgatónak már van megoldása a sorban, akkor azt lecseréljük az újra). A rendszer egy válaszlevélben azonnal jelzi, hogy a beküldött program hányadik a sorban. Tapasztalataink szerint enélkül a visszajelzés nélkül a diákok nem elég türelmesek és többször egymás után beküldik ugyanazt a megoldást. A sorban álló programokat egy démon egyesével feldolgozza, és levélben értesíti a hallgatót a megoldásának tesztelési eredményéről.

A következő szakaszban tárgyalt nagy házi feladatok feldolgozását egy hasonló rendszer végzi.

5. Nagy házi feladatok

Ebben a szakaszban két példát mutatunk nagy házi feladatra, az egyiket 1996-ból, a másikat 2002-ből, valamint ismertetjük a beküldött megoldások kiértékelésére használt eszközöket.

A nagy házi feladat kitűzésekor a legfőbb cél, hogy olyan problémát találjunk, amely „intelligens" algoritmusok felhasználását kívánja meg a megoldáskor.

5.1. Aknakereső

1996-ban a nagy házi feladat a népszerű *Aknakereső* játék egy módosított változatának kidolgozása volt. A módosítás egy súgási lehetőség bevezetése volt, mely azért volt szükséges, hogy a véletlen szerepét kiküszöböljük a játékból.

A játékban egy aktív és egy passzív játékos vesz részt, akik interaktívan a következő lépéseket hajtják végre.

- Kezdéskor a passzív játékos létrehozza a pályát (aknamezőt).
- Az első lépésben az aktív játékos súgást kér.
- A passzív játékos vagy visszautasítja a súgást, vagy információt ad egy vagy több mezőről (hogy többesélyes szituációkban megszüntesse a véletlenszerűséget).
- Ha a passzív játékos visszautasította a súgást, akkor az aktív játékos (kezdetben véletlenszerűen) feltár egy mezőt.
- Az összegyűjtött információ alapján az aktív játékos
  - kijelentheti, hogy ismeri a pálya bizonyos mezőit (azaz tudja, hogy hol van akna, illetve hogy hány elaknásított szomszédja van egy mezőnek),
  - feltárhat egy vagy több biztonságosnak ítélt mezőt,
  - kérhet súgást,
  - kijelentheti, hogy ismeri az egész pályát.
- A passzív játékos információt ad minden mezőről, amit az aktív játékos fel akar tárni, de további mezőkről is adhat felvilágosítást.
- Ha az aktív játékos megpróbál feltárni egy aláaknázott mezőt, akkor a következő lépésben be kell fejeznie a játékot. Amennyiben ezt nem teszi meg, a passzív játékos feladata, hogy véget vessen a játéknak.

- Ha az aktív játékos eltér a protokolltól, akkor passzív játékos hibaüzenetet küld, amire válaszul az aktív játékos kiírja belső állapotát. A következő lépésben a játék véget ér.
- Ha az aktív játékos nem lép aknára, a játék akkor fejeződik be, amikor már ismeri az egész pályát. A passzív játékos ilyenkor ellenőrzi, hogy a megoldás jó-e.

A teljes feladatot több részre bontottuk, és a hallgatók választhattak, hogy mennyit valósítanak meg az egész játékból. Mindenkinek el kellett készítenie legalább az aktív játékos inicializáló és következtető részét. Ezen kívül az alábbi követelményeket támasztottuk a következtetés teljességével szemben:

- az aktív játékosnak fel kell tárnia minden szomszédját az olyan mezőknek, amelyeknek nincs elaknásított szomszédja, és
- a következtetésnek lokálisan teljesnek kell lennie abban az értelemben, hogy a játékosnak fel kell ismernie az elaknásított és a biztonságos mezőket, amennyiben ez kikövetkeztethető a feltárt szomszédos mezőkből.

A játékosok közötti kommunikációnak mind az absztrakt (Prolog/SML adatstruktúra), mind a konkrét (szöveges) protokollját specifikáltuk. A konkrét protokoll megvalósítása nem volt kötelező, ezeket mi rendelkezésre bocsátottuk. A hallgatók ezáltal játszhattak mind a saját mind a többiek programja ellen.

Azok között a megoldások között, amelyek a feladat összes részét megvalósították egy „versenyt” rendeztünk, melynek során mértük

- a súgáskérések számát,
- a mezőfeltárások számát, és
- a teljes megoldáshoz szükséges időt

különböző állapotokból kiindulva. Egy másik tesztben azt számoltuk, hogy egy program hányszor lépett aknára, ha a passzív fél minden súgást visszautasított.

Ilyen módon a legjobb megoldások azok voltak, amelyek a legtöbb információt tudták kikövetkeztetni. Ezeknek a megoldásoknak a szerzői bónuszpontokban részesültek, amelyeket beszámítottunk a félév végi vizsgába.

Megfigyeléseink szerint az érdekes feladat, a versenyzési lehetőség és az értékes jutalompontok lehetősége mind nagymértékben fokozták a hallgatók érdeklődését.

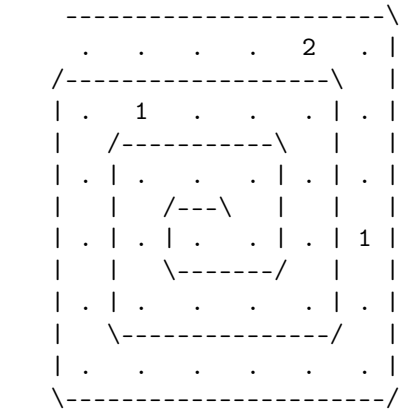
Az ilyen interaktív játékok hátulütője, hogy a protokollok specifikációja hosszú (10–12 oldal), ezáltal a hallgatók könnyen elvesznek olyan részletekben, amiket mi nem ítélünk fontosnak. Emiatt fordultunk inkább az egyjátékos logikai feladványok felé, melyek általában könnyebben specifikálhatók de mégis lehetőséget adnak okos algoritmusok kitalálására.

5.2. Bűvös csiga

2002-ben a *Bűvös csiga* nevű feladványt választottuk nagy házi feladatul. Ebben a feladványban adott egy $n * n$-es tábla, a feladat pedig ennek néhány mezejére lerakni egész számokat az $[1..m]$ intervallumból úgy, hogy a következő tulajdonságok fennálljanak:

1. minden sorban és oszlopban az $[1..m]$-beli egészek pontosan egyszer fordulnak elő, és
2. a bal felső sarokból induló csigavonal mentén az egészek a $[1, 2, \ldots m, 1, 2, \ldots, m, \ldots]$ minta szerint követik egymást.

Néhány egész szám helye már előre le lehet rögzítve. Az 1. ábra egy $n = 6$ és $m = 3$ paraméterű feladványt mutat.

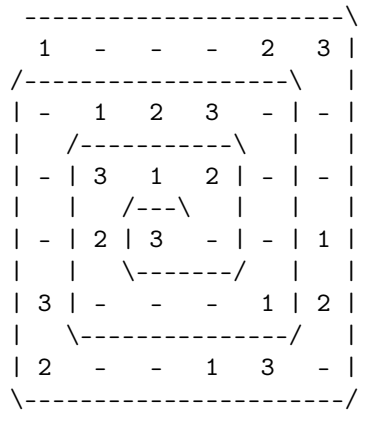


1. ábra. Egy bűvös csiga feladvány

A 2. ábra az 1. ábra feladványának megoldását mutatja.

Az ilyen típusú feladványoknál csak az (általában igen egyszerű) interfészt specifikáljuk. A bűvös csiga esetében egyetlen függvényből/predikátumból áll, mely kap két egészet (n, m) valamint a rögzített elemeket leíró (sor, oszlop, érték) hármasok listáját. A Prolog megoldásnak visszalépések során fel kell sorolnia az összes megoldást, az SML függvénynek vissza kell adnia az összes megoldás listáját. Egy megoldást egyszerűen egészek listájának listájával ábrázolunk, ahol a 0 érték üres mezőt, a többi a tábla megfelelő pozícióján lévő számot jelöli.

A hallgatók munkáját megkönnyítendő, egy keretprogramot biztosítunk, ami képes feladványokat fájlból



2. ábra. A feladvány megoldása

beolvasni, megoldásokat a 2. ábrán látható alakban kiírni, valamint mérni az összes megoldás előállításához szükséges időt.

A megoldások minőségének felméréséhez négy tesztkészletet állítunk elő. Az első készletet a keretprogrammal együtt odaadjuk a hallgatóknak, hogy ők maguk is tudják tesztelni programjaikat. A második készletet a 4. fejezetben említett automatikus tesztelőrendszer használja. A harmadikat a beadási határidő után használjuk, amikor már megvannak a végső megoldások. Azokat a megoldásokat, amelyek minden tesztesetre helyesen és a megadott időkorláton belül futnak le, a negyedik, nagyobb és nehezebb feladványokat tartalmazó készletre is lefuttatjuk. A legjobb programok szerzőit a vizsgán figyelembevett plusszpontokkal jutalmazzuk.

A hallgatók magas létszáma miatt lehetetlen minden megoldást külön megvizsgálni, ezért automatikusan kell feldolgozni a megoldásokat. Így viszont nehéz biztosítani, hogy a diákok egyéni munkát végezzenek. A csalás megelőzésének illetve felderítésének érdekében két dolgot tettünk. Először is nem tettük kötelezővé a házi feladatok megoldását, másodszor pedig munkába állítottunk egy rendszert, amely képes forráskódú programok hasonlóságának mérésére azok hívási gráfja alapján [1].

6. Vizsgáztatás

A számonkérések utolsó fázisa a vizsga. Eredetileg a vizsga teljes egészében írásbeli volt. A megoldandó feladatok között szerepelt néhány egyszerű gyakorlat és egy nemtriviális programozási feladat. Az utóbbi években a vizsga során a magas létszám ugyanazt a nehézséget okozta, mint a zárthelyiknél. Azt is tapasztaltuk, hogy nem volt lehetséges mindent számonkérni, ami az előadásokon elhangzott. Ezek miatt az okok miatt áttértünk egy kevert szóbeli és írásbeli vizsgáztatásra.

A vizsga három részből áll mindkét nyelv esetében. Az első rész egyszerű, a zárthelyin használtakhoz hasonló feladatokból áll. Ezeket nagyjából 15 perc alatt kell megoldani. A következő rész a tárgy egy bizonyos részéhez kapcsolódó elméleti kérdés. Ez is kb. 15 perc. Az utolsó rész egy programozási feladat, melynek megoldására legalább 30 perc áll rendelkezésre.

Néhány hallgató számára a programozási feladat túl összetett, ezért specifikálunk egy részfeladatot, melyet fel lehet (és kell) használni a teljes feladat megoldásához. A feladat ilyen szétbontására példa a következő.

„Adott egy fa, amely az alábbi konstruktorokkal épül fel:

```
datatype fa = L | N of int * fa * fa
```

A fa minden szintjére határozza meg az értékek összegét! Az eredmény egy lista, melynek n-ik eleme a fa n-ik szintjén lévő értékek összege. A fa gyökere a 0-ik szinten van.

```
szintOssz : fa -> int list

szintOssz L                        = [0]
szintOssz (N(3,L,L))               = [3,0]
szintOssz (N(5, N(2,L,N(3,L,L)),
                N(9,N(6,L,L),L))) = [5,11,9,0]
```

A fenti függvény megvalósításához írjon egy segédfüggvényt, amely két fa összegét számítja ki! Két fa összege az a fa, amelyet úgy kapunk, hogy a bemeneti fákat egymásra fektetjük és összegezzük az átfedő csúcsokban levő értékeket.

```
faOssz : fa * fa -> fa

faOssz (L, L)             = L
faOssz (L,
        N(4,N(2,L,L),L)) = N(4,N(2,L,L),L)
faOssz (N(2,N(3,L,L),L),
        N(4,N(2,L,L),L)) = N(6,N(5,L,L),L)
```

A megoldáshoz ne definiáljon semmilyen más segédfüggvényt!”

A fenti specifikáció nem véletlenül ilyen elnagyolt. Ha pontos és elegáns specifikációt akarnánk adni, éppen a megoldást kellene leírnunk!

Egy hallgató félév végi jegyét a szorgalmi időszakban és a vizsgán szerzett pontjai alapján határozzuk meg. Hogy a nagymennyiségű adattal megbirkózzunk, a vizsga előtt vizsgalapokat nyomtatunk minden vizsgázó számára, ami tartalmazza az összes a félév során összegyűjtött pontját. A lapon egyéb mezők is vannak, amiket a vizsgáztatók töltenek ki a vizsga során. A legfontosabb adat a vizsga egyes részeiben szerzett pontszám és az időbélyegek. Valahányszor a hallgató beszél egy vizsgáztatóval, a vizsgáztató felírja a vizsgalapjára

az időt. Ilymódon képesek vagyunk egyidejűleg 30–40 vizsgázó előrehaladását számontartani.

A diákoknak le kell írniuk a válaszokat és megoldásokat, de lehetőségük van szóban korrigálni/pontosítani, mikor a vizsgáztatóval beszélnek.

7. Összefoglalás

Bemutattuk a különböző módszereket, amiket a hallgatók tudásának felmérésére használunk a Deklaratív programozás tárgyban. Leírtuk hogyan fejlődtek ezek a módszerek az évek során, hogy megbirkózzunk a laboratóriumi gyakorlatok hiányával és a hallgatók számának rohamos növekedésével.

Példákkal illusztrálva leírtuk milyen különböző bonyolultságú programozási feladatokat használunk, hogy rávegyük a hallgatókat, hogy gyakorolják a deklaratív programozást. Részletesen tárgyaltunk két viszonylag összetett példát, amik lehetővé teszik, hogy a diákok intelligens algoritmusokkal kísérletezzenek.

Köszönetnyilvánítás. Köszönet illeti az összes diákot, aki segítettek nekünk a tárgy oktatásában. Külön köszönjük Lukácsy Gergelynek, hogy elkészítette és a rendelkezésünkre bocsátotta programmásolás felderítő programját, valamint Berki Lukács Tamásnak és Békés András Györgynek a gyakorlórendszer első változatának elkészítését.

Hivatkozások

[1] Lukácsy Gergely: *Forráskódú programok hasonlóságvizsgálata*
2000, Országos Tudományos Diákköri Konferencia, Eger

[2] Hanák Dávid: *Deklaratív nyelvek oktatásának támogatása számítógéppel*
2001, Budapesti Műszaki Egyetem, diplomamunka

[3] Hanák Péter, Szeredi Péter, Benkő Tamás, Hanák Dávid: *„Magad, uram, ha szolgád nincsen" – Egy Web-alapú intelligens tanító-rendszer*
2001, NETWORKSHOP01, Sopron