Deklaratív nyelvek oktatásának támogatása számítógéppel

Hanák Dávid

2001. június 6.

Feladatkiírás

Ismerkedjen meg az ITS (Intelligent Tutoring System) és más, oktatást támogató rendszerek definícióival és tervezési módszereivel, különös tekintettel az Internet- és Web-alapú rendszerekre. Vizsgáljon meg létező Web-alapú rendszereket.

Mérje fel, hogy a deklaratív nyelvek oktatása során milyen számítógépes támogatásra lehet szükség – vizsgálódásainak alapjául a BME VIK Deklaratív programozás című tárgya szolgáljon. Az igények ismeretében tervezzen oktatást támogató rendszert és készítsen részletes specifikációt. Demonstrációképpen valósítsa meg a rendszer bizonyos elemeit a gyakorlatban.

Kivonat

A felsőoktatás szerepe az utóbbi években megváltozóban van: elitképzés helyett egyre inkább tömegek oktatását szolgálja. A hallgatói létszámok meredeken emelkednek a népszerűbb karokon, szakokon, miközben az előadók létszáma inkább csökkenő tendenciát mutat. Ezért fontosabb ma, mint eddig bármikor, hogy az oktatók munkáját olcsó, de hatékony számítógépes megoldásokkal támogassuk. A feladat egy olyan átfogó rendszer tervezése, amely a legegyszerűbb adminisztratív feladatoktól kezdve a házi feladatok önálló értékeléséig bezárólag minden automatizálható feladatot megold, miközben ellátja (de nem elárasztja) a hallgatókat és az oktatókat az összes szükséges információval.

A terv elkészítése előtt alaposan áttanulmányoztam az oktatást támogató rendszerek terjedelmes irodalmának néhány dokumentumát, hogy megismerkedjem a szokásos fogalmakkal, megközelítésekkel, kiforrott módszerekkel. Ennek során rá kellett jönnöm, hogy olyan rendszer, amely a kívánalmaknak megfelelő feladatokat látna el, vagy nem létezik, vagy lehet róla tudni. Ebből adódóan csak kisebb ötleteket vehettem át, módszereket, megoldásokat nem.

A tervezést kimerítő esettanulmány is megelőzte, amelynek tárgya a BME egy deklaratív programozási nyelveket oktató kurzusa volt – ennek háttérmunkáiban több év óta segédkezem. A vizsgálat során felmértem a tárggyal kapcsolatos tennivalókat, feladatokat, az oktatók és a hallgatók igényeit. A tárgy oktatói már évek óta alkalmaznak különálló programokat bizonyos jól definiált feladatokra – ezek analízisére is sort kerítettem.

Az előkészületeket követően lépésről lépésre, az apróbb részletekre is figyelmet fordítva megalkottam az egységes rendszer tervét. Négy fő komponenst sikerült azonosítanom, amelyek egy-egy feladatkört látnak el. Az egyik a házi feladatként beadott hallgatói programok értékelését, egy másik egy interaktív gyakoroltató rendszer üzemeltetését végzi. Ezek a komponensek lazán kapcsolódnak egymáshoz a rendszer magját alkotó központi adatbázison keresztül, amelyben minden fontosabb adat tárolódik. A laza kapcsolat biztosítja, hogy a komponensek egységes rendszerré álljanak össze, ugyanakkor lehetővé teszi a különálló fejlesztésüket és cseréjüket.

A házi feladatokat értékelő komponens számos részletét meg is valósítottam. Az elkészült programok próbái sikeresnek, értékelésük kielégítőnek mondható. Az eredmények alapján valószínűsíthető, hogy az elkészült tervek szerint kivitelezendő rendszer alkalmas lesz feladatának ellátására.

Abstract

The role of tertiary education has been gradually changing in the recent years: instead of building an èlite it serves more and more the teaching of the masses. The number of students is radically increasing at the more popular faculties while the number of lecturers is rather declining. Therefore it is more important now than ever to support the work of the lecturers with cheap though still efficient computerized solutions. The task to accomplish is to design a comprehensive framework which handles every job that can be automated, from the simplest administrative tasks to the unattended evaluation of homework, and at the same time supplies (but not overwhelms) the students and the lecturers with all the necessary information.

Before starting to draw up the plans of this framework I've thoroughly studied several documents about teaching support systems to get to know the usual concepts, approaches and well established methods. After having read many pages I had to realize that there is no system that would properly handle the required tasks, or if there is one it is not known. Therefore I could adopt no method or solution, except minor ideas.

An exhaustive study of a course on declarative programming at the BUTE —I've been participating in the background works of this course for some years— also took place. In this study I examined the tasks and duties about the course and also the demands of the students and the lecturers. Some sole programs have also been utilized to do several well defined jobs recently – I've analyzed these as well.

Following the preparations, I designed the whole system step by step, minding even the smaller details. I identified four major components, each of which is responsible for a specific scope of duties. One of these evaluates students' programs handed in as homework, another one manages an interactive exercising module. The components are loosely bound via a central database which stores all important data. The loose link ensures that the components constitute a uniform system while it is still possible to develop or replace them separately.

I also implemented most parts of the homework evaluating component. The test runs of these programs have been successful, the evaluation can be said to be satisfactory. The results suggest that a framework to be developed in accordance with these plans will meet the expectations.

Nyilatkozat

Alulírott, Hanák Dávid, a Budapesti Műszaki Egyetem hallgatója kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, és a diplomatervben csak a megadott forrásokat használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

. . . .
Hanák Dávid

Tartalomjegyzék

1. Bevezetés 1
1.1. A tervezés okai és célja 1
1.2. A kiírás pontosítása 1
1.3. A szakdolgozat felépítése 2

2. Oktató és oktatást támogató rendszerek 3
2.1. ITS rendszerek 3
2.1.1. Két ITS architektúra 4
A Byzantium-modell 4
A MANIC-architektúra 4
2.2. CAT rendszerek 5
2.2.1. Web-alapú tesztsorok 5
2.3. „Könnyed bevezetés az ML-be” 5

3. Deklaratív nyelvek és oktatásuk a BME-n 7
3.1. Deklaratív programozás 7
3.2. Deklaratív nyelvek oktatásának eszközei a BME-n 8
3.2.1. Zárthelyi dolgozat (ZH) 9
3.2.2. Nagy házi feladat (NHF) 9
3.2.3. Kis házi feladat (KHF) 10
3.2.4. Gyakorló feladatok 10
3.2.5. Vizsga 10
3.2.6. Számítógépes támogatás 11

4. A jelenleg használt programegyüttes 13
4.1. Házi feladatok feldolgozása 13
4.1.1. Nagy házi feladatok 13
4.1.2. Kis házi feladatok 15
4.1.3. Másolatok kiszűrése hasonlóságvizsgálattal 16
4.2. Gyakoroltatás 16
4.3. Egyéb adminisztratív teendők 17
4.4. A programegyüttes értékelése 18

5. Az ETS rendszer terve 19
5.1. Névválasztás 19
5.2. A rendszer elhelyezése 19
5.2.1. Összevetés más, oktatást támogató rendszerekkel 19
Miért nem ITS? 20
Miért nem CILE? 20
Mért nem CAT? 20
Miért nem hallgatói információs rendszer? 20
5.2.2. Az ETS feladatai 20
5.3. A fő komponensek áttekintése 21
Az adatbázis 21

A házi feladatokat értékelő komponens . . . 22
A gyakorló feladatokért felelős komponens . . . 22
A ZH-kkal és a vizsgákkal kapcsolatos feladatokat ellátó komponens . . . 22
A hallgatók és oktatók adatbázis-hozzáférését biztosító komponens . . . 22
A komponensek kapcsolódása . . . 23
5.4. A hallgatói adatbázis . . . 23
5.4.1. A terv kialakítása két lépésben . . . 24
5.4.2. „Személyes” adatok . . . 24
5.4.3. Események . . . 25
Regisztrációk . . . 25
Eredmények . . . 26
Levelek . . . 27
Egyéb események . . . 27
5.4.4. Levelezési lista . . . 27
5.4.5. A megvalósításról . . . 28
5.4.6. Az adatbázis feltöltése . . . 28
5.5. Házi feladatok feldolgozása . . . 28
5.5.1. Fontosabb fogalmak . . . 29
5.5.2. A komponens szolgáltatásai . . . 30
Programok beadása . . . 30
Levelek fogadása . . . 30
Programok tesztelése . . . 31
Levél kicsomagolása . . . 31
Archívum takarítása . . . 32
Értesítések kiküldése . . . 32
Hívási gráfok generálása . . . 32
5.5.3. Könyvtárak, állományok . . . 32
Nyelvi inicializációs állomány . . . 32
Sablonok . . . 33
Spool állományok . . . 33
Zárak . . . 34
5.5.4. A komponens paraméterei és működése . . . 34
Programok beadása . . . 34
Levelek fogadása . . . 34
Webes beadás . . . 35
Programok tesztelése . . . 35
5.6. A gyakoroltató komponens . . . 36
5.6.1. Feladatok . . . 36
A feladatok csoportosítása . . . 36
Témakörök és sémák a *Deklaratív programozás* esetében . . . 36
5.6.2. A felhasználói felület . . . 37
5.6.3. Belső szerkezet . . . 38
Egy témakör adatai . . . 39
Egy séma adatai . . . 39
Példamegoldások . . . 40
5.6.4. A komponens működése . . . 40
5.7. A ZH- és vizsga-támogató komponens . . . 40
5.7.1. Adminisztratív teendők . . . 41
5.7.2. Tárgyfüggő szolgáltatások . . . 41
Sokcsoportos ZH . . . 41
Automatikus pontozás . . . 42
5.8. Adatbázis-hozzáférést biztosító komponens . . . 42
5.8.1. Lekérdezések . . . 42
Hallgatók lekérdezései . . . 43
Oktatók lekérdezései és módosítási lehetőségei . . . 43

Levelezési listával kapcsolatos lekérdezések 44
További lekérdezések 44

6. Az ETS rendszer megvalósított komponensei **45**
6.1. Házi feladatok feldolgozása 45
6.1.1. Könyvtárak, állományok 46
Sablonok 46
Nyelvi könyvtárak 46
A tesztelő paraméterei 48
A levél-fogadó paraméterei és a spool állomány 50
6.1.2. A komponens használata és működése 51
6.1.3. Értékelés 52
6.2. Hívási gráf építése funkcionális nyelvekre 52
6.2.1. A gráfépítés dilemmái 53
6.2.2. A függvényértékek problémaköre 53
Függvények „átnevezése" 53
Anonim függvények 53
Részlegesen alkalmazható függvények 54
Függvényérték mint argumentum 54
Függvényérték mint visszatérési érték 55
6.2.3. Egyéb problémák 55
Lokális kifejezések és deklarációk 55
Kölcsönös rekurzió 55
Modulok használata 56
Infix függvénydefiníciók 56
6.2.4. Gráfépítés futási időben 56
6.2.5. SML hívási gráf építésének megvalósítása 57
Technikai részletek 57
Hiányosságok 58
Kezelt problémák 58
Futási példa 58
6.2.6. Értékelés 59

7. Tapasztalatok **61**
7.1. Az ETS értékelése 61
7.2. Továbbfejlesztési lehetőségek 62

Összefoglalás **63**

A. A webes adatbázis-hozzáférés egyes lapjai **65**

B. ZH eredmények listaállományának elkészítése **67**

C. Egy naplóállomány **69**

D. A funkcionális nyelvek néhány sajátossága **73**
D.1. Értékek és nevek 73
D.2. Függvények „átnevezése" 74
D.3. Anonim függvények 74
D.4. Részlegesen alkalmazható függvények 75
D.5. Függvényérték mint argumentum 75
D.6. Függvényérték mint visszatérési érték 76
D.7. Lokális kifejezés és deklaráció 76
D.8. Infix jelölés 76

1. fejezet

Bevezetés

1.1. A tervezés okai és célja

A felsőoktatás szerepe az utóbbi években fokozatosan megváltozott: elitképzés helyett ma már inkább a tömegek oktatását szolgálja. A hallgatói létszámok meredeken emelkednek a népszerűbb karokon, szakokon, miközben az oktatók létszáma inkább csökkenő tendenciát mutat. Ez nem kedvez a minőségnek, és sajnos az anyagi körülmények sem ideálisak – az intézmények nem engedhetik meg maguknak, hogy egy-egy egyetemi előadónak jónéhány tanársegéd és doktorandusz dolgozzon a keze alá. Az oktatás színvonalának megőrzéséhez egyre égetőbb szükség van az olyan olcsó megoldásokra, amelyek hatékonyan segítik az oktatók munkáját, megszabadítva őket a felesleges terhektől, hogy minél inkább a tananyag minőségének és frissességének megőrzésére, színvonalas előadások tartására tudják fordítani figyelmüket. Napjainkban a számítógépek segítségével olcsón juthatunk olyan adatfeldolgozó kapacitáshoz, amelyről egy évtizede még csak nem is álmodhattunk. Elég természetes ötlet tehát egy olyan számítógépes rendszer kidolgozása, amely pontosan ezt a feladatot látja el. A cél, hogy a lehető legkevesebb beavatkozással üzemelő rendszert kapjunk, amely minden szükséges információval ellátja az oktatót és a hallgatókat egyaránt. A kapcsolattartáshoz a ma már magától értetődőnek tekinthető Internet széles körben elterjedt elektronikus levelezési és WWW szolgáltatásait lehet és érdemes felhasználni.

Egy ilyen rendszer létrehozásának ötlete a Budapesti Műszaki Egyetem Villamosmérnöki és Informatikai Karán a műszaki informatika szakos hallgatóknak tanított *Deklaratív programozás* című tárgy kapcsán merült fel. Mivel a tárgy programozási nyelvek oktatásával foglalkozik, eleve szorosan kapcsolódik a számítógépekhez, bizonyos esetekben – például programok tesztelésekor – szinte nem is lehet kikerülni a számítógépes támogatást. A tárgy oktatói az évek során egyre több célra alkalmaztak számítógépes programokat, melyek kidolgozásához és kivitelezéséhez korábbi évek hallgatóinak, doktoranduszainak segítségét vették igénybe. Ahogy a programok sokasodtak, egyre nehezebbé vált az áttekintésük és a kezelésük, ezért adódott az ötlet, hogy érdemes lenne egy átfogó, egységes rendszert kidolgozni. Ennek a rendszernek a megtervezése és bizonyos szintű kivitelezése a szakdolgozatom témája.

Ugyanebben a témában a tárgy oktatóival közösen beneveztünk a *Networkshop'2001 konferenciára* is (Hanák, Szeredi, Benkő & Hanák 2001). Az előadásban ugyan személyesen nem vettem részt, de az oktatók kikérték véleményemet és felhasználták részeredményeimet a prezentáció összeállításához.

1.2. A kiírás pontosítása

A kiírás alapján az a benyomásom alakult ki, hogy a szakdolgozatban megtervezendő rendszert szervesen be fogom tudni illeszteni a már létező, oktatást támogató rendszerek világába, és az ott kialakított, megbízható módszerek is alkalmazhatóak lesznek a tervezés során. Ezen rendszerek tanulmányozása közben azonban rájöttem, hogy egyik kategória sem felel meg igazán az itt támasztott követelményeknek. A dolgozat egyik fejezetében természetesen foglalkozom ilyen rendszerek bemutatásával és elemzésével, de ennek a résznek a súlya az elmondottaknak megfelelően kisebb, mint ahogy az a kiírás szerint várható lenne.

A tervezés során viszonylag hamar kialakult az az elképzelés, hogy az elkészítendő rendszernek nem csak a kiírásban is említett tárgynál kellene alkalmazhatónak lennie, hanem ennél sokkal szélesebb körben. Az ITS-ekkel való összevetés helyett tehát inkább az kapott hangsúlyt, hogy hogyan építhető fel a számítógépes rendszer terve kellően általánosan ahhoz, hogy tetszőleges tárgy esetében használható legyen. A megvalósított elemek természetesen továbbra is az adott tárgyhoz kapcsolódnak, hogy munkám haszna ne csak elméleti szinten érvényesüljön.

1.3. A szakdolgozat felépítése

A bevezetést követő **második fejezetben** az oktatást támogató rendszerek világát tekintem át. Az olvasó a feladatkiírásban is említett ITS mellett több más, ebbe a témakörbe tartozó architektúrát, és ezek egy-két konkrét megvalósítását is megismerheti.

A **harmadik fejezet** a deklaratív nyelvekről szól. Ebben a fejezetben elhelyezem a deklaratív nyelveket a programozási nyelvek világában, és definiálom a főbb jellemzőiket. Ezt követően röviden bemutatom a *Deklaratív programozás* című tárgy szerkezetét és történetét, elsődlegesen azokra a jellemzőkre koncentrálva, amelyek egy oktatás-támogató rendszer létrehozása szempontjából lényegesek.

A **negyedik fejezet** azokat jelenleg használt programokat mutatja be, amelyek a rendszer létrehozásának ötletét adták.

Az **ötödik fejezet** alkotja a szakdolgozat magját, itt ismertetem az új rendszer tervét. Bevezetésként elhelyezem az oktatást támogató rendszerek világában, ezt követően pedig rátérek a terv ismertetésére, amely során egy átfogó képből kiindulva fokozatosan finomítom a részleteket. Bizonyos pontokon szót ejtek a megvalósítási megfontolásokról is, de a fejezet lényegét tekintve a tervezés szintjén marad.

A **hatodik fejezetben** bemutatom a rendszer azon részeit, amelyeket – a feladatkiírásnak megfelelően – a gyakorlatban is megvalósítottam. Itt a terv bizonyos pontjait konkretizálom, helyenként szűkítve a lehetőségeket.

A **hetedik fejezetben** értékelem a tervet és a megvalósított részeket, ismertetem a tapasztalataimat, és vázolom a továbbfejlesztésre vonatkozó elképzeléseimet is. Ezt követően ismét összefoglalom a szakdolgozat tartalmát, ezúttal – a tartalom ismeretében – az egyes fejezetek lényegi mondanivalójára hivatkozva.

A **függelékekben** a fejezetekhez kapcsolódó, de a szakdolgozat témájának nem szerves részét képező anyagok találhatóak. Akadnak közöttük látványtervek, programkód-részletek, ill. egy rövid áttekintés a funkcionális nyelvek azon sajátosságairól, amelyek lényegesek a hatodik fejezetben foglaltak szempontjából.

2. fejezet

Oktató és oktatást támogató rendszerek

Az oktatás számítógépes támogatása természetesen nem példa nélküli ötlet. Tudok több, ezen a területen működő rendszerről, és a témakörrel foglalkozó szakirodalom is igen terjedelmes. A szakterület alapvető problémája mégis az, hogy minden egyes esetben más a tananyag, mások az oktatás módszerei, a számonkérések fajtái és mennyisége, egyszóval minden eset egyedi. Ebből adódóan nagyon nehéz olyan keretrendszert alkotni, amely könnyen igazítható az előadók igényeihez, ugyanakkor hatékonyan támogatja a munkájukat.

Vannak olyan törekvések, amelyek az oktatás interaktív részét – a tudásanyag átadását és a megszerzett ismeretek ellenőrzését – teljes egészében át kívánják venni az oktatóktól. Az ilyen rendszerek esetében az oktató feladata a tananyag és a gyakorló feladatok összeállítása, valamint a rendszer monitorozása, a hallgatók előrehaladásának figyelése marad. Az ilyen rendszereket a szakirodalom összefoglaló néven többnyire ITS-nek (*Intelligent Tutoring System*-nek) nevezi.

Más rendszerek az elméleti tananyag átadását továbbra is az oktatóra bízzák, és „csupán" a gyakoroltatást, az elmélet gyakorlatba való átültetésének mechanikus munkáját vállalják át. Néhol ez nem jelent többet egy-két tesztsor kitöltetésénél, más esetekben ennél lényegesen komolyabb feladatról van szó. Az ilyen rendszereket CAT-nek (*Computer Aided Training*-nek) szokás nevezni.

CILE (*Computer Integrated Learning Environment*) néven hivatkoznak többnyire azokra a rendszerekre, amelyek megkönnyítik az oktató munkáját az *előadóteremben*, segítik demonstrációk, látványos prezentációk bemutatását, alapvetően az elméleti anyag alátámasztásaként. Az ilyen rendszerek adott esetben arra is kiválóan alkalmasak, hogy a diákok saját maguk is kipróbálják őket, tét nélkül gyakorolhassanak, kísérletezhessenek saját kedvükre. Remek példa erre az, ahogy Vetier András, a BME Sztochasztika Tanszékének oktatója *Valószínűségszámítás* című tárgyának előadásain a Derive matematikai programot használja ilyen célokra.[1]

Egy negyedik csoportba tartoznak azok a rendszerek, amelyek kizárólag adminisztratív feladatokat látnak el, mint például tárgy-felvételek, vizsga-jelentkezések, vizsgajegyek stb. nyilvántartását. Az ilyen rendszerre az egyik legjobb példa a BME-n kifejlesztett *NEPTUN* hallgatói információs rendszer, amelyet kezdetben csak a Villamoskar hallgatói használtak, később bevezették az egész egyetemen, mostanra pedig már az ország számos más oktatási intézménye is átvette.

A fejezet további részében a két elsőként megemlített kategóriát, azaz az ITS-ek és a CAT-k világát vizsgálom meg valamivel részletesebben, példákat is bemutatva. Ezt követően bemutatok egy olyan weblapot, amelyet az SML programozás oktatásának szenteltek.

2.1. ITS rendszerek

Az ITS az angol *Intelligent Tutoring System* rövidítése. Magyarra fordításának lehetőségei kétségesek, talán a legkifejezőbb változat az *intelligens tutoring rendszer*, de ez sem sokkal „magyarabb", mint az eredeti. Ahhoz, hogy az általa jelölt fogalmat definiálni tudjuk, érdemes külön-külön megvizsgálnunk a névben foglalt három szót; de kivételesen tegyük ezt hátulról visszafelé.

[1] E tárgynál egy szemeszteren keresztül demonstrátorként működtem közre.

Rendszer, azaz több, egymással *együttműködő*, ugyanakkor párhuzamosan, egymástól függetlenül is működőképes komponensből álló architektúra.

Tutoring, azaz oktatást és tanulást támogató rendszer. A *tutor* szóra az alábbi fordítást adja az Országh (1992) kéziszótár: *tanulmányvezető/konzultáló tanár (egyes brit egyetemeken); oktató; házitanító.* Ide legjobban a *tanulmányvezető tanár* fordítás illik, mivel ez fejezi ki azt, hogy a diák önállóan, a saját ritmusában haladva sajátítja el a tananyagot, miközben a „tutor" irányítgatja, terelgeti, *vezeti*.

Intelligens, azaz olyan rendszer, amely képes önállóan elemezni és értékelni adatokat, válaszokat, helyzeteket, és reagálni az értékelés alapján. Az önálló döntések meghozatalához természetesen szükséges, hogy a rendszer rendelkezzék bizonyos előzetes információkkal, mint például az elsajátítandó tananyag felépítése, szerkezete, az egyes fejezetek, témák egymástól való függése.

Összességében ITS-en olyan számítógépes (többnyire hálózatba kapcsolt) rendszereket szoktak érteni, amelyek az oktatóra csak a tananyag összeállítását bízzák, de maguk oktatják a diákokat és ellenőrzik ennek sikerességét. Többnyire adaptívan alkalmazkodnak a hallgatók készségéhez, képességeihez és személyre szabott oktatási programot állítanak össze a korábbiakban összegyűjtött „tapasztalatok" alapján.

2.1.1. Két ITS architektúra

A Byzantium-modell

A Byzantium-modell fantázianevű ITS architektúrát az azonos nevű *Byzantium-projekt* – egy hat egyetemet tömörítő együttműködés – keretében tervezték meg. Patel & Kinshuk (1997) dolgozatukban ismertetik a modell főbb céljait és vázlatos felépítését. A szerzők az ITS rendszer elemi építőköveként az ún. *Intelligent Tutoring Tool*-t (ITT-t) jelölik meg, amely képes egy-egy kisebb tudásanyag-részt, témakört önállóan – emberi segítség nélkül – megtanítani. Az ITT-k jól kombinálhatók az oktatás hagyományos, előadótermi formáival is. Egy ITT az alábbi modulokból áll:

1. **A tudásbázis** tartalmazza az elsajátítandó anyagrészt, és információt arról, hogy hogyan tanítható meg.

2. **A hallgató-modell** követi a hallgató előrehaladását és preferenciáit a tanító-környezettel kapcsolatban. A modell segítségével analizálhatóak a hallgató képességei és tudásszintje, és így a tanítási folyamat az egyén igényeihez igazítható.

3. **A szakértő-modell** meghatározza az egyes feladatok megoldásait és a megoldásokhoz vezető utat.

4. **Az oktató modul** a modelleket és a tudásbázist összeköti a hallgatóval egy grafikus felületen keresztül, és javaslatokat tesz a hallgatónak az eddigi teljesítménye alapján. Analizálja a hallgató válaszait, összeveti a helyes válaszokkal és meghatározza az esetleges eltérések okait, ennek alapján útmutatást ad a hallgatónak a helyes válasz megtalálásához.

A MANIC-architektúra

Sok tekintetben hasonló, de Web-alapú rendszert ismertet Stern (1997). Munkájában a *MANIC (Multimedia Asynchronous Networked Individualized Courseware) keretrendszert* mutatja be, amely tanfolyamok Webre vitelét hivatott megoldani. Az ismeretanyag digitalizált video-folyam formájában áll rendelkezésre, amely természeténél fogva lineáris szerkezetű, ugyanakkor tekintélyes mennyiségű adatot jelent. Stern ezért kiemelkedő fontossággal kezel két témakört: az egyik az eredetileg lineáris szerkezetű ismeretanyag „delinearizálásának" lehetőségei, a másik a Web átviteli kapacitásának korlátozottságából eredő problémák csoportja.

2.2. CAT rendszerek

A CAT jelentése *Computer Aided Training*, azaz számítógéppel támogatott gyakorlás vagy edzés. Ennek megfelelően a CAT rendszerek alapvetően a gyakoroltatást hivatottak átvállalni. Természetesen az ebben a témakörben fellelhető példa-rendszerek színvonala erősen változó: a bárki számára ingyen hozzáférhető, egyszerű Internetes weblapoktól kezdve a pilóták kiképzéséhez használt professzionális rendszerig minden megtalálható. Egy dolog közös bennük: a begyakorlandó elméleti anyagot minél teljesebben kívánják lefedni, hogy kellő gyakorlás után a tanuló szinte rutinszerűen tudja kezelni az összes elképzelhető helyzetet.

Számítógép-programozás oktatása esetén kissé esetlenül hathat az iménti megfogalmazás, de voltaképpen ott is arról van szó, hogy a tanulónak ne a helyes szintaktikára és a nyelvi eszközök helyes használatára kelljen koncentrálnia, hanem ez szinte magától jöjjön, és így a teljes figyelme az éppen megvalósítandó algoritmusra irányulhasson.

Az egyszerűbb, Interneten elérhető CAT rendszerek között számos olyan található, amely elektronikus dokumentáció formájában az elméleti anyag elsajátítását is segíti, ez azonban semmiben nem tér el a nyomtatott tankönyvektől, vagy legfeljebb abban, hogy kényelmetlenebb olvasni. Megfigyelésem tárgyát az ilyen esetekben is a tesztsorok, a gyakoroltató részek képezték.

2.2.1. Web-alapú tesztsorok

Keresgetés közben az Interneten több olyan honlapra is bukkantam, amely az angol nyelv oktatásával foglalkozik, pontosabban szólva a tanuláshoz nyújt segítséget online tesztek közzétételével. A legérdekesebb és tartalmilag legszínesebb ilyen lap a http://www.englishlearner.com címen található. Az itt fellelhető tesztek között van hagyományos feleletválasztós, szókincset vizsgáló jelentéspárosító, folyó szöveget megadott vagy kitalálandó szavakkal kiegészítendő, játékos szókitaláló teszt, és sok egyéb fajta. A feladatsorokban az az egyetlen közös vonás, hogy a rendszer mindegyiket „helyben" ellenőrzi, és bizonyos esetekben kérésre segítséget ad. A készítők a válaszok ellenőrzését a HTML lapokba ágyazható JavaScript nyelvű programok segítségével oldották meg.

Hasonló elven működik az a néhány lap, amelyik imperatív programozási nyelvek, elsősorban Pascal és C oktatásával foglalkozik. Tartalmuk jelentős része a HTML nyelvhez illően kereszthivatkozásokkal megtűzdelt, mégis alapvetően lineáris szerkezetű, könyvszerű, szöveges ismertető az adott programozási nyelvről. Az egyes fejezetek végén azonban az adott fejezet anyagának megértését ellenőrző tesztek találhatók. Ezek a tesztek mind feleletválasztósak, némelyik kérdésnél ugyanazon *program* vagy *programrészlet* több megadott változata közül kell kiválasztanunk az egyetlen hibátlant. Az ilyen kérdések nyilvánvalóan nem olyan „erősek", mint a megválaszolandóak, de ezek a lapok nem egy egyetemi tárgy oktatását, hanem az önképzést szolgálják, tehát nem az a feladatuk, hogy értékeljék a válaszadó tudását, hanem csupán az, hogy segítsék az ismeretanyag elsajátítását.

2.3. „Könnyed bevezetés az ML-be"

A cím az angol *A Gentle Introduction to ML* (röviden GIML) nyersfordítása, amely nem más, mint egy SML-programozás oktatásával foglalkozó weblap. Szerkezete alapján akár a CAT rendszerek közé is sorolhatnánk, hiszen voltaképpen nem más, mint egy példákkal alaposan megtűzdelt SML-könyv tesztekkel kiegészítve. Csakhogy a szerző, Andrew Cumming élesen elhatárolódik mindenféle hasonló rendszertől a bevezető egyik szakaszában (Cumming 1999). A következő bekezdésben ezirányú gondolatait kísérlem meg összefoglalni, mert – noha nem mindenben értek velük egyet – nagyon tanulságosak lehetnek.

Véleménye szerint a CAL (*Computer Aided Learning*) módszerek nem jók tanulásra, az egyetlen előnyük, hogy használatuk olcsó. A legjobb – állítja – egy interaktív, multifunkcionális, intelligens, felhasználóbarát *embertől* tanulni. A legfőbb érve a CAL rendszerek ellen az, hogy segítségükkel az oktatás is a futószalag-gyártáshoz lesz hasonlatos. E találmány kizárólagos célja, hogy a munkát hatékonyabbá tegye, és eme, egyébként nemes cél érdekében az egyén által elvégzendő munkát a lehető legjobban leegyszerűsíti, úgy, hogy azt végső soron egy gép is el tudja végezni. A CAL módszerek bevezetésével

az oktatók munkájának értéke jelentősen lecsökken, maguk a tanárok is könnyen cserélhető alkatrészeivé válnak az oktatási gépezetnek. Cumming a CAL rendszerek legnagyobb hibájának az interaktivitás teljes hiányát tudja be, mert noha e szó legtöbbjük nevében benne van, valódi választást nem ajánlanak fel. Hiszen valójában csak előre programozott lehetőségek közül választhatunk, csak olyan utakat járhatunk be és csak olyan kérdéseket tehetünk fel, amelyekre a rendszer készítője is gondolt. Így elvész a valódi interakció varázsa, amely abban rejlik, hogy a hallgató próbára teheti az oktatót ravasz, előre nem látott kérdésekkel, ellenvetésekkel.

Jól látszik tehát, hogy Cumming nincs valami jó véleménnyel a számítógépes oktató rendszerekről. Ennek megfelelően saját szerzeményét sem CAL-nek tekinti, hanem egy leginkább tankönyvre hasonlító, enyhén interaktív dokumentumnak.

A GIML szerkezete a következő: a dokumentum összesen nyolc leckére és egy bevezető részre van osztva. Minden lecke magját az elméleti anyagrészt ismertető leírás adja, ezt követi az ún. *tutorial*, amely nem más, mint egy megoldandó, esetenként részben megoldott feladatokat tartalmazó lap. Innen kereszthivatkozásokon keresztül különböző részletességű, segítségül szolgáló részekhez juthatunk, és a feladatok egy-egy megoldását is megtudhatjuk. Néhány leckéhez kapcsolódik egy-egy ún. *kitérő* is, amely felvet egy nagyobb lélegzetvételű feladatot, és részletes magyarázattal kísérve végigvisz a megoldáshoz vezető úton.

Számunkra a lapok legérdekesebb része az ún. *önteszt*, amelyből három van (az első három leckéhez). Az önteszt egy feleletválasztós teszt, megvalósításának módja azonban eltér a korábban említett JavaScript-es megoldásoktól. A GIML esetében a teljes teszt az összes lehetséges válasz értékelésével együtt *egyetlen* hosszú lapon helyezkedik el, és a kereszthivatkozások a lapon belülre irányulnak. Az egyes kérdéseket, ill. válaszokat üres sorok választják el egymástól úgy, hogy egyszerre csak egy kérdést láthatunk. A megoldás előnye, hogy egyszerűsége miatt bármilyen böngészőn működik. Természetesen figyelembe kell venni, hogy noha a GIML is egyetemi környezetben készült, e lapok segítségével a hallgatók *saját magukat* képezik és vizsgáztatják (ezt fejezi ki az önteszt név is), és így nem áll érdekükben az eredmények meghamisítása. Olyan esetekben, amikor a teszt eredményét más célokra is fel kell használni, ez az út nem járható.

3. fejezet

Deklaratív nyelvek és oktatásuk a BME-n

Mivel a szakdolgozat témája a deklaratív nyelvek oktatásának támogatása, nem árt, ha az olvasó tisztában van azzal, hogy mit nevezünk *deklaratív* programozási nyelvnek, mi különbözteti meg őket a hagyományosabbnak tekinthető *imperatív* nyelvektől, és hogy mely nyelvek tartoznak ebbe a családba. Erről esik szó a fejezet első részében. A folytatásban a BME-n a deklaratív nyelvek oktatása során alkalmazott eszközöket mutatom be, elsősorban a szakdolgozat témájához kapcsolódó kérdésekre koncentrálva.

3.1. Deklaratív programozás

A deklaratív programozást a saját szavaimmal is bemutathatnám, de szerencsére rendelkezésemre áll a fejezet második részében röviden bemutatandó *Deklaratív programozás* című tárgyhoz készült jegyzet, amely kellő részletességgel megteszi ezt helyettem. Éppen ezért ebben az alfejezetben az említett jegyzet legfrissebb kiadásának megfelelő részéből idézek néhány bekezdést (Hanák 2001, pp. 10–12), helyenként mondatokat, sorokat kihagyva.[1]

Az imperatív programozási paradigma

Az imperatív (más néven procedurális) programozási paradigma a legelterjedtebb, a legrégibb; erősen kötődik a Neumann-féle számítógép-architektúrához. Két fő jellemzője a *parancs* és az *állapot*. A program *állapottere* az a sokdimenziós tér, amelyet a program változóinak értelmezési tartománya határoz meg; a program pillanatnyi állapotát változóinak pillanatnyi tartalma írja le. A program állapotát a leggyakoribb paranccsal, az *értékadással* – azaz a változók frissítésével – változtathatjuk meg.

A deklaratív programozási paradigma

Az imperatív stílussal ellentétben a deklaratív stílusban programozónak – elvileg – csak azt kell megmondania, hogy *mit* akarunk, az algoritmust az értelmező- vagy fordítóprogram állítja elő. A deklaratív programozás két válfaját szokás megkülönböztetni: a *logikai* és a *funkcionális* programozást.

Logikai programozás. A programozási paradigmák közül, amint a neve is mutatja, a legerősebben kötődik a matematikai logikához. Jellemzői a *tények*, a *szabályok* és a *következtetőrendszer*. A legelterjedtebb logikai programozási nyelv a Prolog (PROgramming in

[1] A Szeredi-féle jegyzet bevezetője érthető módon többet árul el a logikai programozásról, de nem ejt szót a funkcionális irányzatról (Szeredi & Benkő 2001).

LOGic). Professzionális, gyakorlati feladatok megoldására alkalmas megvalósításai a deklaratív nyelvi elemek mellett imperatív elemeket is tartalmaznak és fejlett programozási eszközöket megvalósító könyvtára van. Természetesen más logikai programozási nyelvek is vannak, pl. az OPS5, a CLP nyelvcsalád, a Mercury. (Az utóbbi a Prologtól átvett logikai programozási elemeket a típusfogalommal és a funkcionális programozást támogató nyelvi elemekkel egészíti ki.)

Funkcionális programozás. A funkcionális programozás két fő jellemzője az *érték* és a *függvényalkalmazás*. (A függvényérték is érték!) A funkcionális programozás nevét a függvények kitüntetett szerepének köszönheti. A tisztán funkcionális programozási nyelvek a matematikában megszokott függvényfogalmat valósítják meg: *a függvény egyértelmű leképzés a függvény argumentuma és eredménye között*, a függvény alkalmazásának nincs semmilyen más hatása. Tisztán funkcionális programozás esetén tehát nincs állapot, nincs (mellék)hatás, nincs értékadás.

A legelső funkcionális nyelv az 1960-as évek elején kidolgozott LISP (LISt Processing) volt. A sokféle változat közül a professzionális célokra alkalmazható Common LISP a legismertebb. A LISP-dialektusok és modernebb utódjuk, a Scheme is típus nélküli nyelvek. Az első *típusos* funkcionális nyelv az ML (Meta Language) egyik korai változata volt a 70-es évek közepén. A HOPE-pal és más funkcionális nyelvekkel szerzett tapasztalatok alapján dolgozták ki a Standard ML (SML) nyelvet a 80-as évek közepétől kezdve. Az SML után kifejlesztett funkcionális nyelv a Miranda, a Haskell és a Clean is.

Az SML – akárcsak a körülményes szintaxisú, típus nélküli Common LISP – gyakorlati programozási feladatok megoldására készült, ezért nemcsak a tisztán funkcionális, hanem az imperatív stílusú programozáshoz szükséges nyelvi elemek is megtalálhatók benne: frissíthető változók, tömbök, mellékhatással járó függvények stb., továbbá a nagybani programozást segítő fejlett modulrendszere van.

3.2. Deklaratív nyelvek oktatásának eszközei a BME-n

A BME másodéves műszaki informatika szakos hallgatói két deklaratív programozási nyelvet: a SICStus Prologot[2] és az MOSML-t[3] ismerhetik meg a *Deklaratív programozás* című tárgy keretében. Az előadótermi oktatást kivetített fóliák és interaktív számítógépes demonstrációk színesítik, a hallgatókat az előadások anyaga alapján összeállított – ám annál valamivel több témát tárgyaló – jegyzet segíti. Az előadásokat a két oktató – Hanák Péter és Szeredi Péter – kisebb blokkokban, felváltva tartja, a rövid bemutatókon az utóbbi időben demonstrátorok működnek közre. Az előadáshoz – sajnos – számítógépes laboratóriumi gyakorlat nem tartozik. Ugyanakkor nyilvánvaló, hogy a megszokottól ennyire eltérő programozási nyelveket és gondolkodást nem lehetséges gyakorlás nélkül elsajátítani, még a vizsgakövetelmény teljesítéséhez elegendő elemző és konstrukciós készség szintjén sem. (A laboratóriumi gyakorlatokkal kapcsolatos személyes tapasztalataim azt mutatják, hogy gyakran még ezek az alkalmak sem bizonyulnak elegendőnek a tananyag elsajátításához.) Ezért az elméleti ismereteket különböző gyakorló feladatok és félévközi zárthelyi dolgozat segítségével erősítik a tárgy oktatói. Az otthoni programozáshoz ingyenes SML- és licenszköteles Prolog-fordítóprogramok – az adott szemeszterre érvényes ingyenes licensszel – állnak a hallgatók rendelkezésére.

Most tekintsük át, hogy a számonkérés milyen formáit alkalmazzák az oktatók. Amint látni lehet majd, egyes számonkérés-fajták a tárgy történetének kezdetétől jelen vannak, mások csak a legutóbbi időben jelentek meg. Az évek során többször változott az is, hogy melyek kötelezőek és melyek nem. A cél mindig az volt, hogy a hallgatók minél nagyobb bizonyossággal sajátítsák el a két nyelvet, miközben leterhelésük nem halad meg egy bizonyos megengedhető szintet.

[2] A SICS, a Svéd Számítógéptudományi Intézet által kifejlesztett Prolog nyelvjárás.
[3] Moscow ML, az SML egy szabadon hozzáférhető megvalósítása.

3.2.1. Zárthelyi dolgozat (ZH)

A félév során a hallgatóknak egy ZH-t kell (ill. lehet, attól függően, hogy kötelező-e vagy sem) írniuk, amely ellenőrzi a tudásukat a félévben addig elhangzott tananyagból. A dolgozat a korábbi években nyolc feladatból állt, négy-négy feladat jutott mind az SML-, mind a Prolog-rész ellenőrzésére, a 2001-es tavaszi félévben ez a szám tízre emelkedett, miközben a feladatok egyszerűbbé, könnyebben javíthatóvá váltak. Ezek között vannak alapvető nyelvi elemeket firtató, egy-egy egyszerű program (eljárás, függvény) *megértését* elváró, és természetesen konstrukciós, azaz program *írását* megkövetelő feladatok.

Mivel a hallgatók létszáma évről évre nő, a konstrukciós feladatokra adott megoldások pedig gyakran erősen túlburjánzóak és így csak fáradságosan ellenőrizhetőek, az elmúlt szemeszterben az oktatókban felmerült az igény egy a megoldásokra mennyiségi korlátot szabó, egyszerűbben és gyorsabban javítható, tesztszerű ZH-sor kidolgozására. A jelenlegi félévben felhasznált ZH-sorok közeledtek ehhez az elképzeléshez, javításuk egyszerűsödött a kötöttebb forma miatt, de tesztszerűnek még nem nevezhetőek. Az a kérdés, hogy érdemes-e ennél is jobban megszabni a válaszok formáját, egyelőre is megválaszolatlan.

3.2.2. Nagy házi feladat (NHF)

1995-ben a tárgy oktatói úgy döntöttek, hogy a gyakorlást szorgalmazandó minden félévben kiadnak egy, a félév során otthon megoldandó nagyfeladatot. Ezt a feladatot a hallgatóknak mindkét nyelven meg kell oldaniuk, és egy közös, a nyelvtől független algoritmust ismertető dokumentációval együtt be kell adniuk. Az azóta gyűjtött tapasztalatok azt mutatják, hogy azoknak a hallgatóknak, akik sikerrel vették ezt az akadályt, nem jelentett komoly problémát a félévvégi vizsga sem, azon egyszerű oknál fogva, hogy alaposabban megismerték a két nyelvet.

Az elektronikus levélben beadott házi feladatokat kezdettől fogva számítógépes program fogadja és ellenőrzi, igaz, ez a program az évek során sokat fejlődött, sőt majdnem teljesen kicserélődött. A jelenleg használt programról egy későbbi fejezetben részletesen is szó esik.

A feladatokról. Az első házi feladatok párbeszédes jellegűek voltak, mint például a *Master Mind* nevű közismert játék. Az ilyen feladatokhoz a hallgatóknak az algoritmuson túl egy pontosan specifikált kommunikációs protokollt is meg kellett valósítaniuk, hogy a programjuk együtt tudjon működni a szerveren található keretprogrammal. Ráadásul az is elvárás volt, hogy a beadott programok mindkét félt (az aktív játékost, azaz a kérdezőt és a passzív játékost, azaz a válaszadót is) meg tudják személyesíteni. Noha ezek a feladatok esetenként nagyon izgalmasak voltak, számos hátrányuk is volt. Lássuk, melyek voltak ezek!

1. A protokoll megvalósítása mechanikus, de sok időt igénylő munka, ráadásul nem is olyan feladat, amely kihasználná a deklaratív nyelvek sajátosságait, azaz – noha általánosságban hasznos – nem kapcsolódik szorosan a tárgy témájához.

2. A hallgatók otthon, programozás közben csak nehézkesen tudták tesztelni a programjukat, mert vagy saját magával „eresztették össze" (miután a protokoll mindkét felét megvalósították), vagy ők maguk „beszélgettek" a programmal, amely viszont lassúsága folytán nem tette lehetővé a tesztelést kellően összetett és – a program számára is – időigényes feladványok esetén.

3. Az ily módon otthon tesztelt programoknál sokszor csak a beadás után derült ki, ha a hallgató – és így a programja – valahol eltévesztette a protokollt. Az ilyen hibák miatt viszont leggyakrabban egyetlen tesztesetre sem adott megfelelő eredményt a program, noha az algoritmus megvalósítása adott esetben hibátlan is lehetett.

Az oktatók néhány ilyen év tapasztalatain okulva később áttértek az olyan feladatok kiadására, amelyek egyszer, a programok futása elején igényelnek bemeneti adatokat, és végeredményként egy másik adatsorral szolgálnak. Ilyen volt például a *Füles* rejtvény-magazinból is ismert téglalap-kirakó játék, amelyben egy téglalapot egy kisebb téglalapokból álló készletből kell fedésmentesen összerakni. A hallgatók ráadásul készen megkapják a program azon részeit, amelyek a beolvassa a bemeneti adatokat és

kiírja az eredményeket, tehát az adatkonverzióval sem kell bajlódniuk, így teljes egészében az algoritmusuk megvalósítására tudnak koncentrálni. A megoldandó feladatok így némiképp egyszerűsödtek, de egyúttal az ellenőrzésük is könnyebb lett.

A beadásról. A programok beadásának körülményei is változtak az évek során. Eleinte a kiírás része volt, hogy a hallgatóknak az elkészített programokat hogyan kell összecsomagolniuk és egy elektronikus levélbe beágyazniuk, hogy azt a fogadóprogram ki tudja bontani. A tapasztalatok azonban azt mutatták, hogy minden évben számos olyan hallgató volt, aki nem olvasta el figyelmesen a tájékoztatót, és valamelyik előírást nem tartotta be, például más nevet adott az állományainak, vagy az uuencode program használata helyett egyszerűen csatolta a programját a levelezőprogramja segítségével. Ebben az időszakban ráadásul a hallgatók még nem kaptak automatikus értesítést arról, hogy a programjuk rendben megérkezett-e, így a figyelmetlenebbek a beküldés után napokig abban a hitben éltek, hogy beadták a programjukat, és sokan csak későn eszméltek rá, hogy ez koránt sincs így. Emiatt jutottak az oktatók arra az elhatározásra, hogy a beadáshoz egy letölthető Unix szkriptet kelljen használni, így garantálható, hogy minden hallgató a specifikációnak megfelelően küldje be a programját. Sajnos még így is van, aki nem ezt a programot használja, de ők egy válaszlevélben értesülnek a helyes használatról.

3.2.3. Kis házi feladat (KHF)

A félév közepén íratott zárthelyi dolgozatok eredményeiből világosan kiderült, hogy a félév végén beadandó nagy házi feladat nem kényszeríti rá a hallgatókat kellő időben a gyakorlásra, mivel a program elkészítését a nagy többség a határidőt megelőző egy-két hétre hagyja. Ugyanakkor a beadási határidőt nem lehetett előbbre hozni, mert a házi feladat megírásához szükséges tárgyi tudás korábban még nem áll teljes egészében rendelkezésre. Természetesen adódott az ötlet, hogy a deklaratív nyelvekre jellemző sajátosságok megértését és begyakoroltatását sokkal kisebb lélegzetvételű, könnyen megoldható, de ezzel együtt gondolkodtató kis házi feladatok kiadásával segítsük elő. Az ilyen feladatok algoritmikailag nem jelentenek kihívást, de nem is ez a lényegük. Rajtuk keresztül a hallgatók megérthetik a nyelvek szintaktikáját és belső logikáját, az interpreterek működését, általában a deklaratív programok írásának mikéntjét. Ebből adódóan ezek a feladatok nem közösek a két nyelvben, hanem célzottan az SML-hez vagy a Prologhoz kapcsolódnak. A beadott házi feladatok – mintegy buzdításként – a vizsgajegybe beszámító ponttal jutalmazzák.

3.2.4. Gyakorló feladatok

Az idei tanévben először – a kis- és nagyfeladatok megoldása mellett – a hallgatóknak megismerkedhetnek néhány, a tananyag egy-egy részéhez kapcsolódó tesztkérdéssel is. A kérdések egy weblapon keresztül érhetők el, és a válaszok is közvetlenül itt adhatók meg, amelyeket a rendszer azonnal kiértékel és az eredményről tájékoztatja a hallgatót. A feladatok többsége tesztszerűen megválaszolandó, azaz több lehetőség közül kell kiválasztani a jó megoldást, mások egy-egy pársoros rutin megírását igénylik.

A tervek szerint a közeljövőben félévente nyelvenként két-három gyakorló feladat csoport megoldása kötelező lesz, ám egyelőre – az éppen fejlesztés alatt álló rendszer bejáratlansága miatt – még opcionális, a diákok önszorgalomból gyakorolhatnak.

3.2.5. Vizsga

A legjelentősebb számonkérés-fajta természetesen a vizsga, amely mindenki számára kötelező. A vizsga alkalmával a hallgatónak bizonyítania kell, hogy mindkét nyelvben járatos az elvárt mértékben, és hogy rendelkezik értelmezési, elemzési és konstrukciós készséggel. A korábbi években a vizsga a ZH-hoz nagyon hasonló, *írásbeli* számonkérés volt, az utóbbi két évben azonban (az oktatók vizsgaidőn kívüli terhelését csökkentendő, valamint a csalások, puskázások megengedhetetlen túlburjánzása miatt) áttértek a *szóbeli* vizsgáztatásra – ennek feladatai sokban hasonlítanak a korábbi írásbeli feladatokhoz.

Az elmúlt néhány szemeszter során többször változott az oktatók véleménye abban a kérdésben, hogy az egyes számonkérések kötelezőek legyenek-e vagy sem. Amikor egyik vagy másik számonkérési forma nem volt kötelező, az ösztönző erő az maradt, hogy eredménye bizonyos százalékban továbbra is beleszámított a vizsgajegybe. Kötelezővé tételük azért ésszerű, mert rákényszeríti a hallgatókat a

félévközi gyakorlásra, ugyanakkor nagyobb munkát jelent az ellenőrizendő vagy kijavítandó feladatok mennyisége. A 2001-es tavaszi félév során csak a ZH volt kötelező: a nagy és kis házi feladatok beadása, valamint a gyakorló rendszer használata opcionális maradt.

A feladatok kötelezővé tétele felvet bizonyos morális problémákat is. A jogtalan másolás, puskázás, a csalások egyéb módozatai persze akkor is felbukkannak, amikor pusztán előnyt jelent egyik-másik feladat megoldása, de ha egyenesen előírás a teljesítésük, óhatatlanul megugrik azon hallgatók száma, akik ilyen eszközökkel élnek. A nagy házi feladat például jellegéből adódóan igényli, hogy a hallgató ne csak a beadási határidőt megelőző utolsó napokban foglalkozzon vele, hanem huzamosabb időn keresztül. Azok a diákok, akik egyéb elfoglaltságaik miatt mégsem így tesznek, a határidő közeledtével gyakran úgy döntenek, hogy nem adnak be megoldást. Azokban az években viszont, amikor kötelező volt, ez a lehetőség nem állt fenn, és így maradt a többiek megoldásának lemásolása.

3.2.6. Számítógépes támogatás

El lehet képzelni, hogy mennyi munkát jelenthetne a két oktatónak, ha mindezt saját maguk akarnák üzemeltetni, fenntartani. Biztosra vehető, hogy nem is lehetne ennyiféle gyakorlási lehetőséget biztosítani megfelelő segéderő igénybe vétele nélkül.[4] Ezt természetesen az oktatók is rögtön átlátták, és a kezdetektől fogva eleve számítógéppel oldották meg, amit csak lehetett. Ahogy a feladatok köre bővült, úgy szaporodtak a különböző programok, amelyek az újabb és újabb feladatokat voltak hivatottak ellátni. Ezeket a programokat a 4. fejezetben ismertetem. Az alábbiakban igyekszem áttekinteni, hogy az egyes számonkérés típusok milyen jól számítógépesíthető feladatokat rejtenek.

ZH, vizsga: Kézenfekvő feladat az eredmények adminisztrálása, a pontok összesítése, a minimál-követelmények teljesítésének ellenőrzése, az osztályzatok megállapítása, egyszóval a pontszámokkal minden kapcsolatos tennivaló. Az összesített pontszámok megjelennek egy ún. kísérőlapon, amely a szóbeli vizsgáztatáskor segíti a vizsgáztató munkáját. Ezek a feladatok a ZH, ill. vizsga fajtájától, szerkezetétől, sőt kellően parametrizálható rendszer esetén még a tárgykövetelményektől is függetlenek.

Az elmúlt évek során az előadók jelentős ZH- és vizsgafeladat-bázist gyűjtöttek össze, ez segítette elő többek között a szóbeli vizsgáztatás bevezetését is (ahol természetesen a vizsgázók más-más feladatot kapnak). A már említett tesztszerű ZH ötletével együtt jött az az ötlet is, hogy a ZH-sorokat is lehetne automatikusan, névre szólóan generálni ebből a feladattárból. A hatása olyasmi lenne, mintha nem egy 'A' és egy 'B' csoport lenne – ahogy sok dolgozat esetében –, hanem akár annyi, ahány hallgató. Az ilyen automatikus generálással kapcsolatban több tennivaló is akad, a ZH-sorok előállításától kezdve a javítókulcs előállításáig vagy akár az automatikus javításig bezárólag.

NHF, KHF: Az ismertetett számonkérési formák közül a legjobban talán a házi feladatok ellenőrzése automatizálható. A hallgatók programjainak természetesen pontos specifikációnak kell megfelelniük, így semmi akadálya nincs annak, hogy egy számítógépes program fogadja, ellenőrizze és értékelje őket. Az eredményekről ugyancsak automatikusan értesítést kaphat mind a hallgató, mind az oktatók, emellett a pontszámok egy könnyen kezelhető adatbázisba is bekerülhetnek. A konkrét feladvány természetesen évről évre változó, így a rendszernek biztosan van olyan része, amely a feladvánnyal együtt változik, de a legtöbb tennivaló független a feladványtól, sőt a nyelvtől is.

Régóta problémát jelent az, hogy egyes hallgatók évfolyamtársaik programjának másolatát adják be, a legtöbb esetben némileg módosítva. Ez természetesen nem megengedett, de a helyesen megoldott házi feladatért járó jutalom nagyon csábító tud lenni. Az ilyen másolások leleplezése nem könnyű feladat, de megfelelő számítógépes eszközzel hatékonyan segíthető.

Gyakorló feladatok: Ez a számonkérés-típus jellegénél fogva eleve automatizált. A webes hozzáférés szerver oldali megvalósítása olyan CGI szkripteket igényel, amelyek nemcsak hogy önállóan értékelik a hallgató válaszait, hanem magukat a feladatokat is automatikusan választják ki egy adott

[4] Személyes tapasztalatból tudom, hogy például a *C++ programozás* tárgy esetében az oktató demonstrátorokra bízza a több száz házi feladat begyűjtését, ráadásul ehhez két héten keresztül rendelkezésre áll a BME Hallgatói Számítógép Központja.

halmazból. Az eredmények – amellett, hogy természetesen azonnal megjelennek a weblapon – itt is adatbázisba kerülhetnek, hogy később az oktatók pontosan megállapíthassák, ki mit teljesített.

Adminisztratív feladatok: Ide tartozik a jegyzetrendelés, valamint a ZH és pótZH jelentkezések nyilvántartása. Ez előbbire azért van szükség, mert a tárgyhoz biztosított jegyzet – egyelőre – nem kapható a Műegyetemi Könyvesboltban, hanem közvetlenül az oktatóktól szerezhető be. Ennek többek között az az oka, hogy a jegyzet az elmúlt évek során folyamatosan bővült, változott, majdnem minden évben újabb kiadást élt meg, és csak közvetlen terjesztéssel volt megoldható, hogy a hallgatók mindig a legfrissebb kiadáshoz jussanak hozzá. Ehhez viszont az oktatóknak minden félévben fel kell mérnie, hogy hány hallgatónak van szüksége jegyzetre, hogy a lehető legalacsonyabb szinten tartsák a feleslegesen sokszorosított példányok számát. A ZH és pótZH jelentkezésre azért van szükség, hogy a minimalizálni lehessen a feleslegesen sokszorosított ZH-sorok számát is.

4. fejezet

A jelenleg használt programegyüttes

Az előző fejezet végén áttekintettem, hogy melyek azok a feladatok, amelyeket automatizálásra érdemesnek tartok. Természetesen ezek között a feladatok között számos olyan van, amelyekhez már most is létezik működő számítógépes program. Ebben a fejezetben ezeket fogom áttekinteni.

A leghosszabb története a nagy házi feladatokat feldolgozó rendszernek van, nem csoda, hogy ez a programcsomag a létrehozása óta számos változáson ment át, a szkriptek mostanra teljesen kicserélődtek a legelső változathoz képest. Ha nem is ilyen mértékben, de természetesen az összes többi program is változott, fejlődött, tökéletesedett. A fejezetben nem kívánok részletesen kitérni a korábbi változatokra, csak a jelenlegi állapotot mutatom be részletesen (a fejezetcím is ezt sugallja). Egy-egy esetben azonban érdemes lehet megemlíteni, hogy egyik vagy másik szolgáltatás azért került bele a programokba, mert egy korábbi változatuk használata közben derült fény a hiányosságra. Ebben az értelemben tehát helyenként a programok fejlődéstörténete is felbukkan.

4.1. Házi feladatok feldolgozása

4.1.1. Nagy házi feladatok

A feladat kiadása. Az oktatók és segítőik minden tanévben elkészítik a kiadandó házi feladat részletes leírását, amely tartalmazza a feladvány ismertetését, a be- és kimeneti adatok formátumát, a megírandó Prolog-eljárás és SML-függvény specifikációját, valamint az adattípus-definíciókat. A kiírás mellett elkészítik az ún. keretprogramot, amely gondoskodik a kiírásban specifikált formátumú bemeneti adatok beolvasásáról és a kimeneti adatok kiírásáról, aktualizálják a beadáshoz használandó szkriptet, és mindezeket elérhetővé teszik a tárgy honlapján.[1]

A megoldás beadása. A házi feladatokat egy erre a célra készített szkript segítségével kell beadni. A BME minden hallgatója hozzáfér a Hallgatói Számítógép Központ Ural2 nevű központi Unix szerveréhez, tehát jogos az a feltételezés, hogy mindenki képes futtatni a szkriptet. (Természetesen nem csak innen lehet beadni a feladatot, bármilyen POSIX szabványú operációs rendszert, például Linuxot futtató, hálózati hozzáféréssel rendelkező gép megteszi.) A szkript ellenőrzi, hogy a szükséges állományok megvannak-e, azonosítja a hallgatót, majd megfelelő módon összecsomagolja a mindkét nyelven elkészített programot és a mellékelt közös dokumentációt, és névvel, hallgatói azonosítóval (NEPTUN kóddal) ellátva elektronikus levélben elküldi a tesztelést végző központi szervernek. A szerver válaszként visszaküldi a teszt naplóállományát a feladó címére (ahonnan a hallgató futtatta a szkriptet), így értesül a hallgató az eredményről.

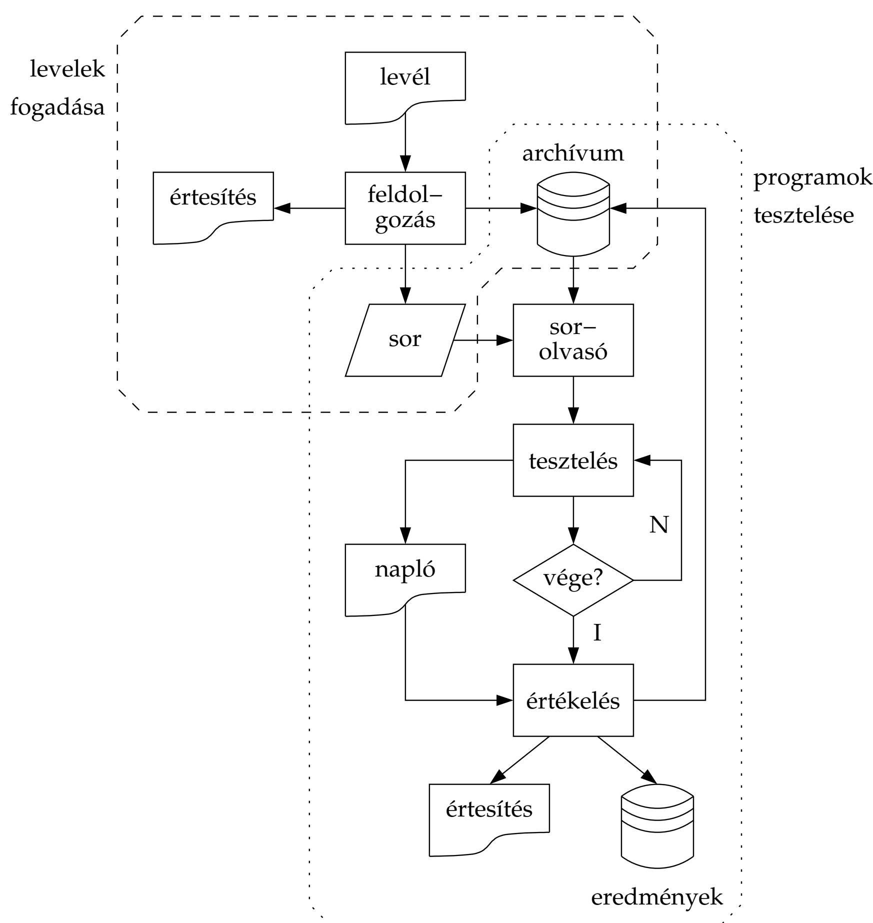
A házi feladatok ellenőrzése. Az itt ismertetett folyamatot a 4.1. adatfolyam-ábra szemlélteti. A szerveren a szkript által generált levelet egy program fogadja, amely kicsomagolja, kiolvassa belőle a beküldő nevét, azonosítóját és elektronikus címét, a teljes levelet eltárolja egy állományban – így az összes

[1] A korábbi évek feladatkiírásai is elérhetők itt, a http://www.inf.bme.hu/~dp címen.

hallgató megoldásának összes változata archiválva van –, és felveszi egy tesztelési sorba. Ezen tennivalók végeztével siker esetén értesíti a hallgatót levele átvételéről, valamint arról, hogy a sorban hány várakozó van előtte, illetve sikertelenség esetén a felmerült hibáról. Ez a levél-fogadó program akkor indul el, amikor egy levél érkezik – így minden levéllel külön processz foglalkozik –, és munkája végeztével kilép.

Ezzel szemben a háttérben *állandóan fut* az ún. teszt-démon, amely rendszeres időközönként ellenőrzi a sort, és ha talál várakozó levelet, akkor a legfrissebb változatot először kicsomagolja egy könyvtárba (minden hallgatónak külön-külön könyvtára van), majd mellémásolva a teszteléshez szükséges keretprogramot és a tesztsort tartalmazó állományokat, minden feladványra teszteli a programokat. Ha egy program egy feladványra hiba nélkül – nem feltétlenül hibátlan megoldást adva – lefut, akkor a megoldás egy kimeneti állományba kerül. Ezt egy másik program egy előre legenerált referencia-megoldással összeveti, és feljegyzi, hogy helyes volt-e vagy sem. A teljes folyamatról, azaz a hallgató programjának lefordításáról, a futás közben kiírt esetleges üzenetekről és az egyes tesztesetek eredményéről naplóállomány készül, amelyet a rendszer archivál, és egyúttal elküld levélben az érintett hallgatónak. Így ha a megoldás nem tökéletes, azaz bizonyos tesztesetekre hibásan működik, a diákoknak még van lehetőségük a beadási határidő előtt megkeresni és kijavítani a hibát.

A hallgatók programjainak a megoldások előállítására korlátozott idő áll rendelkezésére, ez a korlát tesztesetenként más és más. Az idő leteltével a program futása megszakad, eredménye olyan, mintha hibás megoldást állított volna elő. Az idő- és ponteredmények – az oktatók kényelmének kedvéért – a naplóállomány mellett egy erre a célra kijelölt állományban külön megjelennek.



4.1. ábra. Az NHF fogadó program adatfolyam-ábrája

A megoldásokra adott pontszámok, és az, hogy a házi feladat egyáltalán elfogadható-e, természetesen nem a beadási időszak során használt tesztsor alapján dől el, hiszen akkor könnyű lenne becsapni

a rendszert. Az éles teszt eredményéről a hallgató a jelenlegi rendszerben nem levélben értesül, hanem azt a tárgy honlapján nézheti meg, az erre a célra készített HTML űrlapon a NEPTUN kódját megadva.

A feldolgozást végző programok jelenleg a bash Unix shell szkript nyelvén vannak megírva, a levelek és egyéb szöveges állományok feldolgozásához a sed, az awk és a grep nevű segédprogramokat használják. A programegyüttes könyvtárstruktúrájának egy részben kifejtett változatát, amely sokat elárul a lehetőségekről és korlátokról is, a 4.2. ábra mutatja.

```
dp-bhw/                         a programegyüttes gyökérkönyvtára
    spool                       a beérkezett levelek sora
    permanent/                  feladványtól független szkriptek
        mail.script             a leveleket fogadó szkript
        testd.script            a teszt-démon központi szkriptje
    variable/                   feladványtól függő állományok
        00a/                    a 2000-es őszi félév (a = autumn)
        01s/                    a 2001-es tavaszi félév (s = spring)
            pltest/             Prolog-specifikus programok
                SunOS/          tárgykód-változatok Solaris alá
                Linux/          tárgykód-változatok Linux alá
            mltest/             SML-specifikus programok
            tests/              tesztesetek és referencia-megoldások
                1/              1. tesztsor
                2/              2. tesztsor
            limits/             időlimitek
        current → 01s/          „soft link” az aktuális félév könyvtárára
    homeworks/                  beküldött feladatok, archívum
        01s/                    a 2001-es tavaszi félév állományai
            mails/              az archívum
            recent/             a levelek legutolsó verziója kicsomagolva
        current → 01s/
    scores/                     futási eredmények
        01s/                    a 2001-es tavaszi félév eredményei
        current → 01s/
```

4.2. ábra. Az NHF feldolgozó programegyüttes könyvtárszerkezete

4.1.2. Kis házi feladatok

A nagy házi feladatokat feldolgozó keret nem bizonyult minden tekintetben alkalmasnak a kis házi feladatok fogadásához. Ennek okai a következők voltak:

- Az NHF-eket mindkét nyelvből egyszerre kell beadni – a KHF-ek (céljukból adódóan) nyelvspecifikusak. Ebből adódóan a két esetben más a teszteléshez használt programok struktúrája.
- Az NHF-ek teszteléséhez többnyire csak bonyolultan megadható feladványok alkalmazhatóak, és a megoldások ellenőrzése is egy (még a feladvány leírásánál is terjengősebben megadható) referencia-megoldással való összehasonlítást jelent – a KHF-ek „feladványai” azonban egyszerűek, egy-egy sorban leírhatóak. Emiatt a feladványok és a referencia-megoldások megadása más szerkezetet igényel, a jobb átláthatóság kedvéért. Az NHF-ek esetében minden feladvány, valamint a megoldások külön-külön állományokban helyezkednek el, míg a KHF-ek ellenőrzéséhez használt keretrendszerben egyetlen állomány külön soraiban.
- Az NHF-ek ellenőrzése során mérjük a felhasznált időt – erre a KHF-ek esetében nincs szükség, mivel minden feladványra gyakorlatilag azonnal meg kell tudni adni a választ, az időkorlát bőkezűen beállítva sem ér el emberi mértékkel mérve jelentős értéket.

Mivel a KHF-ek kiadásának ötletétől az első beadási határidőig rendelkezésre álló idő igencsak korlátos volt (kb. egy hónap), a meglévő NHF keretrendszer kellő általánosítása nem látszott járható útnak, hiszen csupán a teendők alapos átgondolása hasonló mennyiségű időt vett volna igénybe. Egyszerűbbnek látszott a rendszer lemásolása és apróbb módosítása a megváltozott igényeknek megfelelően, így jelenleg két – nagy mértékben hasonló – rendszer működik egymás mellett. Ennek a megoldásnak a hátrányát szükségtelennek érzem hosszasan ecsetelni, hiszen nyilvánvaló, hogy a megduplázott rendszer karbantartása, hangolása is dupla annyi időt vesz igénybe, mivel minden – általános érvényű – módosítást kétszer kell elvégezni. A két rendszer egyesítése egy általánosabb, összefogó architektúrába már régóta várat magára, és e diplomatervvel remélhetően eljut a tervezés szintjére, amelyet a későbbiekben követhet a megvalósítás is.

Az eltérések az NHF értékelő rendszerhez képest a következők:

- Nincsen külön adatbeolvasó keretprogram, mert a feladványok olyan egyszerűek, hogy kézzel is megadhatóak, az ellenőrzést végző keretprogramba pedig be van építve a beolvasó.
- A beadás az összes KHF esetében *ugyanazzal* a szkripttel történik: a szkript felismeri, hogy éppen melyik feladatot adják be a segítségével, és ezt az információt kódolja a levél törzsében is.
- Ugyanaz a rendszer értékeli az összes kis házi feladatot, az eltérések a variable/ könyvtár megfelelő állományaiban jelentkeznek.
- A teszteseteket egyetlen állományban adjuk meg, melynek minden sorában egy-egy feladvány áll.
- Nem mérjük a futási időt, és a limit is közös az összes feladványhoz. Mivel minden esetben pillanatok alatt le kell futnia a programnak, egy pár másodperces limitnek sokszorosan elégnek kell lennie.

4.1.3. Másolatok kiszűrése hasonlóságvizsgálattal

Mint már említettem, az NHF-ek beadásának kötelezővé tétele ugrásszerűen megnövelte a másolások számát. De ez a jelenség egyébként is megfigyelhető, ugyanakkor küzdeni ellene meglehetősen nehéz. Nyilvánvaló, hogy több száz hallgatói program végigolvasása, alapos tanulmányozása szinte lehetetlen feladat. Ez okból készített a BME egyik informatika szakos hallgatója, Lukácsy Gergely egy olyan programot (Prolog nyelven), amely képes programok hívási gráfjának összehasonlítására és az egyezések detektálására (Lukácsy 2000). A program a gráfokon számos redukciós lépést végez el, hogy azok a strukturális – ám működésbeli eltérést nem okozó – különbségek, melyek például egy eljárás két részre vágásával adódnak, eltűnjenek, ill. lelepleződjenek. A programhoz készített kiegészítő modul alkalmas egy Prolog-program hívási gráfjának feltérképezésére, de a hasonlóságvizsgáló algoritmus megvalósítása nyelvtől független, megfelelő illesztéssel tetszőleges programozási nyelven írt programok összehasonlítására alkalmas.

Az alkalmazott módszer előnye, hogy vele szemben hatástalanok azok a másolást álcázó trükkök, amelyek az eljárások, változók, struktúrák átnevezésén vagy a program primitív átstrukturálásán alapulnak. E program segítségével az elmúlt évben több olyan csalásra is fény derült, amelyek valószínűleg észrevétlenek maradtak volna nélküle. Természetesen csak NHF-ek esetében alkalmazzuk, hiszen a KHF-ek túlságosan egyszerűek ahhoz, hogy ne hasonlítsanak egymásra a megoldások; és egyelőre csak a Prolog-részekre, mivel az SML-hez mindezidáig nem készült hívási gráfot építő modul. A 6.2. fejezetben éppen egy ilyen, általam készített programot mutatok be, amellyel SML-programok összehasonlítása is lehetővé válik.

4.2. Gyakoroltatás

A 2001-es év tavaszi szemeszterével kezdődően az oktatók a nagy és kis házi feladatok kiadása mellett újabb lehetőséget biztosítanak az önálló gyakorlásra. Mivel az ilyen jellegű feladatok megoldása nem igényel többet néhány perc próbálkozásnál, nem sok értelme lenne az offline jellegű ellenőrzésnek. Ehelyett ebben a félévben két korábbi hallgató a tárgyfelelős tanszék által alkalmazott demonstrátorként azt a feladatot kapta, hogy tervezze meg és alakítsa ki a gyakoroltatáshoz szükséges webfelületet és a

hozzá tartozó CGI programokat. A rendszer első, élesben kipróbálható változata el is készült, ez jelenleg a következő egyszerű feladat-típusokat ismeri:

1. Prolog
   - kanonikus alakra hozás (szintaktikus édesítőszerek nélküli felírás);
   - hívás eredményének megállapítása „sikerül/meghiúsul/hiba” jelleggel, siker esetén a változó(k) értéke;
   - hívásra válaszul adott behelyettesítések megadása, a sorrend betartásával;
   - egyszerű eljárás írása, pl. lista hosszának kiszámítására.

2. SML
   - „mi a típusa a kifejezésnek?” jellegű kérdések;
   - kifejezés értékének megállapítása;
   - egyszerű függvény írása.

A hallgatók a NEPTUN kódjukkal azonosítják magukat, és ettől kezdve a program nyilvántartást tud vezetni az egyes diákok haladásáról, ill. a követelmények teljesítéséről. Természetesen a belépések között is emlékszik a legutóbbi állapotra, tehát a hallgató ott folytathatja, ahol legutóbb abbahagyta.

A program igyekszik értelmes hibaüzeneteket adni a hibás válaszokra, ennek megfelelően megkülönbözteti a szintaktikai és a szemantikai hibákat, valamint a formai követelmények be nem tartásából eredő hibákat, és minden esetben többé-kevésbé informatív üzenettel válaszol. Ha egy hallgató az adott feladattal nem boldogul, akkor lehetősége van ugyanabból a típusból másik feladatot kérni, vagy másik típust választani.

4.3. Egyéb adminisztratív teendők

Jegyzetrendelés. A *Deklaratív programozás* tárgyhoz a teljes órai anyagot felölelő és valamivel azon túlmutató, rendszerezett, papírborítású könyv formájában évről évre rendszeresen kiadott jegyzet áll a hallgatók rendelkezésére, ez azonban nem kapható az egyetemi könyvesboltban. Ehelyett a félév első néhány hetében a hallgatóknak nyilatkozniuk kell arról, hogy kérnek-e jegyzetet (a keresztfélévesek, és azok, akik felsőbbévesektől megkapják, nem szoktak rendelni). A rendelés eleinte az előadáson körbeadott ívek aláírásával történt, ám a tárgyat felvett hallgatók számottevő része egyáltalán nem jár előadásra, ilyen módon többen lemaradtak róla. Az utóbbi két évben ennek a problémának a megkerülésére a jegyzetrendelés Weben keresztül zajlik, a NEPTUN kód megadásával. A rendeléssel egyidőben a hallgatóknak meg kell adniuk egy elektronikus levelezési címet is, hogy szükség esetén elérhessük őket.

Jelentkezés ZH-ra. A vizsgákra az egyetemi szabályozásnak megfelelően a NEPTUN hallgatói információs rendszeren keresztül jelentkezni kell, így a vizsgát megelőző napon már pontos létszámadatok állnak az oktatók rendelkezésére. A névsor segítségével elkészíthető a szóbeli vizsga időbeosztása, ill. a korábbi években megállapítható volt, hogy az írásbeli vizsgára a feladatsort hány példányban kell sokszorosítani. A zárthelyi dolgozat esetében azonban nincs ilyen közös jelentkezési rendszer. Emiatt rendszeresen előfordult, hogy az oktatók jelentősen túlbecsülték a ZH-n megjelenő hallgatók létszámát, és a feladatsorból szükségtelenül és aránytalanul sokat sokszorosítottak. A pazarlás minimalizálása érdekében az utóbbi időben a ZH-ra is jelentkezni kell egy weblapon keresztül. A jelentkezés ténye nem ró kötelezettségeket a hallgatóra, a megjelenése továbbra sem kötelező, az így kapott létszámbecslés tehát még mindig meghaladhatja a ténylegeset, a kapott kép mégis pontosabb.

Terembeosztás, ültetési rend. A jelentkezők névsora, a ZH-ra megkapott termek mérete és elrendezése alapján előállítható a hallgatók terembeosztása és az ültetés rendje is. Ez utóbbira azért van szükség, mert ezzel is csökkenthető a csalás esélye.

Kísérőlap készítése. Az oktatók kényelmét szolgálja az a program is, amely a szóbeli vizsgáztatáshoz nyújt segítséget. A vizsgajegybe számos félévközi eredmény beleszámít, ám ezek nyilvántartása odafigyelést igénylő munka. Ezt megkönnyítendő a program használatával *kísérőlap* nyomtatható minden vizsgázó hallgatónak, amely automatikusan tartalmazza a félév közben megszerzett összes pontszámot, továbbá különböző rovatokat a vizsga során megoldott feladatok pontszámainak feltüntetésére. Így a vizsga végén a megszerzett jegy gyorsan megállapítható akkor is, ha a hallgatót többen vizsgáztatták, és ebben esetleg a jegyet az indexbe beíró tanár nem is vett részt.

4.4. A programegyüttes értékelése

A tárgy történetét ismerve érthető, hogy ezen programok fejlesztése mindig ad-hoc jellegű volt, soha nem előzte meg alapos tervezés és előkészítés, az időkorlátok miatt mindig a lehető legegyszerűbb és leggyorsabb megoldást választottuk megvalósításukkor. Mostanra talán az vált a legfőbb hátrányukká, hogy ebből adódóan nem állnak össze átgondolt, egységes rendszerré.

Külön-külön vizsgálva őket azt tapasztalhatjuk, hogy – érthető módon – némelyikük mára egészen letisztult, feladatai részleteiben is világossá váltak, mások azonban még nem ennyire kiforrottak. A nagy házi feladat ellenőrző, legelsőként elkészített – ám azóta folyamatosan átalakított és továbbfejlesztett – rendszer például önálló egységként tekintve alapvetően logikus struktúrájú, koncepcionálisan tiszta, még ha a programozási munka hagy is némi kívánnivalót maga után.

Mostanra az automatizált és automatizálni kívánt feladatok száma elért egy olyan szintet, hogy érdemes a teljes feladatcsoportot újra átgondolni, és az alapoktól kezdve egy egységes rendszerbe fogva megtervezni. Erről lesz szó az 5. fejezetben.

5. fejezet

Az ETS rendszer terve

A jelenleg használt programok ismertetését követően rátérhetek a szakdolgozatom magját képező részre, egy új, egységes rendszer tervének ismertetésére. Hogy tisztábban lássuk a rendszer feladatát, céljait, elsőként megkísérlem elhelyezni az oktatást támogató rendszerek világában, és felfedem, hogy mit jelent a fejezet címében használt ETS rövidítés. Ezt követően rátérek a rendszer részletes tervére, amelyet egy átfogó képtől a részletek felé haladva ismertetek. A tervezés során végig törekszem arra, hogy a rendszer lehetőleg tárgyfüggetlen legyen, sőt bizonyos részek esetében még azt sem használom ki, hogy programozási nyelvek oktatásáról van szó. Azokon a pontokon, ahol egy-egy tervezési döntés meghozatalához konkretizálni kell a feladatot a tárgyra vagy akár a két deklaratív nyelv egyikére, ott ezt külön jelzem.

A tervezés során általában nem használom ki, hogy milyen eszközökkel fog megvalósulni a rendszer. Ahol – bizonyos részletkérdéseknél – ezt mégis megteszem, ott többnyire a Prolog nyelvet fogom javasolni. Ennek okai a következők:

- Az oktatók természetesen jobban ismerik a Prologot, mint a kifejezetten adminisztrációs feladatokra tervezett nyelveket, például a Perlt. Így könnyebben átlátják a programokat, szükség esetén maguk is tudják módosítani őket.

- Prolog-programokhoz kifejezetten könnyű „konfigurációs” állományt írni: egy Prolog-program könnyen be tud olvasni állományokból Prolog-kifejezéseket, eljárásokat. Ennek járulékos hatása, hogy a beállításokat – szükség esetén – nem csak tényállításokkal, hanem programokkal is megadhatjuk (például dinamikusan számíthatjuk egy állomány elérési útját).

- Az olyan részfeladatok ellátásához, amelyekhez egy Unix-os szöveges program (például a `grep`) alkalmasabbnak bizonyul, „ki lehet hívni” a Prolog-keretből. Ebben a tekintetben sem rosszabb tehát, mint más nyelvek.

5.1. Névválasztás

Amint látni fogjuk, a rendszer által ellátandó funkciók tulajdonképpen egy *tanársegéd* feladatainak felelnek meg. Éppen ezért az *Elektronikus Tanársegéd*, azaz ETS elnevezést javaslom. Az ETS rövidítés előnye, hogy emlékeztet a közismert ITS-re, és több olyan angol név is adható, amelynek szintén ez a rövidítése, mint például Electronic Tutoring System vagy Environment for Training Students. A továbbiakban az ETS betűszóval fogok hivatkozni a tervezendő rendszerre.

5.2. A rendszer elhelyezése

5.2.1. Összevetés más, oktatást támogató rendszerekkel

A jelenleg használt programokról szóló fejezetet olvasva talán már az olvasóban is kialakulhatott az a vélemény, hogy a tervezendő rendszer nem sorolható be egyik kategóriába sem a 2. fejezetben említettek

közül. Mielőtt azonosítanám a rendszer feladatait, elsőként azokra a különbségekre szeretném felhívni a figyelmet, amelyek egy új kategória, az ETS létrehozását indokolják.

Miért nem ITS?

A BME-n a képzés a hagyományos előadótermi stílust követi, és ezen a *Deklaratív programozás* tárgy oktatói sem szeretnének változtatni. Az ITS-ek esetében azonban teljes értékű oktatásról, *új ismeretanyag átadásáról* van szó, nem pedig a már – elméletileg – meglévő megerősítéséről és gyakoroltatásáról. Az említett tanuló-modellre a mi esetünkben nincsen szükség, hiszen nem kell nyomon követnünk egy-egy hallgató fejlődését és előrehaladását, hanem elegendő bizonyos jól definiált szinteken ellenőrizni a tudás meglétét. Ennyiben tehát egy ITS *több*. Ugyanakkor az ITS-ek nem látnak el olyan feladatokat, mint például a házi feladatok ellenőrzése, vagy a félév során megszerzett pontok összesítése. Erre ott nincs is szükség, hiszen a kurzus elvégzése nem áll másból, mint az ITS összes tesztjének megfelelő szintű teljesítéséből, nálunk azonban más a helyzet.

Miért nem CILE?

Az új rendszer – elsődleges – célja és feladata nem az, hogy az előadótermi munkát könnyítse, hanem hogy a háttérmunkákat lássa el minél önállóbban. Órai demonstrációkat az oktatók már most is tartanak: egy-egy szárazabb vagy nehezebben emészthető témakörnél színesítik az elhangzottakat azzal, hogy „élesben” is bemutatják az adott algoritmus vagy nyelvi eszköz működését. Ehhez azonban nincs szükség másra, mint egy Prolog- vagy SML-értelmezőre és egy pár frappáns példaprogramra.

Mért nem CAT?

A gyakoroltatás a tervezendő rendszer fontos – talán legfontosabb – részét képezi, e tekintetben tehát talán leginkább a CAT rendszerekhez hasonlítható. Ugyanakkor nem statikus, azaz például a házi feladatok félévről félévre változnak, és ezeket a változásokat természetesen tükrözni kell a rendszerben is. A tervezés egyik célja, hogy az ilyen frissítések a lehető legkevesebb emberi munkát igényeljék, ezt azonban teljesen nem lehet megspórolni. Természetesen egy CAT rendszer is változhat, mégis az ilyen változások inkább a rendszer fejlesztését, fejlődését jelentik, nem pedig az üzemeltetésből adódnak.

A rendszerünk másik fontos feladata az adminisztratív teendők támogatása. Ez a feladatkör semmiképpen sem sorolható a CAT rendszerek által lefedett területhez.

Miért nem hallgatói információs rendszer?

A különbség ezen a ponton már annyira magától értetődő, hogy talán nem is kellene magyaráznom. Mégis érdemesnek tartom megemlíteni, hogy a rendszer a tervek szerint csak olyan adminisztratív feladatokat fog ellátni, amelyek a NEPTUN-ból valamilyen okból kimaradtak. A NEPTUN által ellátott feladatok kapcsán ugyanakkor szükséges lehet a két rendszer közötti együttműködés, amelybe a hallgatók névsorának átvételétől a vizsgajegyek visszaírásáig bezárólag sok minden belefér.

5.2.2. Az ETS feladatai

A 3.2. fejezetben alaposan megvizsgáltuk az oktatás egyes eszközeit és a hozzájuk kapcsolódó, automatizálható feladatokat is. Ezek a feladatok a tervezendő rendszer egy-egy funkciójaként jelennek majd meg. Most az osztályozhatóság kedvéért ismét tekintsük át őket, ezúttal kissé más megvilágításban.

A fő tennivalókat alapvetően két csoportba lehet osztani, bár egy-két esetben nehéz lehet eldönteni, hogy az adott feladat melyik csoportba is tartozik. A két csoport egyikébe a programozási nyelvhez szorosabban kapcsolódó tennivalók tartoznak, mint például a beküldött programok vagy a gyakorló feladatokra adott megoldások ellenőrzése. A másik csoportba az alapvetően adminisztratív jellegű teendők tartoznak, mint például a ZH- és vizsgafeladatokra adott pontszámok nyilvántartása. Ez a felosztás abban is a segítségemre lesz, hogy amikor szükséges, megállapíthassam, hogy mely tennivalók függetleníthetők teljesen a tárgytól, és melyek nem. Most viszont segít láttatni, hogy az ETS feladatköre valóban egy tanársegédé. Nézzük át még egyszer, címszavakban, hogy melyek ezek a feladatok!

1. Adminisztratív teendők:
    - névsor, elektronikus levélcímek nyilvántartása;
    - ZH-jelentkezések, jegyzetrendelések összegyűjtése;
    - ZH- és vizsga-eredmények regisztrálása;
    - pontok nyilvántartása, összesítése;
    - hallgatók értesítése különböző eseményekről;
    - vizsgajegy megállapítása.
2. A tananyaggal kapcsolatos teendők:
    - házi feladatok ellenőrzése;
    - gyakorló kérdések feladása, a válaszok ellenőrzése;
    - ZH- és vizsgasorok összeállítása.

5.3. A fő komponensek áttekintése

A tervezés során elsőként azt kell eldönteni, hogy a létrehozandó ETS rendszer alapvető szerkezete milyen legyen. Lehetséges lenne egy monolit rendszer létrehozása is, én azonban – és ezzel, azt hiszem nem vagyok egyedül – sokkal jobbnak és rugalmasabbnak tartom a moduláris, kisebb komponensekből építkező architektúrákat, ezért ezúttal is ezt a módszert választom. Ha pedig modulokból építkezünk, meg kell állapítani, hogy pontosan milyen komponensekre lesz szükség, s csak ezután láthatunk neki a részletek kidolgozásának. Ebben a részben erről lesz szó.

Az adatbázis

Nyilvánvaló, hogy minden feladat alapvetően egyes hallgatókhoz kapcsolódik, ha úgy tetszik, az adatműveletek legnagyobb önálló egysége a *hallgató*. Az egyes feladatok is csak és kizárólag a hallgatókon keresztül kapcsolódnak egymáshoz, pontosabban szólva a hallgatók által elért eredményeken keresztül. Az egyetlen logikus következtetés ezekből a megfigyelésekből az, hogy az új rendszer lelke csakis egy, a hallgatók adatait, elsősorban pontszámait nyilvántartó adatbázis lehet. Ez az az egységes, közös adatbázis, ami jelenleg hiányzik, ezért sem állhattak össze a más-más funkciójú komponensek rendszerré. A komponensek közül elsőként tehát magát az adatbázist érdemes megtervezni, összegyűjteni a benne tárolandó értékeket és kialakítani a szerkezetét. Egy jól megtervezett, a jelenlegi funkciókat kiszolgáló, a jövőbeni bővítéseknek teret hagyó adatbázis létfontosságú, ha jól működő, könnyen fenntartható és menedzselhető rendszert akarunk kapni.

A központi adatbázishoz kapcsolódik a többi komponens, amelyek – a moduláris felépítésnek köszönhetően – tetszőleges programozási nyelven valósíthatók meg (akár komponensenként más-más nyelven), és ízlés szerint módosíthatóak a rendszer élete során mindaddig, amíg az adatbázison keresztül megfelelő módon tartják a kapcsolatot a többi komponenssel. Hogy melyek ezek a komponensek, azt legjobban a feladatok analizálásával és csoportosításával tudjuk megállapítani.

Az 5.2.2. fejezetben a tananyaggal kapcsolatban három feladatot azonosítottunk. Mivel ez a három feladat lényegesen különbözik egymástól, logikusnak látszik, ha három külön komponenst rendelünk hozzájuk. Az adminisztratív feladatok közül néhány szorosan kapcsolódik ezekhez, ezért azt javaslom, hogy az ilyen feladatokat is ide soroljuk. Közülük egyeseket – mint például a hallgatók értesítését – szükséges lehet többfelé osztani, hiszen több komponenshez is kapcsolódnak. A megmaradó, valamint az adatbázis létéből adódó újabb adminisztratív feladatokat rendeljük egy negyedik, tisztán adminisztratív komponenshez. Ekkor a következő négy komponens alakul ki:

1. házi feladatok fogadása, ellenőrzése, értékelése, éles tesztelés esetén a pontszámok regisztrálása, a hallgatók értesítése az eredményekről, hasonlóságvizsgálat;

2. gyakorló feladatokkal kapcsolatos tennivalók, a hallgatók haladásának követése és adminisztrálása;

3. zárthelyi-, ill. vizsga-feladatsorok (névre szóló) generálása, a pontszámok adminisztrálása, vizsgáztatás könnyítése kísérőlap generálásával, a pontszámok nyilvántartásával és a vizsgajegy kiszámításával;

4. adatbázis-hozzáférés, névsor és elektronikus levélcímek nyilvántartása, jegyzetrendelések, ZH-jelentkezések és egyéni preferenciák adminisztrálása.

Az alábbiakban mind a négy komponensről szólok pár szót, mielőtt belefognék a részletes megtervezésükbe.

A házi feladatokat értékelő komponens

A komponens legnagyobb részét az a program teszi ki, amelynek jelenleg is van egy jól működő megvalósítása. Mégis érdemes a teljes komponens felépítését újra átgondolni, a következők miatt:

- A nagy és kis házi feladatok fogadását jelenleg két külön program végzi. Ez az állapot nem éppen ideális, születésétől kezdve átmenetinek tekintjük. Oka egyedül a rövid határidő volt, most, hogy alapos átgondolásra van mód, feltétlenül érdemes foglalkozni a kettő egyesítésével.

- Mivel eddig nem létezett adatbázis, az eredmények különböző szöveges állományokba kerültek. Most újra kell tervezni azokat a részeket is, amelyek az eredményeket tartják nyilván.

- A program megvalósításához a `bash` nevű Unix shell szkript-nyelvét választottuk, és az áttekinthetőség kedvéért több kisebb állományra tördeltük. A feldarabolás kezdetben koncepciózus volt, ám ahogy a program fokról fokra módosult, a felosztás némileg értelmét vesztette, vagy legalábbis átstrukturálásra lett volna szükség. Utólag visszagondolva úgy látom, hogy a fizikai feldarabolás helyett célszerűbb lett volna csupán logikailag felosztani a programot, és az egyes részfeladatokat az egyazon állományon belül elhelyezkedő különböző rutinok belsejében megvalósítani.

Természetesen ide tartozik a házi feladatok beadását lehetővé tevő program, a webfelületen keresztül a teszteredményekhez hozzáférést biztosító CGI szkript, valamint a hasonlóságvizsgálatot megoldó program is.

A gyakorló feladatokért felelős komponens

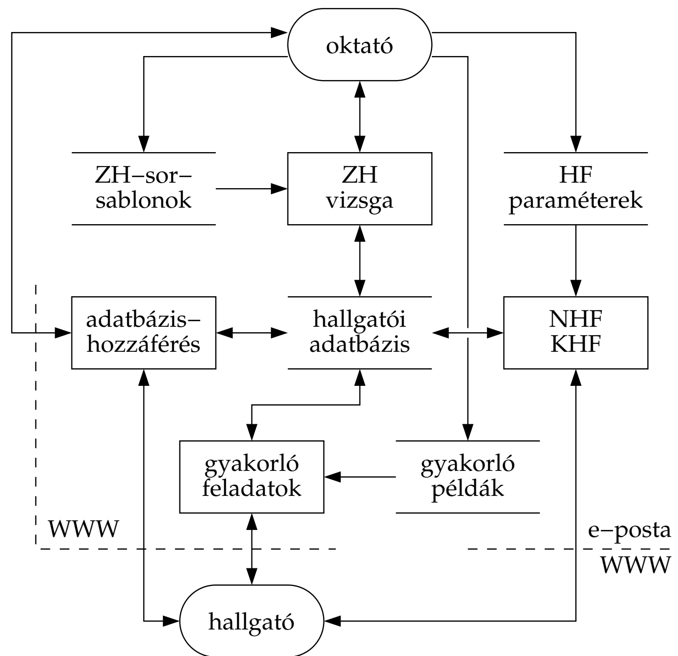
Ez a programegyüttes legfiatalabb tagja, kísérleti stádiumban. Ebből adódóan újratervezésről nem lehet szó, hiszen nincsenek mérvadó tapasztalatok a használatával kapcsolatban. Az oktatóknak ugyanakkor határozott elképzeléseik vannak a funkcióját, működésének főbb jellemzőit illetően, így ez alapján, valamint a már elkészült programrészekben felhasznált ötleteken okulva elfogadhatóan pontos tervek készíthetők egy ideális alrendszerről.

A ZH-kkal és a vizsgákkal kapcsolatos feladatokat ellátó komponens

A névre szóló feladatlapok generálásától eltekintve ez a komponens alapvetően adminisztratív jellegű, a feladatok többségét a központi adatbázishoz kapcsolódó egyszerű adatfeltöltő és lekérdező program el tudja látni. A generáláshoz viszont szükség van egy jól szervezett feladatkészletre is, amelynek feltöltéséhez ugyan lehet segítséget nyújtani, mégis alapvetően emberi munka.

A hallgatók és oktatók adatbázis-hozzáférését biztosító komponens

Természetesen a többi komponens is olvassa és módosítja az adatbázist, ahogy éppen szükséges. De ez a komponens az, amelyik *hasznosítja* is, hiszen rajta keresztül érhetők el a benne tárolt értékek. Fontos, hogy a hallgatók meg tudják nézni az eddig megszerzett pontjaikat, ill. hogy az oktatók különböző szempontok szerint csoportosítva le tudják kérdezni az évfolyam egészére vagy az egyes hallgatókra vonatkozó adatokat. Mindezt egy webfelületen keresztül a legcélszerűbb elérhetővé tenni. E mellett az adatbázisban tárolható még számos olyan információ, amely nem pontszám-jellegű – ezek módosítását és lekérdezését is itt kell megoldani.



5.1. ábra. A fő komponensek kapcsolódása

A komponensek kapcsolódása

A központi adatbázison kívül több adatbázisszerű állomány szükséges az egyes komponensek működéséhez. Ezen adatbázisok, a két fő felhasználói csoport (hallgatók és oktatók), valamint a komponensek kapcsolódását az 5.1. ábra mutatja. Az ábrán a négy fő komponenst négy téglalap reprezentálja, ezek mind kapcsolódnak a központi szerepet betöltő, két párhuzamos vonallal jelölt adatbázishoz. A négy komponens közül háromhoz egy-egy további adatbázis kapcsolódik, amelyekben paraméterek, bemeneti adatok vannak, ezeken keresztül hangolható az egyes komponensek viselkedése. A lekerekített sarkú téglalapok jelölik a hallgatókat és az oktatókat, akik jól definiált felületen keresztül férnek hozzá az egyes komponensekhez, ill. az adatbázisokhoz. A hozzáférés két felületét, nevezetesen a Webet és az elektronikus levelezési felületet külön jelöltem egy-egy szaggatott vonallal, a nyilak az adatok áramlásának irányát jelzik.

5.4. A hallgatói adatbázis

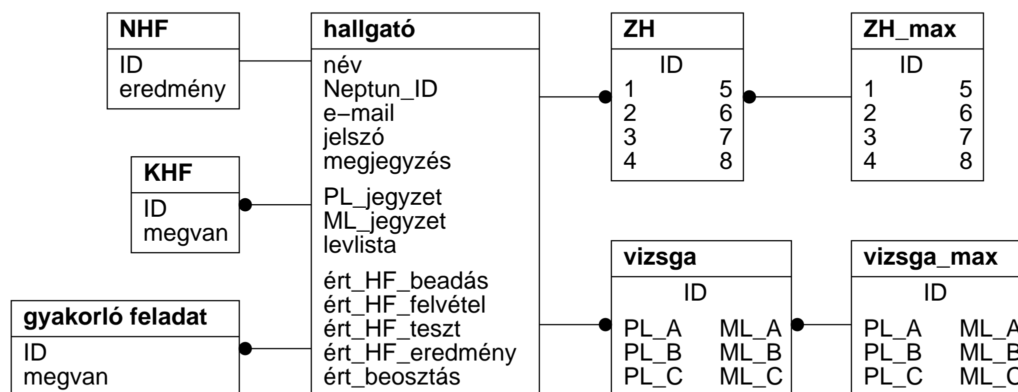
Az adatbázis létrehozásának alapvető célja, hogy a hallgatók által megszerzett pontszámokat egy helyen összegyűjtve lehessen tárolni. Ennek megfelelően az adatbázis mezőinek döntő többsége valamelyik pontszámot fogja tárolni. Ha azonban már amúgy is van egy adatbázisunk, kézenfekvő ötlet minden egyéb járulékos adatot is itt tárolni. Ezen adatok egy része olyan, hogy magának a hallgatónak kell tudnia (be)állítani, mások csak az oktatók által hozzáférhetők.

Miért van szükség ezekre a mezőkre? Ha az oktatók szeretnének valamit közölni a hallgatókkal, azt könnyen megtehetik, akár előadáson, akár a tárgy honlapján, akár a tárgyhoz rendelt elektronikus levelezési listán keresztül. A fordított irányú információáramlás azonban nehézkesebb. A hagyományos módszerek közé tartozik a különböző ívek körbeadása és az elektronikus levélen keresztüli információgyűjtés, azonban mindkettőnek megvan a maga hátránya. Az ívek körbeadásával az a gond, hogy érthető módon csak azok tudnak föliratkozni, akik jelen vannak az előadáson (vagy megkérik valamelyik előadás-járó barátjukat). Az oktatóknak küldött elektronikus levelek pedig feleslegesen lekötik az oktatók figyelmét kézhez vételükkor, később pedig körülményes a névsor összeállítása. Nos, a hátrányokat talán kissé eltúloztam, hiszen ezek azért gyakran alkalmazott és alapvetően jól bevált módszerek, de a saját tapasztalatom az, hogy a webes regisztráció mindig lényegesen hatékonyabb és a hallgatók köré-

ben is népszerűbb. A BME Híradástechnika Tanszékén például – Mihály Zsigmondnak köszönhetően – a különböző mérésekre való jelentkezés egy ügyes webfelületen keresztül van megoldva, így a hallgatók akkor és onnan jelentkeznek, amikor és ahonnan csak akarnak (feltéve persze, hogy van Internet hozzáférésük), ezáltal elkerülhető a felesleges sorbanállás is. A lényeg tehát az, hogy ha biztosítunk egy webes felületet a különböző jelentkezések lebonyolítására, és az igényeket bevisszük az adatbázisba, az a hallgatók és az oktatók szempontjából is kedvező.

5.4.1. A terv kialakítása két lépésben

Az adatbázis szerkezetének kialakításakor természetesen a *Deklaratív programozás* tárgy struktúrájából indultam ki. Első lépésként egy olyan adatbázist terveztem, amely képes tárolni a jelenleg alkalmazott számonkérés-fajták eredményeit, illetve egy-két járulékos információt a hallgatóról. Ennek eredményét mutatja az 5.2. ábra, amelynek részletes ismertetésétől eltekintek. Bemutatásával az a célom, hogy láthatóvá tegyem, hogyan jutottam el a végeredményként kapott általános struktúrához. Az ránézésre is jól látszik, hogy ez a szerkezet nagyon szorosan kötődik a jelenlegi tárgystruktúrához, és a kisebb változásokat is nehezen lenne képes követni.



5.2. ábra. Egy specializált adatbázis szerkezete

A végső struktúráról áttekintést nyújtó entitás-reláció diagram látható az 5.3. ábrán. Hogy milyen tartalmat takarnak az egyes mezőnevek, és hogy az egyes relációk számossága miért pont az, ami, egyelőre nem fontos, de mindenre fény derül a továbbiakban az adatbázis tartalmának részletezésekor. Az áttekinthetőség érdekében osszuk az adatbázis mezőit három fő csoportra, és vizsgáljuk őket külön-külön. Ez a három csoport a *személyes adatok* csoportja (ebbe a hallgató nevű entitásban tárolt mezők tartoznak), a különböző *események* adatait rögzítő mezők csoportja, és végül a *levelezési listát* adminisztráló mezők csoportja.

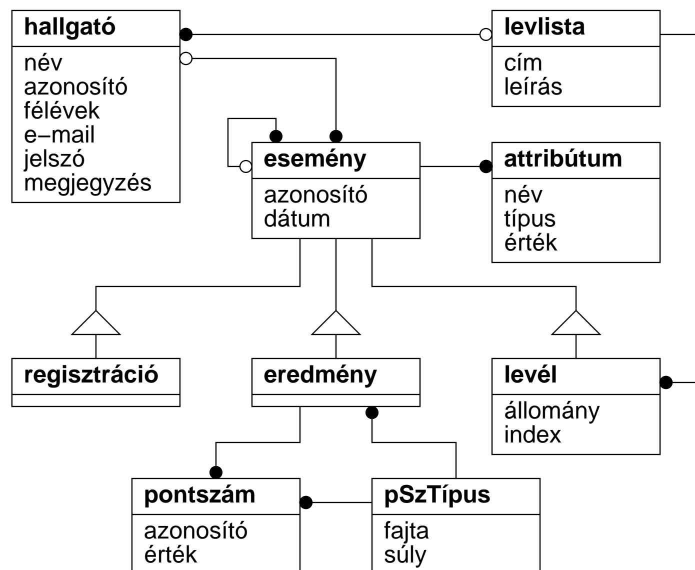
5.4.2. „Személyes” adatok

Az adatbázis alapját azok a mezők képezik, amelyek a hallgatók azonosítására szolgálnak. Ezek az alábbiak:

- a hallgató neve;
- egy egyértelmű azonosító, pl. a hallgató NEPTUN kódja;
- azon félév(ek) azonosítója, amely(ek)ben a hallgató felvette a tárgyat;
- elektronikus cím az elérhetőség kedvéért.

E három mező mellett a hallgató entitás további két mezőt tartalmaz:

- jelszó a hallgató által beállítható adatok védelméhez;
- megjegyzés a tanárok észrevételeinek tárolásához.



5.3. ábra. Az általánosított adatbázis felépítése

5.4.3. Események

Az adatbázis magját a félév közben bekövetkező különféle *események* adatai alkotják. Amint hamarosan látni fogjuk, az események több fajtáját különböztethetjük meg. Ezen fajták közös jellemzőinek leírására szolgál az esemény entitás, amelynek két mezője van:

- az esemény – az adott fajtán belül – egyértelmű azonosítója;
- egy dátum, amely időponthoz köti az eseményt (esetleg érdemes lehet tárolni a félév azonosítóját is, mert nem biztos, hogy ez a dátumból kinyerhető).

Az események leggyakrabban egy-egy hallgatóhoz kapcsolódnak, de ez nem szükségszerű. Egy hallgatóhoz ugyanakkor számos esemény kapcsolódhat. Ezt a kapcsolatot fejezi ki a diagramon az „nulla-vagy-egy – több” reláció.

Egyes események más eseményekhez is kapcsolódhatnak, ezt jelzi a visszahurkolódó „nulla-vagy-egy – több” reláció. E kapcsolat értelme nyilvánvalóan függ az adott esemény jellegétől.

Az eseményekhez ún. *attribútumok* is tartozhatnak, amelyekkel az adott esemény tetszőleges adatát le tudjuk írni, ezek tárolására szolgál az attribútum entitás. Minden attribútumnak van egy

- neve;
- típusa, amely megadja, hogy az adott attribútum milyen típusú értéket tárol (számot, szöveget, igazságértéket stb.);
- értéke.

Arra, hogy az ilyen attribútumokkal konkrétan miket tudunk leírni, az egyes eseményfajták tárgyalásánál láthatunk példákat.

Most nézzük meg, hogy melyek lehetnek az eseményfajták. A jelenlegi tárgystruktúrában hármat tudtam értelmes módon elkülöníteni, de ez nem jelenti azt, hogy a későbbiekben ne lehetne ennél többet is megkülönböztetni. A különböző fajtákat leíró entitások „is-a” relációban állnak az esemény entitással, azaz mindegyik rendelkezik ez utóbbi összes jellemzőjével, és ehhez adnak hozzá továbbiakat (minimálisan azt az információt, hogy milyen eseményfajtáról van szó).

Regisztrációk

A legegyszerűbb eseményfajta az ún. *regisztráció*. Ez az esemény teszi lehetővé a fejezet bevezetőjében is említett, a hallgatóktól az oktatók felé irányuló információáramlást . Az ezt leíró entitásnak nincsenek

további mezői, hiszen az esemény azonosítója elegendő a regisztráció *tárgyának* megadásához, szükség esetén azonban kapcsolódó attribútumokkal tetszőleges paraméter megadható. Ilyen entitásban a jelenlegi tárgyszerkezet mellett a következők adminisztrálását tudom elképzelni:

- Mint már többször szóba került, a *jegyzetrendelés* jelenleg is webfelületen keresztül zajlik, de a generált nyilvántartás szöveges alapú. Egy egységes központi adatbázis létrehozása esetén érdemes ezt itt tárolni. Ehhez a jelenlegi tárgyhoz az entitás két példányára van szükség, egyre a Prolog- és egyre az SML-jegyzet rendeléséhez.

- A különböző események kapcsán a hallgatók különféle dolgokról kérhetnek és kaphatnak automatikus *értesítést*. Ilyen például az egyes házi feladatok beadási határidejéről szóló automatikus értesítés, a tesztek végeredményét ismertető naplóállomány, vagy a szóbeli vizsga terem- és időbeosztása. Az egyes értesítések igénylését egy-egy regisztráció entitással lehet megadni.

- A *ZH-jelentkezések* is remekül leírhatók egy-egy ilyen entitással.

Eredmények

Talán a legfontosabb eseményfajta a hallgató által elért *eredmény*. Ennek segítségével tároljuk a hallgató által megszerzett pontokat, amelyek alapján végül összeáll a vizsgajegy. Fontos, hogy ezeket az adatokat olyan formában tároljuk az adatbázisban, hogy egy újabb számonkérésfajta bevezetése vagy egy meglévő formai átalakítása az adatbázis szerkezetének módosítása nélkül azonnal tükrözhető legyen a tárolt adatokban. Ennek érdekében az eredményeket három egymáshoz kapcsolódó entitással írjuk le.

Az egyes pontszámok különböző számonkérés-típusokhoz kapcsolódnak (ilyen a ZH, a vizsga és a házi feladat is), amelyeket egységesen egy-egy eredmény entitással írunk le. Az esemény entitás két attribútuma mellett továbbiakra általában nincs szükség, az esetleges járulékos információkat attribútumok segítségével adhatjuk meg. Ilyen lehet például a beadott NHF-programok futási ideje – akár tesztesetekre lebontva –, a generált naplóállományok elérési helye, vagy a gyakorlás során a hallgató által adott megoldások listája.

Az eredmények olyan események, amelyek kapcsolódhatnak más eredményekhez is, ilyenkor a fő eredmény összpontszámába be kell számítani az aleredmények összpontszámait is. (Ezzel a módszerrel remekül le lehet írni a vizsgát, amelynek eredményébe nem csak az aznapi teljesítmény számít bele, ld. az 5.4. ábrát.)

Pontszámok. Az eredményeket alkotó pontszámokat egy-egy pontszám entitás definiálja, a két entitást összekapcsoló reláció „egy – több” jellegű. Minden pontszámnak van egy
- azonosítója, amely tulajdonképpen a részfeladat sorszáma vagy betűjele;
- és egy értéke.

Pontszámtípusok. Mind a pontszámokhoz, mind az eredményekhez hozzárendelünk egy-egy *pontszámtípust*, amely megadja, hogy az adott pontszámot hogyan kell értelmezni, ill. hogy az eredményhez kapcsolódó pontszámokat milyen módon kell összegezni. Ezt a pSzTípus entitás definiálja.

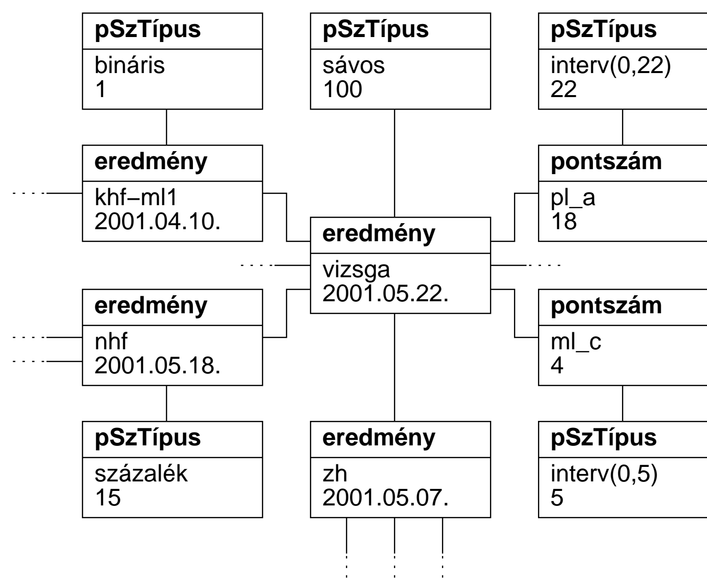
Mit jelent a pontszámtípus *fajtája*? Ha az adatbázisban csak egy pontértéket tárolunk, annak értelmezése az adatokat feldolgozó programon múlik. Ezt célszerű elkerülni, ezért tároljuk a fajtát is:

- **százalék**: a maximális pontszám ismeretében a kapott pontszámot úgy skálázzuk, hogy a 0–100 intervallumba essék;
- **intervallum**: az előzőhöz hasonló, de maga az intervallum a típus paramétere (a százalék ennek a típusnak egy speciális esete);
- **sávos**: a leképezés a százalékos eredmény és a kapott pontszám között nemlineáris, ilyenkor egyes százalék-intervallumokhoz meg kell adni egy-egy pontértéket (pl. osztályzatok esetében 0–39: 1, 40–54: 2, 55–69: 3, 70–84: 4, 85–100: 5);
- **bináris**: „igaz” (vagy 1) az érték, ha a hallgató a feladatot hibátlanul teljesítette, „hamis” (vagy 0) egyébként.

E négy fajta segítségével minden ismert pontszámtípust le lehet fedni, sőt az egyes fajták között vannak átfedések. Alapvetően csak egész értékeket érdemes tárolni, fél pontok osztogatásának nem sok értelme van. Ha ugyanis finomabb felbontás szükséges, akkor semmi akadálya, hogy a maximális pontszám a korábbi duplája legyen.

A pontszámtípus *súlya* azt adja meg (mondjuk százalékban), hogy az adott pontszám milyen mértékben vesz részt egy eggyel magasabb szintű összpontszám kialakításában. Erre azért van szükség, mert előfordulhat például, hogy a ZH nullától százig pontozódhat, de a vizsga összpontszámának – legyen mondjuk ez is maximum száz – csak a harmadrészét alkotja. Ebben az esetben a ZH-ra kapott pontszámot természetesen el kell osztani hárommal. Ennek jelzésére szolgál a súly, amely itt 33 lenne.

Az 5.4. ábrán látható, hogy hogyan lehetne leírni a jelenlegi vizsgát a fenti három entitással. Az ábrán nem látható a teljes adatbázis vonatkozó része, csak egyes részletek, a bemutatás kedvéért.



5.4. ábra. A vizsgajegy kialakulása az adatbázisban

Levelek

A levél mint esemény bemutatását a következő szakasz végére hagyom, mert a harmadik fő csoport ismertetése nélkül nem tárgyalható.

Egyéb események

Azok az események, amelyek nem sorolhatók a három konkrét kategória egyikébe sem, leírhatók egy általános esemény entitással, a járulékos adatok pedig attribútumként adhatók meg.

5.4.4. Levelezési lista

A tantárgyhoz tartozik egy *levelezési lista*, amelyet jelenleg a mailman nevű közismert Unixos levlistagondozó programmal működtetünk. Erre a hallgatóknak külön fel kell iratkozniuk. Ha azonban amúgy is tároljuk az e-mail címüket, semmi akadálya annak, hogy a levelezési listát akár közvetlenül az adatbázisban tárolt címlista alapján működtessük. A lista nyilvántartásának középpontjában a levlista entitás áll, ennek mezői a következők:

- a lista e-mail címe;
- rövid ismertető a lista témájáról.

A lista járatása igény szerinti, ezért az entitást a feliratkozott hallgatókhoz kapcsoló reláció is „nulla-vagy-egy – több" jellegű.

A lista lényegét természetesen a listára küldött levelek jelentik, amelyeket ugyan szöveges állományokban tárolunk, de valami minimális információt az adatbázisban nyilvántartunk róluk, az esemény entitás egy harmadik fajtája segítségével.

Levelek. A harmadik eseményfajtát tehát a levelezési listára küldött levelek alkotják. Leírásukra szolgál a levél entitás, amelynek az esemény entitástól örökölt azonosítójában megadhatjuk az adott levél sorszámát, a megfelelő relációval hozzárendelhetjük ahhoz a hallgatóhoz, aki küldte (ha hallgató volt), és további mezőkkel leírhatjuk

- azon állomány elérési útját, amelyben maga a levél található;
- a levél sorszámát ezen az állományon belül (általában több levelet szokás egy állományban tárolni).

A gyorsabb keresés érdekében egy attribútum segítségével a levél tárgyát is tárolhatjuk, sőt a „visszahurkolódó" reláció segítségével téma szerint szálakba is rendezhetjük a leveleket.

Egy bináris értéket tároló attribútum csatolásával azt is nyilvántarthatjuk a levelekről, hogy részét képezik-e egy FAQ-csoportnak: ide azok a levelek tartoznak, amelyek fontos és gyakran visszatérő kérdéseket feszegetnek. Ha lekérdezhető az ilyen attribútummal rendelkező levelek listája, akkor már meg is van oldva egy FAQ lap összeállítása.

5.4.5. A megvalósításról

Noha entitás-reláció diagramon ábrázoltam az adatokat, a megvalósításnak nem kell relációs adatbázison alapulnia. Igaz, kézenfekvőnek látszik valamely megszokott, SQL-t ismerő relációs adatbázis-kezelő alkalmazása, a teljes ETS Prolog-beli megvalósítása azonban más megoldást sugall. A SICStus Prolog könyvtárai között megtalálható egy olyan modul, amely az ún. *Berkeley DB* struktúrát támogatja. Ennek segítségével – az adatbázis létrehozásakor meghatározott funktorú – Prolog-kifejezéseket tárolhatunk indexelt állományokban. Mivel a szóban forgó adatbázis mérete nem túl nagy, nem jelent veszteséget az sem, ha esetleg a Berkeley DB hatékonysága rosszabb, mint mondjuk a *PostgreSQL* relációs és objektum-orientált adatbázis-kezelőé. (Ez puszta feltevés, nem végeztem méréseket e tekintetben.)

Az sem előírás a megvalósításkor, hogy az ábrán és a leírásban összekapcsolt entitások külön táblában (vagy Prolog-kifejezésben) legyenek. A sokat emlegetett Prolog-megvalósítás esetén lehetséges egy olyan megoldás is, hogy a tárolt kifejezésekben a megfelelő helyen egy-egy lista áll, amely természetesen akárhány részkifejezést tárolhat. Egy ilyen lehetséges Prolog-kifejezés látható az 5.5. ábrán. Itt a regisztrációkat és az eredményeket, azaz azokat az eseményeket, amelyek pontosan egy hallgatóhoz kapcsolódnak, a hallgatót leíró `student/7` funktorú kifejezés egy-egy argumentumában elhelyezett listában tároljuk. A pontszámtípusokat nem ebben a struktúrában tartjuk nyilván, hanem elkülönítve (ez az ábrán nem látszik), a két struktúra közötti kapcsolatot az eredmények azonosítói (`nhf`, `zh` stb.) jelentik. A levelezési listát nyilvántartó részek szintén nem láthatóak az ábrán.

5.4.6. Az adatbázis feltöltése

Az adatbázis inicializálását a NEPTUN-ban tárolt tárgyfelvételi nyilvántartás alapján lehet elvégezni. Innen automatikusan kinyerhető a hallgatók neve és NEPTUN kódja. Az egyéni beállítások értelmes alapértelmezett értékkel inicializálódnak, ezután a hallgatók maguk állíthatják őket az 5.8. fejezetben bemutatandó webes felületen keresztül. A pontszámok természetesen fokozatosan kerülnek be az adatbázisba, ahogy az ETS egyes moduljai aktiválódnak.

5.5. Házi feladatok feldolgozása

Ahogy a fejezet bevezetőjében jeleztem, a cél egy olyan értékelő program megtervezése, amely egyaránt képes a jelenlegi nagy és kis házi feladatokat kiszolgálni, mindezt kényelmesen, könnyen konfigurálhatóan teszi, és lehetőséget hagy a házi feladatok palettájának további bővítésére is. Ehhez a jelenlegi

```
student(
   'Lapos Elemér',        % név
   lapose,                % NEPTUN kód
   'elapos@mail.hu',      % e-mail cím
   '******',              % jelszó
   [],                    % megjegyzések listája
   [                      % regisztrációk listája
    pl_jegyzet,           % rendel Prolog-jegyzetet
    levlista,             % fel van iratkozva a levlistára
    hf_teszt,             % ért. a HF-ek teszteléséről
    hf_eredmeny,          %      a HF-ek végeredményéről
    beosztas              %      az időbeosztásokról
   ],
   [                      % az eredmények listája
    nhf([pl-60,           % NHF Prolog-részének eredménye
         ml-70,           % NHF SML-részének eredménye
         doc-40,          % NHF dokumentáció minősítése
         total-57]),      % NHF összesített eredmény
    khf([pl1-1,           % KHF-ek eredményei
         ml1-0]),
    zh('2001.05.07'-      % a ZH dátuma
       [1-6,              % ... és a feladatok pontjai
        2-2,...]),
    vizsga('2001.05.22'-  % a vizsga dátuma
           [pl_a-18,      % ... és a feladatok pontjai
            pl_b-6,...]),
    vizsgajegy(4)         % a kapott vizsgajegy
   ]).
```

5.5. ábra. Egy lehetséges Prolog BDB bejegyzés

struktúrát alapvetően újra kell gondolni, természetesen az évek során felgyülemlett tapasztalatok figyelembe vételével. Ennél a tervezési résznél érthető módon nem tekinthetek el attól, hogy programozási nyelvek oktatásáról van szó, de ha csak lehetséges, nem használom ki, hogy konkrétan mely nyelvek oktatásáról. Amint látni fogjuk, a tervezés során arra sem építek, hogy egy esetleges megvalósítás milyen eszközökkel, milyen programozási nyelven készül. A saját magam által elkészített és a 6.1. fejezetben bemutatott változat – a bevezetőben foglaltaknak megfelelően – Prologban készült.

Elsőként néhány fogalmat vezetek be, melyek megkönnyítik a tervek tiszta formába öntését, majd azonosítom a komponens szolgáltatásait. Ezt követően javaslatot teszek a komponens állományainak elhelyezésére egy logikus és áttekinthető könyvtárszerkezetben, amelynek jelenlegi, a nagy házi feladatok ellenőrzésére használt programegyütteshez alkalmazott változatát már bemutattam a 4.2. ábrán (15. oldal). Végül, de nem utolsó sorban ismertetem a komponens paramétereit és az ezeken keresztül vezérelhető működését.

5.5.1. Fontosabb fogalmak

Ebben a részben azokat a fontosabb fogalmakat tekintem át, amelyekre a továbbiakban sokat fogok hivatkozni. Bizonyos fogalmak esetében angol elnevezéseket is bevezetek; ezekre a megvalósításkor lehet szükség, ugyanis jobban kedvelem azokat a programokat, amelyek angol neveket használnak az eljárások és változók jelölésére, mert magyar nevek használata esetén „csúnyán” illeszkedik a saját magyar névtér a könyvtárak angol névteréhez. Most pedig lássuk, melyek ezek a fogalmak!

tesztosztály (test class) A tesztosztály voltaképpen nem más, mint az aktuális feladat típusa, azaz a

Deklaratív programozás esetében NHF, első Prolog KHF, második SML KHF stb. Sokszor fog szó esni a *tesztosztály azonosítójáról*, amely egy ehhez jól megválasztott rövidítés, például NHF esetén bhw, első Prolog KHF esetén shw_pl1 stb.

tesztsor (test suite) Mint említettem, egy – adott tesztosztályba tartozó – házi feladatot több tesztsorral is tesztelünk. Egy a beadáskor használatos, egy újabb kell a pontszámok megállapításához, és esetleg még egy a legjobb megoldások versenyeztetéséhez (ezt az oktatók *létraversenynek* szokták nevezni). Ha feltesszük, hogy ez a három tesztsorunk van minden tesztosztályhoz, akkor a tesztsorok *azonosítója* lehet például test, fi nal és hard.

teszteset (test case) Egy tesztsor számozott tesztesetekből áll, a számozás egytől indul. Egy teszteset a kiadott feladat egy-egy konkrét feladványa, amelyre a hallgatók programja lefuttatható. A programra adott pontszám – éles tesztelés esetén – attól függ, hogy a tesztesetek hány százalékát oldotta meg jól, adott időn belül; az időlimitek tesztesetenként állíthatók.

sor (spool) A hallgatók által beküldött programokat nem egyszerre, hanem szekvenciálisan dolgozza fel a számítógép, a tervek jelen állapota szerint tesztsoronként egy szálon. Ehhez a hallgatók leveleit beérkezésükkor sorba kell állítani, és ezt a sort kell folyamatosan feldolgoznia a HF értékelő programnak. Egy sort egy állomány ír le, amelynek szerkezetéről még lesz szó a továbbiakban.

5.5.2. A komponens szolgáltatásai

Ahogy azt a 16. oldalon pontokba szedve leírtam, a jelenleg használt házifeladat-ellenőrző KHF és NHF értékelő része között számos eltérés van. A cél most olyan szolgáltatások kialakítása, amelyek eleget tesznek mind a két tesztosztálynak, sőt ennél többre is képesek. Ebben a fejezetben azonosítom a szolgáltatásokat, és meghatározom, hogy milyen igényeket kell kielégíteniük.

Ezen szolgáltatások közül több küld üzeneteket a hallgatóknak. Hogy az üzenetek megfogalmazása, nyelvezete könnyen módosítható legyen, célszerű őket a komponens többi részétől elkülönítve tárolni. Bizonyos üzeneteket ki kell tölteni adatokkal, például a sorban várakozók számával. Ezek az adatok, ha úgy tetszik, az üzenetek paraméterei, ezért az ily módon elkülönített részeket ezentúl *üzenet-sablonnak* fogom nevezni.

Programok beadása

Az elkészült házi feladatokat továbbra is egy erre a célra készítendő beadó-programmal kell beadni, amely továbbra is félévfüggő, hiszen tartalmazza a névsort, hogy a hallgatóknak csak kiválasztani kelljen saját azonosítójukat, ne beírni, így nincs valós esélye annak, hogy valamelyikük véletlenül elírja. Egy féléven belül azonban ugyanazt a programot kellene használni az összes feladat beadásához. Ez nem különösebben bonyolult, csupán fel kell készíteni a programot arra, hogy felismerje, melyik tesztosztályról van éppen szó, ezután megfelelően becsomagolja az adott állományokat, kiegészítse a szükséges információkkal, és postázza.

Az elektronikus levélként való beadás mellett érdemes létrehozni egy webfelületet is ugyanerre a célra. Ez a lap voltaképpen egy űrlap lenne, amelyen a hallgatónak egy listából ki kell választania a saját azonosítóját, a feladat típusát (azaz a tesztosztályt), és meg kellene adnia, hogy mely állományok azok, amelyeket be kíván küldeni. (A HTML szabvány szerencsére lehetőséget ad állományok *feltöltésére* is.)

Levelek fogadása

A cél az, hogy az összes ilyen tárgyú levelet egyetlen program fogadja – ahogy a KHF-ek esetében most is van –, és így ne kelljen egy újabb feladat kiadásakor a fogadóoldal ezen részén is módosítani. Ehhez a levelek törzsében benne kell, hogy legyen a tesztosztály azonosítója. A programnak ezen kívül ismernie kell az egyes tesztosztályok beadási időszakát, és ezen az időszakon kívül nem szabad elfogadnia az ide érkező leveleket, sőt erről értesítenie kell a levelek feladóit is. Így megoldható, hogy a fogadóprogramot egyszer, a félév elején kelljen csak felparaméterezni, és aztán ne legyen szükség további változtatásokra – legfeljebb a beadási időszakok módosulása esetén.

Programok tesztelése

A tesztelés tesztsoronként külön-külön zajlik. Erre azért van szükség, mert többször előfordult, hogy miközben még folyt az NHF beadása (próbateszteléssel), a háttérben már zajlott a korábban beadottak létraverseny-tesztelése. A próba és a létraverseny teszthez pedig nyilvánvalóan két külön tesztsor tartozik.

A tesztelő legnagyobb része állandó. Vannak azonban olyan részek, amelyek félévről félévre változnak. Ezek egy része csak a feladatosztálytól függ, mások a nyelvtől is. Ilyen nyelvtől függő pl. a *keretprogram* is, amely a hallgatók által beadott programot teljes értékű, futtatható programmá egészíti ki, gondoskodik a tesztesetek és a megoldás belső és nyelvtől független alakja közötti konverzióról, igény esetén méri a futási időt stb.

A KHF-eket csak egy-egy nyelven, míg az NHF-eket mindkét nyelven be kell adni, és ezt a tesztelőnek tudnia kell kezelni. Ennek általánosításaként elvárható, hogy akármelyik tesztosztályra megadható legyen, hogy mely nyelveken kell tesztelni. Ennek érdekében a tesztelő nyelvtől függő részeit megfelelően el kell különíteni. Előfordult párszor, hogy egy beadott NHF egyik felét újra kellett tesztelni. Ehhez az kell, hogy a tesztelőnek meg lehessen adni egy nyelvet egy adott hallgatóra vonatkozóan, a spool állományban. Az ilyen tesztelésből előállított napló érthető módon nem teljes, elhelyezésekor két lehetőség közül választhatunk. Vagy külön állományba írjuk (amelynek nevében is jelezzük, hogy melyik nyelvről van szó), vagy a már meglévő naplóállomány megfelelő részét írjuk felül, a többi részt meghagyva. Külön nyelvi naplóállomány létrehozásakor jobb a teljes naplót tartalmazó állományt törölni, hiszen egy része érvényét veszti, és félrevezető lehet. A meglévő napló módosításához viszont a tesztelőnek fel kell ismernie az egyes nyelvi részeket.

A tesztesetek elhelyezése is eltér a két tesztosztály esetén. Az NHF-ek esetében egy könyvtár külön állományai tárolják az egyes teszteseteket és a referencia-megoldásokat, míg a KHF-ek esetében egyetlen állomány egy-egy sorában van mindez az információ. (Természetesen minden tesztsorhoz tartozik egy ilyen könyvtár vagy állomány.) A tesztelőnek mindkét lehetőséget meg kell engednie bármelyik tesztosztály esetében, sőt elképzelhető olyan lehetőség is, hogy közvetlenül keretprogramba vannak beépítve a tesztesetek, ilyenkor a keretprogram meghívásakor egy argumentumban kell átadnia a tesztsor és a teszteset azonosítóját.

Az időlimitek kezelése is más és más. Az NHF-ek esetében tesztesetenként kell tudni állítani az időlimitet, mert lényegesen különbözik a számítási igényük, és a futási időt is mérni kell. A KHF-ek esetében a futási idő nem lényeges, és az időlimit is csak a végtelen ciklusok kivédése érdekében szükséges, mértéke majdnem teljesen közömbös. Ezért a tesztelőnek meg kell tudnunk mondani, hogy mérje-e a futási időt, és az időlimiteket is meg kell tudni határozni akár tesztesetenként, akár egységesen.

A programok ellenőrzésére is több séma van. Az egyik, hogy – a komponens tesztosztály-függő részét képező – különálló ellenőrző programot használunk, amely egy bizonyos formátumban várja a referencia-megoldást és a beadott program megoldását, és ezek egyezését vagy eltérését jelzi valamilyen módon. A másik lehetőség, hogy az ellenőrzést maga a keretprogram végzi el, ilyenkor a kimenete nem maga a megoldás, hanem közvetlenül az összehasonlítás eredménye. A harmadik lehetőség, hogy a tesztelőbe beépített ellenőrzőt használjuk. Ennek előnye, hogy csökken a tesztosztály-függő részek mérete, ugyanakkor eleget kell tenni bizonyos formai követelményeknek, és szükség van referencia megoldásokra is. A megoldás *algoritmikus* ellenőrzése csak a keretprogramba építve lehetséges.

A tesztelés végeztével a tesztelőnek adott esetben módosítania kell az adatbázist. Ehhez tudnia kell, hogy az adatbázis melyik mezője tárolja az adott tesztosztályra vonatkozó eredményeket.

A tesztelő azon részét, amely állandóan fut a háttérben és várakozik újabb programokra, *tesztdémonnak* nevezem. A Unix világban ugyanis démon (angolul *daemon*) elnevezéssel szokás illetni azokat a programokat, amelyek folyamatosan futnak, és időnként felébredve elvégeznek bizonyos feladatokat.

Levél kicsomagolása

Mint említettem, a teljes archívumot megőrizzük a félév során, azaz a programok összes beküldött változata megvan. Erre azért is szükség van, mert pl. minden évben előfordul, hogy egy-két hallgató az utolsó levelében elfelejti mellékelni a Prolog-megoldást, mert már korábban beadta a véglegesnek tekintett változatát. Az is megesik, hogy valaki kéri, ne a legutolsó változatot vegyük figyelembe, mert abba egy módosítás során belekerült egy olyan hiba, amit csak utólag vett észre. Ilyen esetekben jól jöhet,

ha az ETS üzemeltetője könnyen ki tudja csomagolni a korábbi leveleket, és ily módon kézzel bele tud avatkozni a tesztelési folyamatba.

Archívum takarítása

Az archívum egy idő után feleslegesen nagyra duzzadhat, és hosszú távon elegendő a legutolsó verziók tárolása. E célból praktikus egy olyan szolgáltatás, amely kitakarítja az archívumot, és csak a legfrissebb változatokat tartja meg.

Értesítések kiküldése

A beküldött levél azonnali tesztjéről automatikusan kap értesítést minden hallgató, aki kérte. Az éles tesztelésről, amely pontokat ér, a jelenlegi változatban nem. Ugyanakkor nem lenne bölcs dolog az éles tesztelés során azonnal elküldeni az értesítéseket, mert előfordulhat, hogy valahol hiba csúszik a folyamatba. A legjobb megoldás az, hogy a tesztelés után, miután az oktató meggyőződött róla, hogy az eredmények rendben vannak, egyetlen paranccsal ki lehessen küldeni az értesítéseket az archivált naplóállományok alapján.

Hívási gráfok generálása

Mint említettem, Lukácsy Gergely hasonlóságvizsgáló programja nyelvfüggetlen, csak arra van szüksége, hogy egy megfelelő alprogram feltérképezze a vizsgálat tárgyát képező programok hívási gráfját. Ennek megfelelően a komponens egyik feladata az ilyen gráfok elkészítése lehet.

5.5.3. Könyvtárak, állományok

A komponensnek helyet adó könyvtárat három alkönyvtárra osztanám. Az egyikben magát a tesztkörnyezetet tárolnám, a másikban a spool állományokat, a harmadikban pedig az archívumot és azokat a könyvtárakat, amelyekben a hallgatók programjait teszteljük. Ezeken belül további könyvtárakat hoznék létre, az 5.6. ábra szerint.

Az ábrán a csúcsos zárójelekbe írt szavak a jelentésüktől függő állomány- vagy könyvtárnevet takarnak. <name> az adott hallgató nevéből és azonosítójából generált, jól olvasható, informatív de egyértelmű nevet jelent.

Szintén a megvalósításhoz választott programozási nyelven múlik, hogy a forrásállományok közvetlenül alkalmasak-e futtatásra (a nyelv *interpretált*, mint pl. a Perl, a Bash, vagy akár a Prolog vagy az SML), vagy le kell-e őket fordítani valamilyen köztes kódba (Prolog- és SML-programokkal ez is megtehető, ilyen formában gyorsabban tölthetők be). Az ilyen köztes kód leggyakrabban architektúra-függő, ezért az utóbbi esetben érdemes a bináris állományokat elkülöníteni, hogy a rendszer kényelmesen hordozható maradjon, és véletlenül se keveredjenek össze a különböző változatok. Ilyen elkülönítésre alkalmas például egy-egy, az adott architektúra nevét viselő alkönyvtár, de az is jó, ha ugyanezt a bináris állomány nevének végére egy ponttal elválasztva odaírjuk.

Nyelvi inicializációs állomány (env/<semester>/<class-ID>/<lang>/setup)

A nyelvi könyvtárakban (ml, pl) a keretprogram mellett található egy *inicializációs állomány* is, amely megadja, hogy az adott nyelv teszteléséhez

1. mely állományokat kell bemásolni (linkelni) a tesztkönyvtárba;
2. hogyan kell összeszerkeszteni a hallgató programját a keretprogrammal (az SML esetében ehhez külön meg kell hívni egy programot, a Prolog esetében ez futtatás közben történik meg);
3. hogy hívják a teszteléshez meghívandó programot (amely az előbbi lépés eredménye);
4. milyen állományokat kell kitakarítani a teszt végeztével.

Az állomány szerkezete természetesen attól függ, hogy a komponenst milyen eszközökkel valósítják meg. Lehet egyszerű shell szkript vagy program, amely elvégzi a szerkesztést, és kiírja a kimenetére a többi értéket. Prolog-megvalósítás esetében a legcélszerűbb egy alkalmas Prolog-kifejezés megadása.

```
env/                                   a tesztkörnyezet
    permanent/                         a környezet évről-évre állandó része
    <semester>/                        az adott félévre vonatkozó állományok
        <class-ID>/                    az adott tesztosztályhoz
            ml/                        az SML-részhez
                setup                  inicializációs állomány
            pl/                        a Prolog-részhez
                setup                  inicializációs állomány
            <suite-ID>                 a tesztadatok, állomány vagy könyvtár
    templates/                         sablonok automatikus levelekhez és szkriptekhez
        mail-refused                   „nincs beadási időszak”
        ill-formatted-mail             „hibás formátumú levél”
        mail-accepted                  „a levelet felvettem”
spools/                                a spool állományok helye
    <class-ID>.<suite-ID>              spool állomány
    <class-ID>.<suite-ID>.lock         spool állomány zárja
hwks/                                  a házi feladatok
    <semester>/                        az aktuális félévben
        <class-ID>/                    az adott tesztosztályhoz
            mails/                     a beküldött (és elfogadott) levelek
                <name>.<version>       az adott hallgató levele(i)
            reports/                   a naplóállományok
                <suite-ID>/            az adott feladatsorhoz
                    <name>             az adott hallgatónak
                    <name>.<lang>      ha kiemelten csak egy nyelven teszteltük
            <name>/                    a hallgató tesztkönyvtára
            <name>.lock                a tesztkönyvtár zárja
                mail                   „soft link” a megfelelő levélre
```

5.6. ábra. A házi feladatokat feldolgozó komponens könyvtárstruktúrája

Sablonok (env/templates/)

A *sablonok* a hallgatóknak szánt üzenetek formáját és az állandó szövegrészeket leíró állományok, amelyek formátuma nyilvánvalóan függ a megvalósítástól. Lehetnek szöveges állományok speciális karaktersorozatokkal a sablon mezőinek jelzésére, lehetnek egyszerű programok, amelyek argumentumként kapják meg a mezők értékét, vagy akár összefoghatjuk őket egyetlen Prolog-modulba, amelyben a sablonoknak predikátumokat feleltetünk meg.

Spool állományok (spools/)

Egy-egy *spool állomány* reprezentálja az egy tesztsorhoz tartozó, beérkezett de még feldolgozásra váró hallgatói levelek várakozási sorát. Alapvetően csak a hallgatók nevét (pontosabban nevéből és azonosítójából generált nevet) tárolja, de egy-egy névhez több *opció* is tartozhat:

- Tesztelés előtt ki kell csomagolni a legfrissebb (vagy egy megadott verziójú) levelet. Ennek hiányában a már kicsomagolt változatot teszteli a rendszer, például amikor az éles tesztelés zajlik.
- Nem az összes nyelven kell tesztelni a programot, hanem csak egy bizonyoson. Erre abban a már említett esetben lehet szükség, amikor valamilyen okból csak az egyik nyelven kell újratesztelni a programok egy részét.

A hallgatók neve mellett előfordulhat egy speciális név is (mondjuk halt), amely a teszt-démont futásának befejezésére szólítja fel. Ezzel az utasítással oldható meg, hogy a teszt-démon kilépjen, miután

tesztelte egy spool állomány összes programját, ellenkező esetben ugyanis végtelen ciklusban kell várakoznia az újabb és újabb levelekre.

Zárak

Több állomány és könyvtár zárolására szükség lehet a többszörös hozzáférés miatt. Egyfelől zárolni kell a spool állományokat, hogy a leveleket fogadó program és a tesztelő ne akarja egyszerre módosítani ugyanazt az állományt, hanem várják meg egymást. Másfelől zárolni kell az éppen tesztelt hallgatói könyvtárakat, hogy egy másik tesztelő ne írja felül az adott könyvtárat egy levél kicsomagolásával.

Valamilyen módon gondoskodni kell a beragadt zárak törléséről. Erre jó módszer lehet az, hogy a zárat létrehozó program beleírja a saját processz azonosítóját, ha pedig valamely másik program ellenőrizni akarja a zár érvényességét, nem kell mást tennie, mint megnéznie, hogy létezik-e ilyen azonosítójú processz. Kevésbé biztonságos megoldás egy zár-időlimit meghatározása, amelynek letelte után az adott zárat beragadtnak tekintjük.

5.5.4. A komponens paraméterei és működése

Ahhoz, hogy a tesztosztályokat a leírtaknak megfelelően, egységesen tudjuk kezelni, megfelelő paraméter-rendszerre van szükségünk. A paraméterek megnevezése mellett egyúttal lépésről lépésre ismertetem az egyes részek pontos működési mechanizmusát is.

Programok beadása

A beadó program valamilyen formában tárolja a névsort, a jelenlegi változat például egy a szkript végéhez csatolt, uuencode-dal „titkosított” blokkban. Futtatásakor elsőként ellenőrzi, hogy minden részfeladathoz elérhetők-e a szükséges programok (zip, uuencode, mail stb.), ha valami hiányzik hibaüzenettel leáll. Ezt követően tájékoztatja a hallgatót a beadás formai követelményeiről. Ha minden rendben van, az aktuális könyvtár tartalma alapján megpróbálja felismerni, hogy melyik tesztosztályról van szó, majd ezt a hallgatóval is megerősítteti. Ezután kilistázza az elküldendő állományokat, és ismét megerősítést kér. Innentől kezdve beavatkozás nélkül végzi a munkáját: összetömöríti az állományokat (pl. zip vagy tar és gzip programok használatával), kódolja valamilyen 7 bites kódolóval (pl. uuencode), hogy biztosan átjusson a levelezőrendszeren, kiegészíti fejléc-információkkal (a hallgató neve, a tesztosztály kódja), majd elküldi a fogadóprogram beégetett e-mail címére.

Levelek fogadása

Annak, hogy a levél melyik tesztosztályba tartozik, a törzsében megadott információkból kell kiderülnie. Az összes többi paramétert egyetlen állományban tárolhatjuk (ennek elhelyezése tetszőleges, de az elérési utat meg kell adni a program az elindításakor). Ezek a következők:

- az aktuális félév azonosítója;
- minden tesztosztályhoz:
  - egy alapértelmezett tesztsor-azonosító;
  - opcionálisan egy spool állomány neve;
  - a beadási időszak eleje és vége.

A fogadóprogram elsőször ellenőrzi, hogy a levél megfelel-e a formai követelményeknek, ha nem, akkor értesíti a feladót a helyes beadás módjáról. Ezután megvizsgálja, hogy az adott tesztosztályra vonatkozó beadási időszakon belül vagyunk-e, ha nem, megfelelő értesítést küld a hallgatónak (ehhez már az adatbázisban tárolt címét használja, nem pedig a levélben szereplőt). Ha minden rendben, az aktuális félév, a levélből kinyert tesztosztály és az ennek megfelelő tesztsor azonosítója alapján a archiválja a levelet, és a spool állományt is frissíti, majd nyugtát küld a hallgatónak.

A spool állomány neve azért opcionális, mert a tesztosztály és a tesztsor azonosítója alapján előállítható egy alapértelmezett név.

Webes beadás

Ez a szolgáltatás egyesíti a beadó- és a fogadó-oldalt. Ahogy azt a szolgáltatások ismertetésénél is leírtam, a webes beadáshoz a hallgatónak egy HTML űrlapot kell kitöltenie, megadva minden szükséges adatot (a hallgató azonosítója és jelszava, a tesztosztály azonosítója, valamint a beadni kívánt állományok neve). Az adatok megadása után a szerver-oldali CGI programnak ellenőriznie kell azok helyességét.

Ha minden rendben van, a rendszer áttérhet a fogadó-oldali részfeladatokra. Ezen a ponton két lehetőség tárul elénk. Az egyik, hogy a feltöltött állományok „hagyományos" módon, elektronikus levélen keresztül folytatják útjukat, a beadó program szerver-oldali futtatása után. A másik, talán egyszerűbb és feltétlenül hatékonyabb megoldás az, hogy a CGI program saját maga végzi el a további teendőket is, tehát ellenőrzi a beadási időszakot, majd a programot megfelelően összecsomagolva elhelyezi az archívumban és a spool állományban. Ebben az esetben a művelet nyugtázása sem levélben, hanem webfelületen keresztül történik.

Programok tesztelése

Teszt-démonból több is futhat egyszerre, mindenesetre egy teszt-démon egyetlen tesztsorért felelős. Ez azt jelenti, hogy csak azokat a programokat kell tesztelnie, amelyeket a hozzá rendelt tesztsorra kell futtatni, ezeket pedig egy spool állományban kapja meg. Többek között ezt kell a paramétereknek is tükrözniük:

- az aktuális félév azonosítója;
- a tesztosztály és a tesztsor azonosítója;
- opcionálisan egy spool állomány neve (ld. az ide vonatkozó előbbi megjegyzést);
- a tesztelendő programozási nyelvek listája;
- a tesztesetek elhelyezése:
  - könyvtárban, állományonként egy ⇒ a könyvtár neve;
  - egyetlen állományban, soronként egy ⇒ az állomány neve;
  - keretprogramba építve;
- az időlimitek listája (tesztesetenként) és/vagy egy alapértelmezett időlimit;
- a futási időt kell-e mérni;
- a megoldás ellenőrzése:
  - külön programmal ⇒ a program neve;
  - keretprogramba építve;
  - a komponens beépített ellenőrzőjével;
- az adatbázist kell-e módosítani, ha igen, melyik mezőjét.

A teszt-démon működése a következő. Elsőként megvizsgálja a spool állományt, ha ez üres, akkor adott várakozási idő után újra próbálkozik. Ha a sorban van program, akkor a megfelelő opció megléte esetén kicsomagolja a levelet, majd ezután sorra veszi az összes tesztelendő nyelvet (akár opció, akár paraméter alapján). Minden nyelvre bemásolja a megfelelő nyelvi állományokat a hallgató tesztkönyvtárába, összeszerkeszti a programot, futtatja a megadott tesztesetekre, és gyűjti az eredményeket. Ha valamelyik tesztesetre a program túllépi az időlimitet, kilövi, és az esetet hibás megoldásnak tekinti. A tesztelésről folyamatosan naplót készít, feltüntetve a futási időket és eredményeket. A nyelvi teszt befejeztével kitörli az újragenerálható állományokat, majd továbblép a következő nyelvre. A tesztelés befejeztével archiválja a naplót, és elküldi a hallgató adatbázisba bejegyzett elektronikus címére, ha ezt a hallgató kérte. Ezután kitörli a spool állományból a programot és veszi a következőt. (Eddig azért nem törölhetett, mert ha tesztelés közben a tesztelő futása megszakad áramszünet vagy hasonló hiba folytán, akkor az adott program tesztelését újra kell kezdeni.)

A többi szolgáltatás lényegesen egyszerűbb, ezért nem látom értelmét, hogy azok működését is elmagyarázzam lépésről lépésre. A levelek fogadásához és a teszteléshez megvalósított rutinok, eszközök – esetleg kis módosítással – eleve alkalmasak ezen kiegészítő szolgáltatások megvalósítására, csupán a felhasználói felületet kell kialakítani. A komponens működtetését alapvetően parancssori utasítások kiadásával képzelem, de megfelelő autentikáció esetén egy webfelület távolról vezérelhetővé is teheti.

5.6. A gyakoroltató komponens

Az erről a komponensről alkotott tervem főként az oktatók által vázolt elképzeléseken alapul, de nagy hatással volt rá a gyakorlórendszer próbaváltozata is, amelyet két demonstrátor készített Perl nyelven, s ezt követően maguk az oktatók fejlesztettek tovább. Különösen igaz ez a program kezelői felületére, amelyet alaposan tanulmányoztam, szemben a programkóddal, amelyet csak felületesen, nagy vonalakban ismerek. Talán nem túlzás azt állítanom, hogy ez a legszerencsésebb állapot, mert így tudok ötleteket meríteni, de nem ragadok le egy-egy jelenleg alkalmazott megvalósítási eszköznél és ezek korlátainál. A rengeteg hasznos ötletért köszönettel tartozom a rendszer készítőinek, Berki Lukács Tamásnak és Békés András Györgynek, valamint a tárgy oktatóinak.

A komponens struktúrájának megtervezésekor ezúttal is igyekszem függetlenedni az adott tárgytól, és csupán a mondanivaló megvilágítása érdekében hozok helyenként egy-egy konkrét példát. Mivel a modul létrehozásának az a célja, hogy a hallgatók gyakorolhassák vele tárgyi tudásukat, nagyon fontos, hogy a felülete felhasználóbarát legyen. Emellett természetesen az is fontos, hogy az oktatók át tudják tekinteni a belső struktúráját és relatíve könnyen tudják újabb feladatokkal és feladat-típusokkal bővíteni a rendszert. A továbbiakban is e két szempont szerint fogom csoportosítani a mondandómat: elsőként a felhasználói felületével kapcsolatos terveimet mutatom be, majd áttérek a belső szerkezetre, az adatok és információk szervezésére. Mielőtt azonban belevághatnék e két témába, meg kell határoznom, hogy hogyan csoportosíthatjuk a feladatokat.

5.6.1. Feladatok

A feladatok csoportosítása

A feladatokat programozási nyelvenként kétféleképpen tudjuk csoportosítani. Mindkét csoportosítás a feladatok egy-egy jellemzőjén alapul, ezek a következők lehetnek:

feladat-témakör: meghatározza, hogy az adott feladat a tananyag mely részéhez kapcsolódik, mit hivatott elsősorban gyakoroltatni. Ilyen például a Prolog vezérlési eljárások témaköre. Az egyes témakörök között persze lehetnek átfedések, az ilyen feladatok esetében önkényesen eldönthetjük, hogy melyik témakörbe soroljuk (hacsak nem akarjuk több témakörbe sorolni).

feladat-séma: a feladat *formai* tulajdonságait jellemzi, elsősorban az adott weblap megjelenését és a megoldás alakját határozza meg. Ilyen kategória például az SML típusegyenlet, amikor egy-egy kifejezés típusát kell meghatározni, vagy a programozás, amikor egy egyszerűbb rutint kell megírni.

Az ETS programozásának szempontjából alapvetően a sémák szerinti felosztás hasznos. Egy sémába azok a feladatok tartoznak, amelyekhez ugyanolyan jellegű adatokat kell kérdezni a hallgatótól, és ezen adatok ellenőrzése is ugyanúgy zajlik. Érthető módon ez független a témakörtől, hiszen például programozási feladat esetében mindig egy rutint kell megírni, amit mindig valamilyen – a feladattal együtt meghatározott – bemeneti adathalmazon tesztelünk.

Felhasználói oldalról közelítve inkább a témakörök szerinti felosztásnak van értelme, mivel így lehet jól elkülöníteni a tananyag egymástól relatíve független részeit. Márpedig mindig könnyebb egy-egy kisebb önálló rész megértése, mint az összefüggő egészé.

A két tulajdonság tehát független egymástól, elméletileg akármelyik témakörben előkerülhet akármelyik séma. El tudok képzelni például listákkal kapcsolatos programozást, és listák kanonikus alakra hozását is. Természetesen bizonyos témakörök szűkítik az alkalmazható sémák körét.

Témakörök és sémák a *Deklaratív programozás* esetében

Noha a tervezés során célom, hogy az ETS rendszer tárgyfüggetlen maradjon, a módszereket, fogalmakat időnként érdemes valós példákkal alátámasztani. Ezt teszem ebben az esetben is, amikor bemutatom, hogy a *Deklaratív programozás* című tárgy esetében milyen konkrét témaköröket és sémákat lehetne definiálni. A témakörök felsorolásakor megadom, hogy milyen sémákat rendelhetünk hozzájuk (a sémákra a tömörség kedvéért a betűjelükkel hivatkozom). Fontos megjegyezni, hogy ezek csak példák, a megvalósítás során csak egy alapos felmérés után szabad meghatározni a „végleges” kategóriákat.

1. Prolog-sémák:
   (a) **Alapstruktúra-alak:** egy kifejezés kanonikus változatát kell megadni.
   (b) **Sikerül/meghiúsul/hiba:** meg kell adni, hogy egy hívás végrehajtása milyen eredménnyel jár. Siker esetén egy beviteli sorban meg kell adni egy megnevezett változó értékét a hívás után, hibából pedig megkülönböztetjük a szintaktikai és a futási hibát.
   (c) **Értékek sorrendje:** egy hívás eredményeként az egyik megnevezett változóban előálló összes értékeket kell megadni sorrendhelyesen.
   (d) **Programozás:** egy adott specifikációt teljesítő eljárást kell megírni.
2. Prolog-témakörök:
   (a) **Struktúrák:** struktúrák kanonikus alakja (*a* séma), egyesítése (*b* séma).
   (b) **Listák:** listák kanonikus alakja (*a* séma), egyesítése (*b* séma) és kezelése könyvtári függvényekkel (*b*–*c* séma), listafeldolgozó és -építő eljárások írása (*d* séma).
   (c) **Vezérlési eljárások:** a vágó, a negálás, a call eljárás stb. használata (*b*–*c* séma).
   (d) **Meta-logika:** A functor/3, number/1 stb. eljárások használata (*b*–*d* séma).
3. SML-sémák:
   (a) **Típus meghatározása:** egy kifejezés típusát kell megadni.
   (b) **Típus és érték meghatározása:** egy kifejezés típusát és értékét kell megadni.
   (c) **Kanonikus alak:** egy kifejezés kanonikus változatát kell beírni.
   (d) **Típusdeklarációk:** specifikáció alapján absztrakt adattípust kell definiálni.
   (e) **Programozás:** egy adott specifikációt teljesítő függvényt kell megírni.

   Az *a* sémára azért van szükség a *b* séma mellett, mert pl. egy függvény értékét nem lehet másképp meghatározni, mint a definiálásával, az pedig a feladat része. Ilyenkor elegendő a típusát kérdezni.
4. SML-témakörök:
   (a) **Primitív típusok:** az SML beépített adattípusai (*b*–*c* séma).
   (b) **Egyszerű függvények:** primitív típusú adatokat manipuláló függvények (*a, b, e* séma).
   (c) **Listák:** listák típusa és kezelése könyvtári függvényekkel (*b* séma), kanonikus alak (*c* séma), feldolgozás és építés (*e* séma).
   (d) **Absztrakt adattípusok:** definiálásuk (*d* séma) és használatuk (*e* séma).
   (e) **Részlegesen alkalmazható függvények:** típusuk (*a* séma), használatuk (*b* séma) és készítésük (*e* séma).
   (f) **Magasabb rendű függvények:** típusuk (*a* séma) és használatuk (*b* séma).

A programozás séma esetében lehet bizonyos megkötésekkel élni, pl. a klózok számát vagy a felhasználható, felhasználandó beépített rutinokat illetően, meg lehet adni előre bizonyos klózokat vagy segédeljárásokat, vagy elő lehet írni, hogy a megoldás jobbrekurzív legyen. Ezek a megszorítások akár a séma paramétereinek is tekinthetők.

5.6.2. A felhasználói felület

A komponens jelenlegi megvalósításának felhasználói oldalát alapvetően kielégítőnek találtam, ezért apró változtatásoktól eltekintve nem módosítottam rajta. A felület weblapjai háromszintű hierarchiát alkotnak. A legkülső szinten, a bejelentkező lapon a hallgatói azonosítót és az adatbázisban tárolt jelszót kell megadni, így tudunk csak hozzáférni a további lapokhoz. A második szinten a hallgató nyomon tudja követni saját haladását egy táblázat segítségével, amelyben témakörönként meg van adva, hogy összesen hány feladattal próbálkozott idáig és hogy ezek közül hányat oldott meg sikeresen. Emellett

azt is ki tudja választani, hogy most melyik témakör feladatait szeretné megoldani. Ennek technikai megvalósítására több lehetőség is kínálkozik. Vagy egy külön listában ajánljuk fel a választást, vagy a fenti statisztika egyes soraiban található egy-egy keresztivatkozás is, amely elvezet az adott témakör feladataihoz. A legfontosabb persze a harmadik szint, a feladatok lapja, ennek tartalma a következő:

1. a legutóbbi akciónkat, választásunkat nyugtázó kommentáló üzenet;
2. a kiválasztott témakör neve;
3. az a számadat, hogy a témakörben hány feladatot oldottunk meg jól, ill. összesen;
4. a feladat sémája, azaz annak rövid ismertetése, hogy hogyan várja a választ a rendszer (pl. „adja meg a kifejezés kanonikus alakját...");
5. esetleg egy keresztivatkozás egy példára az adott témakör-séma párhoz;
6. a kapott feladat sorszáma, hogy kérdés, panasz esetén lehessen mire hivatkozni;
7. a feladat tömören megfogalmazva;
8. a sémafüggő űrlap, amelyben megadhatjuk a válaszunkat;
9. egy „másik példa", „másik témakör" és egy „kilépés" keresztivatkozás.

A lap tetején megjelenő üzenet elsősorban arra szolgál, hogy a legutóbb adott megoldásunk ellenőrzésének eredményét közölje. Legjobb esetben az áll benne, hogy a megoldás hibátlan, ezt követheti valamilyen magyarázat arra, hogy miért az (azok számára, akik találgatással jutottak el ide, és nem tudják, hogy lehetett volna kitalálni). Ilyenkor ugyanabból a témakörből kapunk egy következő feladatot. Az is lehet persze, hogy a megoldás rossz, ilyenkor az üzenet valamilyen módon rá kell hogy vezessen a hiba jellegére és helyére, és természetesen ugyanazt a feladatot kell visszakapnunk. A kényelem szempontjából fontos, ha az űrlap sem üres, hanem tartalmazza az előző megoldásunkat, és csak módosítani kell rajta. A lapalji hivatkozások arra az esetre kellenek, ha a hallgató nem boldogul az adott feladattal, elunja a témakört, vagy be akarja fejezni a gyakorlást.

Minden témakörhöz van egy teljesítendő keret is: egy adott számú feladat megoldása után a rendszer úgy ítéli meg, hogy az adott témakört kellően elsajátítottuk, és másik választását javasolja. Ez a keret szintén feltüntethető mind az összefoglaló táblázatban, mind a feladatlap tetején megjelenített statisztikák részeként. Természetesen egy-egy témakör gyakorlását meg kell engedni a keret teljesítése után is, ilyenkor elég figyelmeztetni a hallgatót. A témakör kiválasztásakor a döntés azzal is segíthető, ha a lapon más színnel van írva azon témakörök neve, amelyeket már teljesített az érintett hallgató.

5.6.3. Belső szerkezet

Az előző szakasz alapján kiderült, hogy egyfelől nyilván kell tartani az egyes hallgatók haladását, másfelől különbözőképpen kell kezelni az egyes nyelveket és ezen belül az egyes feladat-sémákat.

A hallgatók eredményeit természetesen az adatbázisban tároljuk, de ugyanitt feljegyezhetjük a haladásukat is. Rendeljünk minden hallgatóhoz egy olyan entitást, amelyhez témakörönként egy-egy attribútumot rendelve nyilvántartjuk a jól és a rosszul megoldott feladatok listáját, és egy további attribútumban azt, hogy melyik témakörrel foglalkozott legutóbb. Igény szerint azt is bejegyezhetjük, hogy milyen megoldásokat adott az egyes feladatokra, akár a megoldás direkt tárolásával, akár egy naplóállományra való hivatkozás formájában. Ehhez persze újabb attribútumokra van szükség, amelynek neve összeállhat a feladat egyedi azonosítójából (témakör-azonosító + sorszám) és a kísérlet sorszámából (ti. hogy hanyadszorra próbálja megoldani az adott feladatot).

A sémák és témakörök kezeléséhez egy könyvtárstruktúra kialakítását látom célszerűnek. Ennek alapján – mint látni fogjuk – a komponens tárgyfüggetlen részét képező CGI szkriptek automatikusan ki tudják nyerni a lapon megjelenítendő és az adatbázisba beírandó információkat, anélkül, hogy külön konfigurációs állományra lenne szükség. A tervet, amely a jelenleg meglévő rendszer könyvtárstruktúrájának továbbfejlesztett változata, az 5.7. ábra mutatja.

A csúcsos zárójelek közé írt nevek mind egy rövid, egyedi azonosítót jelölnek. A nyelvek esetében ennek nincs jelentősége, de a témaköröknél ez a rövid azonosító épül be például a feladatok egyedi azonosítójába is, a témakörön belüli sorszámmal együtt. A séma esetén azért fontos az azonosító, mert így kell rá hivatkozni a témakör-adatokat rögzítő állományon belül (ez hamarosan kiderül).

<lang1>/	1. nyelvi könyvtár
<topic1>	1. témakör adatai
<topic2>	2. témakör adatai
<topic3>	3. témakör adatai
<scheme1>/	1. séma adatai
check	ellenőrző program
template	séma-sablon
<scheme2>/	2. séma adatai
examples/	példakönyvtár
<lang2>/	2. nyelvi könyvtár

5.7. ábra. A gyakoroltató komponens tárgyfüggő könyvtárai

Egy témakör adatai

A témakörök adatait az 5.7. ábrának megfelelően egy-egy állomány rögzíti. Ennek szerkezete megvalósítási kérdés, de az alábbi adatokat mindenképpen tartalmaznia kell:

- egy egyedi füzért az állomány elején (pl. „topic-fi le”), hogy az adott könyvtárban lévő más állományokat a rendszer ne akarja témakör-állományként beolvasni – ez a megkülönböztetés megoldható úgy is, hogy az állomány *neve* speciális, pl. kap egy „.tpc” végződést;
- a témakör hosszú nevét, amely megjelenik a weblapokon is – ebbe a névbe érdemes belefogalmazni a nyelv nevét is, pl. „SML: absztrakt adattípusok”;
- azt a mennyiséget, ahány feladatot meg kell oldani ahhoz, hogy a rendszer másik témakör választását javasolja;
- és feladatonként
  - a feladat sémájának azonosítóját (egyeznie kell a megfelelő séma-könyvtár nevével);
  - egy rövid leírást;
  - teszteseteket vagy referencia-megoldást az ellenőrzéshez.

A jelenlegi megoldásban egy feladat adatai egyetlen sort foglalhatnak el. Ez nem szerencsés, mert az eredmény igen nehezen tekinthető át. Praktikusabb olyan formátumot választani, amiben többsoros bejegyzéseket tárolhatunk, pl. XML[1]-t.

Egy séma adatai

Egy séma adatait egy könyvtár állományaiban tároljuk. Az egyik ilyen állomány egy *HTML sablon*, amely az ilyen sémájú feladatok lapján megjelenítendő információk (az 5.6.2. szakaszban lévő felsorolás 4–8-as pontjainak megfelelő adatok) sablonja. A sémafüggő részek természetesen fixen be vannak írva ide, míg a témakör- és feladatfüggő adatok helye csak jelezve van valamilyen módon (például speciális HTML tag-ekkel, amelyeknek amúgy nincs értelmük). A sablon legfontosabb része az űrlap. Az ebben elhelyezkedő mezők neve olyan kell, hogy legyen, hogy azt a tárgyfüggetlen CGI program felismerje, hiszen a mezők tartalmát át kell adnia az ellenőrző programnak, amely a séma-könyvtár másik állománya.

Az *ellenőrző program* megkapja az űrlap mezőinek tartalmát párosítva a mezőnevekkel, és a témakör-állományban elhelyezett referencia-megoldást vagy teszteseteket. Mindkét adatcsoportot egy-egy (vagy egyetlen közös) állományban célszerű átadni, mivel méretük ahhoz esetenként túl nagy lehet, hogy parancssori argumentumban kapja meg. Természetesen ebben az esetben az állomány(ok) neve az argumentum. A program a megfelelő tesztek elvégzése után a megoldás elfogadhatóságát (jó-e vagy sem)

[1]eXtended Markup Language

a kilépési kódjával jelzi, a standard kimenetére pedig kiírja a vizsgálat eredményét ismertető HTML üzenetet, amelyet a következő feladatlap tetején jelenít meg az őt futtató CGI program. Ebben közölheti például hibás megoldás esetén a hiba jellegét és helyét. Egy ilyen hibaüzenet akár részletes, intelligens analízis eredménye is lehet, amelyből nem csak az derül ki, hogy a hibát mi okozza, hanem az is, hogy ebből milyen ismeret hiányára lehet következtetni, ill. hogy ennek hol lehet utána nézni a jegyzetben.[2]

Példamegoldások

A példa megoldásokat témakörönként külön-külön a megfelelő könyvtár HTML állományaiban vagy akár összefogva egyetlen HTML állományban is elhelyezhetjük, azzal a feltétellel, hogy minden példához elhelyezünk egy HTML horgonyt. Ennek neve témakörönkénti HTML állományok esetén a séma azonosítója lehetne, közös HTML állomány esetén pedig a témakör- és a séma- azonosítójából állna össze. Ezeket a CGI program felismerheti, és ha olyan feladat lapját jeleníti meg, amelynek témakör-séma párjához van példa, akkor a lapon elhelyez egy kereszthivatkozást a példalap megfelelő horgonyára.

Érdemes lehet a példákat nem közvetlenül HTML-ben leírni, hanem valamilyen olyan formátumban (például TEXinfo-ban), amelyből szükség esetén nyomtatható változat is előállítható, így akár a jegyzetbe is könnyen be lehet őket emelni.

5.6.4. A komponens működése

Az eddigi információk alapján már nem okozhat meglepetést a működés ismertetése. A CGI állomány először is belépteti a hallgatót a jelszó és az azonosító bekérésével. Ezután összegyűjti a témakörök nevét (a témakör-állományokból), a hallgató helyzetét, és megjeleníti a statisztikákat tartalmazó és témakör-választást felkínáló lapot.

A feladatlapok megjelenítése a legizgalmasabb feladat. Az 5.6.2. szakaszban leírt felsorolásnak megfelelően elsőként ki kell írni a legutóbbi akció nyugtáját – ez vagy az egyik ellenőrző program kimenete, vagy maga a CGI program állította elő (pl. „Sikeresen belépett a rendszerbe."). Ezután következik a kiválasztott témakör neve és az erre a témakörre vonatkozó statisztikák. Ezen a ponton véletlenszerűen kiválaszt egy feladatot azok közül, amelyekkel a hallgató még nem foglalkozott. Ha ilyen nincs, akkor azok közül, amelyeket még nem sikerült megoldania. Ha ilyen sincs, akkor véletlenszerűen választ egyet. Majd betölti a kiválasztott feladat sémájának sablonját és a megfelelő információkkal kitöltve (feladat azonosítója, hivatkozás a példára, feladat leírása) ezt is kiírja. A lapot a szükséges kereszthivatkozásokkal zárja (másik példa, másik témakör, kilépés).

Az űrlap postázásakor egy állományba kimenti a tartalmát, egy másikba (vagy ugyanennek a végére) pedig a feladat megoldókulcsát, és az állomány(ok) nevével meghívja az ellenőrző programot. Még jobb lehet egy közös állomány helyett a program standard bemenetét használni erre a célra, mivel ekkor biztosan nem maradnak hátra átmeneti állományok. A kilépési kód alapján a CGI módosítja az adatbázis megfelelő bejegyzéseit (jól/rosszul megoldott feladatok listája, megoldás helye), és megjeleníti a program kimenetén kapott üzenettel kezdve a következő feladatlapot.

5.7. A ZH- és vizsga-támogató komponens

Ahogy azt az 5.3. fejezetben elmondtam, ez a komponens is alapvetően adminisztratív jellegű. Ezért feladatai és a következő szakaszban ismertetendő adatbázis-hozzáférést biztosító komponens feladatai között vannak átfedések. Amint látni fogjuk, ott is vannak olyan szolgáltatások, amelyek a ZH- és vizsgaeredmények lekérdezését és bevitelét támogatják. Ezekre nem is térek ki ebben a szakaszban. Minthogy azonban ezek mind a Weben keresztül érhetőek el, ismertetek néhány olyan hasonló szolgáltatást, amelyek parancssor-alapúak, bizonyos helyzetekben ugyanis hasznos lehet a szöveges hozzáférés is.

Az ígéretemnek megfelelően vázolom elképzeléseimet az automatikus ZH-sor generálással és javítással kapcsolatban is. Hogy ezekből mi valósul meg, az nagyban múlik a gyakorlati alkalmazhatóságukon, amelyről szintén ejtek pár szót.

[2] A komponens ezen szolgáltatása bizonyos mértékben emlékeztet az ITS-ek intelligens hallgató-modelljére.

5.7.1. Adminisztratív teendők

Az adminisztratív teendők ismertetésénél időrendi sorrendben haladok, elsőként a ZH, majd a vizsga támogatásáról ejtek szót.

ZH eredmények feltöltése: noha a következő szakaszban ismertetendő komponens erre Weben keresztül lehetőséget nyújt, nem szerencsés, ha ez az egyetlen lehetőség. Ennek két oka is van: egyfelől sok esetben sokkal kényelmesebb lehet egy szöveges állomány kitöltése egy űrlapénál, másfelől a javítók többsége – ahogy én is – otthon dolgozik, így az eredmények is otthon állnak elő, és mivel egy nagy méretű űrlap kitöltése időigényes, a modemes kapcsolat fenntartása erre az időre nem olcsó mulatság. Sokkal jobb tehát, ha lehetőség van *offline* feltöltésre is.

A jelenlegi félévben is valami hasonló rendszer működik: a javítók egy Prolog-kifejezésekből álló szöveges állományt küldtek az oktatóknak, akik az eredményeket így egy egyszerű programmal tudták összegezni. A módszer tovább fejleszthető az összegzés automatizálásával. A javaslatom egy elektronikus cím létrehozása, ahová a javítók beküldhetik ezeket az állományokat, ott egy program fogadja őket, ellenőrzi és az eredményekkel automatikusan frissíti az adatbázist. A manipulációk kivédése érdekében korlátozni lehet azon e-mail címek listáját, ahonnan a program elfogadja az adatokat. A fogadóprogram azt is megteheti, hogyha egy-egy hallgató minden adata[3] beérkezett, akkor küld neki egy összesítést, amennyiben ezt az érintett kérte.

A speciális állomány előállításának megkönnyítéséhez érdemes írni egy offline űrlap-programot. Kísérleti jelleggel elkészítettem egy ilyen programot, amelyet az ingyenes *GNU Emacs* szövegszerkesztőben lehet használni. A programot a B. függelékben mutatom be.

Kísérőlap nyomtatása: ez a szolgáltatás már most is működik, változtatni annyit kell, hogy az adatait az adatbázisból vegye. A lényege, hogy a szóbeli vizsgáztatáshoz olyan lapot készít minden hallgatónak, amelyen fel vannak tüntetve a félév közben megszerzett pontszámok, és vannak rubrikák a vizsgapontszámok beírásához.

Vizsgaeredmények feltöltése: nagyjából itt is ugyanazt lehetne elmondani, mint a ZH eredmények feltöltésénél, tehát itt is szükséges az offline lehetőség. Mindenképpen igaz ez írásbeli vizsgáztatás esetén, amikor a hallgatókat külön értesíteni kell az eredményekről, de szóbeli vizsgáknál is jól jöhet pusztán az adminisztrálás megkönnyítéséhez. Az oktatóknál rendszerint van a vizsgán hordozható számítógép, amelyen a kísérőlap alapján ki lehet tölteni az űrlapot.

5.7.2. Tárgyfüggő szolgáltatások

Ide sorolhatunk minden olyan szolgáltatást, amely a ZH- és vizsgasorok előállítását, a dolgozatok *javításának* megkönnyítését célozza. A jelenlegi ZH rendszerben az oktatók 2–4 csoport kérdéseit dolgozzák ki, a ZH-sor előállításában egy TEX sablonállomány segít. A csoportok számát lehet növelni, de a javításhoz szükséges idő, attól tartok, nem csökkenthető. Ha ugyanis számítógép javítja a ZH-sorokat, akkor be kell gépelni a hallgatók megoldását, és ez feltehetően időigényesebb, mint a kijavításuk.

Sokcsoportos ZH

Érdekes lehet egy olyan konstrukció, amikor nem két csoport van, hanem sokkal több, akár minden hallgató más-más feladatsort kaphat. Ehhez alapvetően egy tekintélyes méretű feladat-adatbázisra van szükség, amely minden feladattípusból tartalmaz kellően nagy számú példát. Ebből egy alkalmas program generálhatna – a TEX sablon felhasználásával – ZH-sorokat, feltüntetve rajtuk többek között a hallgató nevét és azonosítóját. A módszer előnye, hogy praktikusan lehetetlenné tenné a megoldások másolását. Hátránya abban rejlik, hogy feltehetően lelassul a javítás. Ha ugyanis egyazon feladatsort kell javítani valakinek, akkor előbb-utóbb automatikusan rááll a szeme a típushibákra, és nem kell minden alkalommal újra meglátnia az egyes megoldások mögött meghúzódó gondolatokat. Ha azonban minden dolgozat más feladatok alapján íródott, akkor minden dolgozatot külön kell megérteni is. Ebben

[3] A javítók többnyire csak egy nyelven, azaz egy fél dolgozatot javítanak.

segíthet egy olyan program, amelynek megadva a hallgató nevét vagy azonosítóját, előállítja a neki generált dolgozat megoldásait, és ismerteti a pontozási szempontokat és azokat a típushibákat, amelyek az egyes feladatoknál előjöhetnek. Ehhez persze az kell, hogy a feladat-adatbázis ezekre az információkra is kiterjedjen, a feltöltése tehát nem kis munka.

Jobban automatizálható a javítás, ha a ZH – vagy egy része – feleletválasztós. Ilyenkor a megoldókulcs alapján nagyon gyorsan lehet javítani, hiszen nem kell megérteni a hallgató gondolatmenetét. Az is igaz, hogy egy ilyen teszt sokkal felületesebb képet ad a hallgatók tudásáról, mint egy kifejtendő kérdéses feladatsor.

Automatikus pontozás

A jelenlegi pontozási szisztéma a következő: minden megoldás alapesetben megkapja a maximális pontszámot, és minden javítás, amely ahhoz szükséges, hogy hibátlanná tegye, pontlevonással jár. Elvben ez a módszer elég egzakt ahhoz, hogy automatizálni lehessen. Persze ehhez pontosan definiálni kellene az „egy javítás” fogalmát, és találni kell egy módszert, amellyel megtalálhatóak ezek a javítások. Lukácsy Gergely a hasonlóságvizsgáló programjának ismertetésekor (Lukácsy 2000) részletesen foglalkozik olyan algoritmusokkal, amelyek forrásprogramok *távolságának* megállapítására szolgálnak. Elképzelhetőnek tartok egy olyan pontozó algoritmust, amely ezekhez az algoritmusokhoz hasonlóan méri a hallgató megoldása és a referencia-megoldás távolságát, és ez alapján von le pontokat.

Gondot jelent természetesen az a nyilvánvaló tény, hogy általában egy feladatnak nem csak egyetlen jó megoldása létezik. Ez részben feloldható úgy, hogy nem egy, hanem több referencia-megoldás létezik, és a legkisebb távolságot veszi figyelembe a program, de még így is kicsi az esélye, hogy minden megoldást elfogadhatóan pontozzon. Sokkal jobb a helyzet azokban az esetekben, amikor valóban egy jó megoldás létezik, mint például az SML-típusegyenletek vagy a Prolog-egyesítések esetében.

Ha feltesszük, hogy létezik egy ilyen automatikus pontozó, akkor is elgondolkodtató, hogy mely feladatoknál érdemes használni. Egy típusegyenletet például – a helyes megoldás ismeretében – gyorsan ki lehet javítani, feltehetően gyorsabban, mint amennyi ideig a begépelése és a javítás visszamásolása tartana. A programozási feladatok esetében már több értelme lenne, ott viszont az előző bekezdésben vázolt problémával szembesülünk. Összefoglalva elmondható tehát, hogy egy ilyen pontozóprogram elvileg ugyan kellemes megoldás lehet, a gyakorlatban azonban nem elég praktikus ahhoz, hogy érdemes legyen vele behatóbban foglalkozni, főleg ha tekintetbe vesszük, hogy megvalósítása egyáltalán nem rutinszerű feladat.

5.8. Adatbázis-hozzáférést biztosító komponens

Az összes komponens közül talán ez igényli a legkevesebb tervezést. Voltaképpen nem kell mást tudnia, mint különböző lekérdezéseket megfogalmaznia és az eredményeket jól olvasható formában prezentálnia, célszerűen egy HTML lapon. A lekérdezések egy része csak olvas az adatbázisból, mások módosítják is. Mivel a komponens a Weben keresztül tartja a kapcsolatot a felhasználókkal, érdemes CGI programként megvalósítani. Az alkalmazott programozási nyelv megint csak közömbös, ha a specifikációt betartja, de ha az adatbázis a javaslatomnak megfelelően a Prologhoz illesztett Berkeley-rendszerben valósul meg, akkor érdemes ezt a programot is Prologban írni. A SICStus rendszer ráadásul támogatja Prolog CGI programok írását a Spanyolországban kifejlesztett ún. *Pillow* moduljával. A fejezet hátralévő részében a legalapvetőbb lekérdezéseket definiálom. A fejezethez tartozó A. függelékben bemutatok néhány weblap-tervezetet is.

5.8.1. Lekérdezések

Ebben a szakaszban arról lesz szó, hogy milyen lekérdezéseket kell minimálisan tudnia a komponens egy megvalósításának. Az áttekinthetőség kedvéért osszuk ezeket három csoportra. Az elsőbe azokat sorolom, amelyekkel a hallgatók fordulhatnak az adatbázishoz, a másodikba a csak az oktatók által elérhető lekérdezések tartoznak, a harmadik csoport pedig a levelezési lista archívumával kapcsolatos.

Hallgatók lekérdezései

Alapvetően minden hallgató csak a saját adataihoz férhet hozzá. A hitelesítés az azonosítójuk és egy, az adatbázisban tárolt jelszó bekérésével történhet. A jelszó alapértelmezésben jobb híján üres. Szerencsére a BME esetében elmondható, hogy a hallgatók nem nagyon ismerik egymás NEPTUN azonosítóit, tehát kicsi az esélye az illetéktelen hozzáférésnek a jelszó módosítása előtt.

Elképzelésem szerint egy központi lapon keresztül kellene belépniük a hallgatóknak a rendszerbe azonosítójuk és jelszavuk megadásával, és ezután akárhány lekérdezést, módosítást eszközölhetnének újabb azonosítás nélkül. Ennek megvalósításához a CGI programozásban jól bevált eszközök állnak rendelkezésre.

Beállítások lekérdezése: minden hallgatónak hozzá kell tudnia férni az egyéni beállításaihoz, amelyek a jelszavát, az elektronikus címét, a jegyzetrendeléseit, a levlistára iratkozását és a különböző értesítések igénylését tartalmazzák. Az adatokat tudnia kell módosítani is. Érdemes a belépés után elsőként ezt a lapot megjeleníteni, és ide helyezni el kereszthivatkozásokat a további lekérdezésekre.

Eredmények lekérdezése: fontos, hogy mindenki meg tudja nézni a saját eredményeit, az eddigiekben összegyűjtött pontszámait. Az adatokat úgy kell tálalni, hogy abból könnyen megállapítható legyen, kinek milyen vizsgajegyre van kilátása, ill. mit kell még teljesítenie.

Naplóállományok lekérdezése: előnyös, ha a hallgatók az egyes házi feladatok legfrissebb tesztjelési naplóit is meg tudják nézni a Weben keresztül. Erre például akkor lehet szükség, ha valaki nem tud könnyen hozzáférni a leveleihez onnan, ahonnan éppen beküldte a programját; ez könnyen előfordulhat webes beadás esetén.

Oktatók lekérdezései és módosítási lehetőségei

Az oktatóknak hozzá kell tudniuk férni mindenhez, amihez a hallgatók hozzáférnek. Ehhez lehet definiálni például egy vagy több operátori jelszót, amellyel bármilyen azonosító esetében beenged, vagy itt is alkalmazható az a megoldás, hogy elsőként azonosítjuk az oktatót a jelszavával, majd ezután megengedjük neki, hogy az előbb vázolt lekérdezéseket tetszőleges hallgatói azonosítóval elvégezze. E mellett persze számos egyéb lekérdezésre szükség van az adatok karbantartásához. Minden esetben fontos az oktatók hozzáférésének naplózása, hogy az esetleges illetéktelen hozzáférések legalább utólag felderíthetők legyenek.

Egy számonkérési forma eredményeinek lekérdezése: ZH javításakor például az a legkényelmesebb, ha a javító kap a rendszertől egy táblázatot, amelyben minden azonosító fel van sorolva, mellettük pedig egy űrlapban adott megfelelő számú mező a pontszámok beírásához. Így csak ki kell tölteni a táblázatot, majd a megfelelő gombra klikkelve eltárolni az egészet. Fontos persze, hogy csak a módosított mezőket vegye figyelembe a program, hiszen nem minden hallgató eredményeit tudja egy javító. A módosítások nyugtázásához vissza lehetne adni csak a módosítottak táblázatát, hogy a javító ellenőrizni és szükség esetén javítani tudja. Lehet, hogy az összes hallgató adatainak felsorolásánál szerencsésebb megoldás az, hogy a táblázat megjelenítése előtt egy listából ki kell választani a listázni kívánt azonosítókat vagy neveket.

Ugyanez a lekérdezés persze nem csak a ZH eredményeinek beírásához használható, hanem bármely más számonkérési forma esetében is. Az is előnyös lehet, ha lehetőség van arra, hogy ne egy módosítható űrlapot kapjunk, hanem egy szöveges – és akár nyomtatható – táblázatot. Ilyenkor természetesen a számított összpontszámot is javasolt feltüntetni. Azon számonkérések esetében, amelyek eredményébe más részeredmények is beszámítanak – ez kiderül az adatbázis relációból –, fel kell tüntetni ezeket az áthozott eredményeket is, és esetleg egy-egy kereszthivatkozással lehetővé tenni, hogy az oktató részletes bontásban is megtekintse őket.

Egy hallgató eredményeinek lekérdezése: az egyes hallgatók eredményeit nem csak úgy kell tudni lekérdezni, ahogy maga a hallgató is, azaz csak olvasható formában, hanem az előbbihez hasonlóan, módosítható űrlap formájában is. Ez a megjelenítési mód hasznos lehet például szóbeli vizsgáztatáskor, amikor a hallgató és az oktató együtt tekinti át az eredményeket, és számítja ki a vizsgajegyet a frissen megszerzett vizsgapontok megadásával.

Jelentkezések lekérdezése: fontos, hogy a különböző jelentkezéseket – amelyeket egy-egy regisztráció entitás rögzít – lista formájában is le tudjuk kérdezni. A listában mindenképpen szerepelnie kell a jelentkezett hallgatók nevének és azonosítójának, és opcionálisan – az oktató kérésére – szerepelhet a jelentkezés dátuma is.

Hasznos, lehet az olyan összesítés, amelyben az egyes jelentkezések vannak felsorolva, és mindegyikhez meg van adva, hogy hányan jelentkeztek rá összesen.

Statisztikák: érdekes lehet különböző statisztikák készítése. A gnuplot program segítségével akár diagramokkal tarkított dinamikus weblapokat is lehet készíteni. Ilyen statisztika lehet például egy adott számonkérés eredményeinek eloszlása (a minimálisan elfogadható szint bejelölésével), vagy a házi feladatok futási idejének eloszlása, átlagértéke az egyes tesztesetekre. A legáltalánosabb az a megoldás lenne, amely megengedi, hogy egy egyszerű, SQL-szerű nyelv segítségével magunk fogalmazzunk meg statisztikai lekérdezéseket.

Levelezési listával kapcsolatos lekérdezések

Az az információ, amit a levelekről a listában tárolunk, a levelek sokféle csoportosítását teszi lehetővé. A lekérdezések mind egy-egy csoportosítást tesznek láthatóvá a felhasználó számára.

Amikor egy levél érkezik a lista címére, egy szűrőprogramnak archiválnia kell a levelet és minden érintettnek tovább kell küldenie. Emellett ki kell olvasnia a levélből a feladót és esetleg a tárgyát, és ezeket az adatokat az elhelyezésével kapcsolatos információkkal együtt be kell írnia az adatbázisba. A levél fejlécében elhelyezett referenciák és a tárgy alapján felderíthető az is, hogy melyik szálhoz[4] tartozik, ezt relációval lehet jelezni.

A lehetséges listázásokban mindig megjelenik a dátum, a feladó és a tárgy, és egy kereszthivatkozás a levél törzsére.

Időrendi listázás: a legegyszerűbb lekérdezés a levelek érkezési ideje szerint rendezett listázás. Feltehetően érdemes a listázott leveleket idő szerint csoportokba osztani, és egyszerre csak egy csoportot megjeleníteni, pl. egy hónapot egyszerre.

Szálak szerinti rendezés: ebben a listázásban az azonos témával foglalkozó levelek egymáshoz közel, a szál szerkezetének megfelelően indentálva jelennek meg. Elképzelhető, hogy elsőként csak a szálak gyökerénél lévő – „vitaindító" – levelek láthatók, mellettük pedig egy kereszthivatkozás a szál részletes kifejtésére, ahol viszont a többi szál nem jelenik meg.

Feladó szerinti rendezés: hasznos, ha le lehet kérdezni, hogy ki milyen leveleket postázott. Előnyös, ha ugyanezt egy feladó-csoportra is meg tudjuk tenni, kiváltképp az oktatók csoportjára. Így ugyanis könnyen kaphatunk egy olyan listát, amelyben a legfontosabb információkat hordozó levelek vannak felsorolva.

További lekérdezések

Természetesen a fenti lekérdezések bemutatásával csupán lehetőségeket kívántam felvillantani. Ezeken túl számos más, hasznos lekérdezés is megvalósítható, csak a fantázia és az idő szab határt.

[4] Egy szálba (angolul *thread*) azok a levelek tartoznak, amelyek ugyanarra a levélre adott válaszok, válaszok válaszai stb. Ami összeköti őket, az a szál gyökere.

6. fejezet

Az ETS rendszer megvalósított komponensei

Az előző fejezetben az olvasó megismerkedhetett az ETS rendszer tervével. Annak igazolásaként, hogy a tervek alapján könnyen kivitelezhetőek az egyes komponensek, mintaként magam is megvalósítottam a házi feladatokat feldolgozó komponens egy-két részletét. Elkészült a tesztelő jelentős része, valamint az SML programok hasonlóságvizsgálatához szükséges hívási gráfot építő program. E két programot mutatom be a fejezet két szakaszában.

6.1. Házi feladatok feldolgozása

Az 5.5. szakaszban bemutattam az ETS azon komponensének tervét, amely a házi feladatok feldolgozásáért felelős. Hogy a terv életképességét igazolhassam, úgy döntöttem, hogy elkészítem a komponens egy kísérleti megvalósítását, amelyet remélhetőleg a gyakorlatban is lehet majd alkalmazni. Azért is ezt a komponenst választottam, mert a nagy házi feladatok értékeléséért felelős szkript-csomag jelenleg használt változatának létrehozásában is döntő szerepem volt, üzemeltetésének egyik felelőse jelenleg is én vagyok.

A programot – az 5. fejezet bevezetőjében elmondottaknak megfelelően – Prologban valósítottam meg. Az elkészült részek sajnos nem fedik le a komponens összes megtervezett szolgáltatását, de ami megvan, az elegendő ahhoz, hogy bemutassam a megvalósítással kapcsolatos elképzeléseimet.

Természetesen a legfontosabb szolgáltatás, a *programtesztelés* megvalósításával kezdtem. Erre két okom is volt. Az egyik, hogy ennek megléte nélkül nem nagyon beszélhetünk HF-eket feldolgozó komponensről. A másik, hogy ez a szolgáltatás változott a tervekben a jelenleg használthoz képest a legtöbbet, tehát ennek az elkészítése jelentette a legnagyobb kihívást. A programozás során megírtam számos olyan általános célú eljárást, amelyet később a többi szolgáltatás megvalósításakor is felhasználhattam, ill. felhasználhatok majd. Így például alig jelentett többletmunkát egyes levelek kicsomagolásának vagy az archívum kitakarításának mint szolgáltatásnak az elkészítése. Most lássuk, mi az, ami nincs meg:

- *beadó program* – de ez igényli talán a legkevesebb változtatást a jelenlegihez képest;
- *levelek fogadása* – ezt sajnos leginkább a szükséges idő hiányával tudom indokolni;
- *kapcsolat a hallgatókkal* – a levelezés azért nem működik, mert ehhez a tervek szerint szükség lenne az adatbázisra és annak bizonyos mezőire, ez azonban egyelőre szintén hiányzik.

A hívási gráfok generálása viszont immár nem csak Prolog-, hanem SML-programokra is működik, az ezzel kapcsolatos megfontolások és a megvalósítás rövid ismertetése a 6.2. fejezetben olvasható.

A továbbiakban igyekszem bemutatni az elkészült részeket, ezek működését és vezérelhetőségét. A programot *GUTS*-nak, azaz *Grand Unified Testing System*-nek neveztem el. Ennek az az oka, hogy az

új program – a korábbival ellentétben, a terveknek megfelelően – egyformán képes tesztelni a jelenlegi nagy és kis házi feladatokat, sőt olyan feladatfajtákat is, amelyek egyelőre nem is léteznek.[1]

6.1.1. Könyvtárak, állományok

A megvalósításkor alapvetően tartottam magam a megtervezett és az 5.6. ábrán (33. oldal) bemutatott könyvtárstruktúrához. Az egyetlen eltérés a sablonok elhelyezésében mutatkozik, ennek okára hamarosan fény derül. A hallgató nevéből és azonosítójából képzett név, amit ott <name>-ként jelöltem, a megvalósításban úgy áll elő, hogy a hallgató nevének szóközök és ékezetek nélküli változatához egy ponttal elválasztva hozzáfűzzük a hat karakteres, az adatbázisban tárolttal egyező, nagy- vagy kisbetűsített NEPTUN kódját, a sajátom pl. HanakDavid.e6070b lenne.

Sablonok

Ahogy a tervezéskor is írtam, Prolog-megvalósítás esetén logikusabb szöveges állományok helyett Prolog-modulként megadni a komponens által üzenetként felhasznált szövegeket. Ennek az is az előnye, hogy nagyon könnyű paraméterezni, azaz változó tartalmú mezőkkel megtűzdelni az egyes blokkokat. Az egyes tesztesetek fejlécét kiíró Prolog-predikátum például a következő lehet:

```
print_testcase(N, Limit) :-
        format('~t~d~2+. teszteset, időlimit = ~|~t~d~3+ sec~n',
               [N, Limit]),
        print('--------------------------------'), nl.
```

amely kiírja az adott teszteset számát és a hozzá rendelt időlimitet is.

Ez és a többi sablon a templates.pl állományban található, ez pedig az env/permanent könyvtárban helyezkedik el, a komponens törzsét képező guts.pl állománnyal együtt.

Nyelvi könyvtárak

A félévenként változó részek legfontosabb része az egyes nyelvek tesztelésére szolgáló keretprogramok és inicializációs állományok csoportja. Ezek a *Deklaratív programozás* tárgy esetében a megfelelő SML- és Prolog-állományok, ezek architektúra-függő változatai, valamint az inicializálásért felelős setup nevű állomány, amely szintén Prolog-kifejezéseket tartalmaz, mivel ezt a tesztelő könnyen be tudja olvasni. A példa kedvéért a 2001-es tavaszi félév NHF-jeinek ellenőrzéséhez használható SML-állományok elhelyezését mutatja a 6.1. ábra.

A keretprogram. Az ellenőrzéshez használt keretprogramról már sok szó esett az 5.5. szakaszban. Feladata a tesztesetek bemeneti adatainak beolvasása, a hallgató programjának meghívása, és a megoldás kiírása vagy értékelése, attól függően, hogy a tesztelőt milyen paraméterekkel futtatjuk. Tulajdonképpen azt mondhatjuk, hogy a keretprogramban a tesztosztálytól, a nyelvtől és a tesztsortól egyaránt függő részek vannak. Egyetlen kikötés van: minden tesztsor minden tesztesetére ugyanennek a keretprogramnak kell működnie. A tesztesetfüggő viselkedését a parancssori argumentumai szabályozzák, amelyből kettő van. Az első a tesztelendő program bemenő adatainak megadására szolgál, a második pedig egy állomány neve, amelybe az előállított megoldást kell valamilyen formában kiírnia a keretprogramnak. Hogy a két argumentum konkrétan mit jelent, az attól függ, hogy mi van a tesztelő paraméter-állományában – ennek bemutatására később kerül sor.

A nyelvi inicializációs állomány (setup). Ebben az állományban azok a beállítások vannak, amelyek függnek a tesztosztálytól és a nyelvtől, de *függetlenek a tesztsortól*. Tartalmára egy példa a 6.2. ábrán látható, amelyet a következőkben részletesen bemutatok.

Az állomány – mint többször említettem – Prolog-kifejezésekből áll, ezek sorrendje tetszőleges. A tartalmuk a következő:

---

[1] Ezen túlmenően a mozaikszónak jelentése is van (bátorság), és noha ennek nem sok köze van a programhoz, talán így könnyebben megjegyezhető.

env/01s/bhw/ml/	a nyelvi könyvtár
ESatrak.sml	a keretprogram forrása
Satrak.sig	a beadandó program szignatúrája
TSatrak.sml	a közösen használt típusok forrása
Makefi le	a futtatható állomány összeszerkesztéséhez
setup	nyelvi inicializációs állomány
Linux/	Linux-specifikus bináris állományok
ESatrak.ui	a keretprogram lefordított szignatúrája
ESatrak.uo	a keretprogram tárgykód-állománya
Satrak.ui	a beadandó program lefordított szignatúrája
Tsatrak.ui	a típusok lefordított szignatúrája
Tsatrak.uo	a típusok tárgykód-állománya
SunOS/	Solaris-specifikus bináris állományok

6.1. ábra. SML-segédállományok a 2001-es tavaszi félév NHF-jéhez

```
delete(['*.u?', 'ESatrak']).

link(['ESatrak.uo', 'ESatrak.ui', 'TSatrak.uo', 'TSatrak.ui',
      'Satrak.ui', noarch('Makefile')]).

init :-
        print('A házi feladat fordítása és szerkesztése \
        folyamatban...'), nl,
        (   run([make, depend]),
            run([make, 'TARGET=Satrak'])
        ->  print('A fordítás sikeres volt.'), nl, nl
        ;   print('A fordítás nem sikerült!'), nl, nl, fail
        ).

program('./ESatrak').
```

6.2. ábra. Egy lehetséges SML inicializációs állomány

`delete(Files).`

`Files` egy olyan lista, amely a hallgató könyvtárából a tesztelés *előtt* és *után* törlendő állományok nevét, ill. az ezekre illeszkedő mintákat sorolja fel. Az értékelő program e lista alapján takarít ki a nyelvi teszt megkezdése előtt.

`link(Files).`

`Files` szintén egy lista, minden ebben szereplő állományt az adott rendszer-architektúrához tartozó alkönyvtárból „átlinkel” (`ln -s` paranccsal) a tesztkönyvtárba. A `noarch/1` funktorba csomagolt állományokat nem az architektúra-könyvtárból veszi, hanem közvetlenül a nyelvi könyvtárból. Ilyen a példa esetében a `Makefile`.

`program(Cmd).`

`Cmd` egy atom, amely a tesztesetekre futtatandó program neve.

`init :- Cmds.`

A negyedik kifejezés opcionális, és formája is eltér az eddigiektől. Alakra olyan, mint egy predikátum, akár több klózból is állhat. Argumentuma nincs, a törzse pedig közvetlenül a tesztek megkezdése előtt, a linkelés elvégzése után fut le. Ha meghiúsul, azzal azt jelzi, hogy valami nem

sikerült, ilyenkor a tesztek elmaradnak. Itt bármilyen beépített Prolog-parancsot használhatunk, és még néhány igen hasznos eljárást, amelyeket az értékelő program definiál. Ilyen a példában is látható `run/1` eljárás, amely úgy futtatja az argumentumában megadott parancsot, hogy annak standard kimenete a naplóállományba kerüljön.

A tesztelő paraméterei

A 35. oldalon egy felsorolásban áttekintettem, hogy melyek a tesztelő paraméterei. Ennek ismeretében itt az egyetlen újdonság paramétereket meghatározó állomány szerkezete, amely – nem meglepő módon – Prolog-kifejezések sorozata. Mivel ezt az állományt maga a Prolog-interpreter értelmezi, egyszerű tényállítások helyett eljárások is szerepelhetnek. Elhelyezésére vonatkozóan nincs megkötés, ugyanis – amint az a későbbiekből kiderül (ld. 51.oldal, 6.5. ábra) – az elérési útja parancssori argumentuma a GUTS programnak. A 6.3. ábrán a példákhoz már eddig is felhasznált 2001-es tavaszi félév NHF-tesztelésének paraméterei láthatók.

```
semester('01s').
testID(bhw,test).
languages([pl,ml]).
timelimits(60,[10,10,20,20,30,30,40,40,50,50]).
timer(yes).
testsuite_placement(directory).
evaluator(builtin).
database(ignore).
```

6.3. ábra. A tesztelő egy lehetséges paraméter-állománya

Az ábra természetesen szorul némi magyarázatra. Az itt szereplő beállítások határozzák meg a tesztosztályt és a tesztsort, valamint az függô, ám a *nyelvtől független* paramétereket (vö. nyelvi inicializációs állomány). Az alábbiakban áttekintem, hogy egy ilyen állományban milyen kifejezéseknek kell lenniük (a sorrendjük tetszőleges).

`semester(SemID).`
: `SemID` az aktuális félévet azonosító atom, a példa esetében `'01s'`.

`testID(ClassID,SuiteID).`
: Azonosítja a tesztosztályt és a tesztsort, amelyet a tesztelőnek vizsgálnia kell.

`languages(Langs).`
: `Langs` a tesztelendő nyelveket azonosító atomokból álló lista. Az atomoknak meg kell egyezniük a megfelelő nyelvi könyvtárak nevével.

`timelimits(Default,PerCase).`
: Az időlimitek megadására szolgál. `Default` az alapértelmezett időlimit, `PerCase` pedig egy lista, amelynek az i-edik eleme megadja az i-edik teszteset időlimitjét.

`timer(YesNo).`
: A tesztelendő program futási idejének mérését szabályozza ez a paraméter. `YesNo` egyike a `yes` vagy a `no` atomoknak. Előbbi esetben a tesztelő méri a futási időket, amelyek a megfelelő helyen megjelennek a naplóállományokban.

`testsuite_placement(Placement).`
: Megadja, hogy a tesztesetek hol és hogyan helyezkednek el. `Placement` háromféle lehet:

  `directory`
  : A tesztesetek külön könyvtárban vannak, állományonként egy. A könyvtár relatív elérési útja `env/<SemID>/<ClassID>/<SuiteID>`, az állományok neve `test<Index>d.txt`,

az esetleges referencia-megoldásoké pedig `test<Index>s.txt`, ahol `Index` a teszteset száma. Ilyenkor a tesztelő minden teszthez megkeresi a megfelelő állományt, és ennek nevét első argumentumként átadja a keretprogramnak.

file
A tesztesetek egyetlen állományban vannak, soronként egy. Az üres sorokat, valamint a `%`, `(*`, `#`, `//` kommentjelek valamelyikével kezdődő sorokat a program figyelmen kívül hagyja. Az állomány relatív elérési útja `env/<SemID>/<ClassID>/<SuiteID>`. Ilyenkor a tesztelő minden teszt előtt kimásolja a megfelelő sort egy külön átmeneti állományba, és ennek nevét első argumentumként átadja a keretprogramnak.

embedded(Count)
A tesztesetek be vannak építve a nyelvi keretprogramokba, számuk `Count`. Ilyenkor keretprogram első argumentuma egy `<SuiteID>-<Index>` alakú füzér lesz, ahol `Index` az éppen tesztelendő teszteset száma.

evaluator(Eval).
Megadja, hogy a megoldásokat hogyan kell kiértékelni. `Eval` háromféle értéket vehet fel:

separate(EvalProg)
Az ellenőrzést külön programmal kell végezni. Ennek az aktuális nyelvi könyvtárhoz képest relatív elérési útja `EvalProg.<Arch>`, ahol `Arch` a rendszer-architektúrát jellemző név, vagy simán `EvalProg`, ha `noarch/1` funktorba van csomagolva. Ilyenkor a keretprogramnak a megoldást úgy kell kiírnia, hogy azt az itt megadott program be tudja olvasni. Az ellenőrző programot két argumentummal hívja meg a tesztelő: az első argumentum a referencia-állomány neve, a második pedig a keretprogram által kiírt megoldást tartalmazó állomány neve. Az ellenőrző programnak a kilépési kódjával (exitcode) kell jeleznie, hogy a két megoldás azonos-e. A kilépési kód 0, ha azonos, ettől eltérő, ha nem.

builtin
Az ellenőrzést a tesztelőbe épített eljárás végzi el. Ehhez mind a referencia-megoldásnak, mind a generált megoldásnak egy-egy Prolog-kifejezésnek, azon belül is egy listának, az összes megoldás listájának kell lennie. A referencia-megoldástól azt is elvárjuk, hogy rendezve legyen. Az összehasonlítás nem áll másból, mint a generált megoldás rendezéséből, és a rendezett lista és a referencia-megoldás azonosságának vizsgálatából.

Ebben a két esetben a keretprogram második argumentuma azon állomány neve, amelybe a megoldást kell valamilyen formában kiírni.

embedded
Az ellenőrzést a keretprogramba épített ellenőrző végzi, feltehetően valamiféle algoritmikus (tehát nem összehasonlításon alapuló) módszerrel; a második argumentumában kapott állományba ilyenkor az ellenőrzés eredményét kell írnia. Ha a megoldás helyes, egy 1-es karaktert kell kiírni, ha helytelen, egy 0-ást.

Ha a tesztesetek elhelyezését leíró kifejezés `directory`, akkor akármelyik értékelőt használhatjuk. Ha azonban `file` vagy `embedded(...)` alakú, akkor csak az `embedded` értékelőt vehetjük igénybe, mivel az első kettőnek szüksége van referencia-megoldásokra.

database(DB).
Az adatbázis frissítésére szolgáló információ. `DB` kétféle lehet:

ignore
Az adatbázist nem kell módosítani.

update(ID)
Módosítani kell az `ID` azonosítójú eredmény entitáshoz rendelt `<Lang>` azonosítójú pontszámot, ahol `Lang` az aktuális nyelvet leíró atom. Az adatbázis pSzTípus entitása egy példányának és a megfelelő hozzárendelésnek már léteznie kell. A pontszámmal együtt az eredmény dátuma is módosul.

Ha a `timer` paraméter aktív (argumentuma `yes`), akkor a mért futási időket egy füzér formájában eltárolja az `ID` azonosítójú `eredmény` entitáshoz rendelt `<Lang>_time` nevű attribútumban.

A levél-fogadó paraméterei és a spool állomány

Noha a komponens ezen része egyelőre nem készült el, határozott elképzeléseim vannak arról, hogy milyen felületen keresztül fog érintkezni a felhasználóval és a tesztelővel. Ebben a szakaszban két állomány felépítését mutatom be: az első a fogadóprogram paraméter-állománya, amelyben a 34. oldalon bemutatott paramétereket adhatjuk meg, a második a két program kapcsolatát biztosító spool állomány, amelyben a beérkezett levelek állnak sorba, tesztelésre várakozva.

```
semester('01s').

testclass(shw_pl1, test, date(2001,4,1), date(2001,4,14)).
testclass(bhw, test, date(2001,5,1), date(2001,5,18)).
```

6.4. ábra. A fogadóprogram egy lehetséges paraméter-állománya

A 6.4. ábrán a fogadóprogram egy lehetséges paraméter-állománya látható. Természetesen itt is Prolog-kifejezések állnak, amelyek lehetnek akár egyszerű tényállítások, akár eljárások, mivel ezt az állományt is közvetlenül a Prolog-interpreter értelmezi. Az állomány elhelyezése, akárcsak a tesztelő esetében, itt sincs megkötve. A kifejezések a következők lehetnek:

`semester(SemID).`
Azonosítja az aktuális szemesztert. Ez azért fontos, mert a fogadóprogram a leveleket félévenként külön archiválja.

`testclass(ClassID, SuiteID, Begin, End).`
A tervezési résznél elmondtam, hogy a tesztosztály a levél törzsében van kódolva, ez tehát nem paramétere a fogadóprogramnak. Az viszont igen, hogy egy adott tesztosztályt melyik tesztsorral teszteljük – tehát melyik spool állományt kell módosítani. Ezt az összerendelést kell megadni ilyen alakú kifejezésekkel: pontosan egy ilyen sornak kell léteznie minden lehetséges tesztosztályhoz (ha több van, a program csak az elsőt veszi figyelembe). A `Begin` és `End` argumentumok az adott tesztosztály beadási időszakának elejét és végét jelzik (inkluzíve). Mindkettő `date(Year,Month,Day)` alakú.

A spool állomány, amelyet a fogadóprogram épít, a tesztelő pedig fogyaszt, szintén Prolog-kifejezésekből épül fel. Minden kifejezés egy-egy tesztelési feladatot definiál.

`job(Name, Options).`
A `Name` azonosítójú hallgató programját kell legközelebb tesztelni. `Options` egy opciólista, elemei a következők lehetnek:

`extract(Version)`
Tesztelés előtt ki kell csomagolni a `Version` verziójú levelet. Ez a művelet kitörli a hallgató tesztkönyvtárát, ezért az ott elhelyezkedő állományok elvesznek!

`extract`
A *legfrissebb* levelet kell kicsomagolni tesztelés előtt.

`language(Lang)`
Csak az adott nyelven kell elvégezni a teszteket. Ilyenkor a naplóállomány neve eltér a szokásostól, ugyanis kap egy `.<Lang>` végződést. Az eredeti naplóállományt a tesztelés megkezdése előtt törli a tesztelő, hogy ne maradjanak érvényüket vesztett adatok.

job(Name).
Ekvivalens egy job(Name,[]) sorral, ilyenkor tehát a már kicsomagolt programokat teszteli, a tesztosztályhoz tartozó összes nyelven.

halt.
A tesztelő ezt sort még törli a spool állományból, de ezután befejezi a futását.

6.1.2. A komponens használata és működése

A program több Prolog-állományból áll (funkcionalitás szerint csoportosítva), de a save_program/2 könyvtári eljárás segítségével egyetlen futtatható állományt állítok elő belőle, így gyakorlatilag úgy használható, mint bármely más program. A működését elsődlegesen az argumentumokkal tudjuk befolyásolni, az első argumentuma ugyanis az igényelt szolgáltatás azonosítója. Ha itt help-et adunk meg, kapunk egy rövid ismertetést a használatról, ez látható a 6.5. ábrán.

```
guts> ./guts help
{restoring /home/guts/env/permanent/guts.Linux...}

Possible actions and mandatory arguments:
  testd    <config-file>
           run test daemon with specified configuration
  extract  <semester> <class> <name> [<version>]
           extract a mail of a student
  purge    <semester> <class>
           clean up mail directory, leaving only latest versions

  help     print this help screen
```

6.5. ábra. A GUTS program meghívása

A három fő szolgáltatás, amely jelenleg elérhető, a testd (teszt-démon), extract (levél kicsomagolása) és purge (archívum takarítása) argumentumokkal hívható elő. A teszt-démon futtatásához meg kell adnunk a már ismertetett szerkezetű paraméter-állomány nevét. A takarításhoz szükség van a félév és a tesztosztály azonosítójára, levelek kicsomagolásához ezen felül még a hallgató nevére is, és opcionálisan a verziószámra.

A működés a terv specifikációja és a paraméterek részletes ismertetetése után, úgy vélem, nem igényel külön magyarázatot. Kivételt talán csak a futási idők mérése és az időlimit figyelése jelent, ezekről érdemes pár szót ejteni.

Az időlimit betartatása POSIX-kompatibilis architektúrákon rendszer-szolgáltatás. Fontos megérteni, hogy az időlimit nem a valós időre vonatkozik, hanem az aktívan felhasznált CPU-időre. Ez utóbbi nem telik eseményre, pl. bevitelre várakozáskor. A korábbi években volt olyan állapota az NHF értékelő rendszernek, amikor ez a rendszer-szolgáltatás nem volt elérhető (egy másik program letiltotta). Ez abban a félévben sok fennakadást okozott, ezért még akkor írtam egy *idő-őr*-nek keresztelt C nyelvű programot, amely egyfelől aktiválja a rendszer-szolgáltatást, másrészt – a biztonság kedvéért – egy állítható szorzóval figyeli a valós időt is. Ez akkor is jól jön, ha a korlátozandó program eseményre várakozik, és ezért nem fogyasztja a CPU-időt. A limit leteltét (akár valós, akár CPU-időről van szó) speciális kilépési kóddal jelzi. Ha a program futása valamilyen hibakóddal szakad meg, akkor továbbengedi a hibakódot, tehát teljesen transzparens. A SICStus Prolog architektúra-független nyelv, a megfelelő rendszerhívás nem is érhető el közvetlenül, tehát duplán jól jártam az idő-őrrel.

A futási időt sem közvetlenül a Prolog-tesztelőben mérem, noha erre már van eljárás definiálva. Csakhogy ez a könyvtári függvény vagy a *saját processz* CPU-idő fogyasztását méri, vagy az eltelt valós időt, itt pedig a gyerek processzként indított hallgatói program CPU-idejére van szükség. Éppen ezért a Unix-architektúrákon meglévő time programot használom, amely képes ennek mérésére.

6.1.3. Értékelés

A C. függelékben olvasható egy minta naplóállomány, amelyet a GUTS rendszer állított elő. Ebben minden információ benne van, amit elvárhatunk. A fejléc pontosan tájékoztat a körülményekről és a tesztelt programról. A tesztesetek jól elválnak egymástól, a program eredményét és futási idejét jelző sor kitűnik környezetéből. A futtatás során kiírt hibaüzenetek és figyelmeztetések is megjelennek. Az eredményekről nyelvenként egy-egy rövid összesítés is olvasható. A naplóban megjelenő fix szövegek mind a sablonok részei, ezért könnyen módosíthatóak.

A rendszer paramétereivel hatékonyan és tömören le lehet írni a tesztfeladatot. Az ismertetés során bemutatott példákból is látszik, hogy a rendszer alkalmas NHF-ek tesztelésére, de a paraméterek lehetővé teszik KHF-ek és más jellegű feladatok ellenőrzését is. Elmondható, hogy egy újabb feladattípus ellenőrzéséhez minimális mennyiségű programozásra van szükség.

Természetesen akad még számtalan kisebb-nagyobb hiányosság, melyek pótlása a GUTS rendszer aktív használatának előfeltétele. A két legfontosabb ezek közül a levelek fogadása és az adatbázis frissítése. Úgy gondolom azonban, hogy a program „befejezése” nem igényel nagyon sok munkát, és valószínűnek tartom, hogy a következő félévben már ez a rendszer fogja értékelni a hallgatók megoldásait.

6.2. Hívási gráf építése funkcionális nyelvekre

A 4.1.3. fejezetben röviden beszámoltam egy olyan rendszerről, amely képes forráskódú programokról megállapítani, hogy másolatai-e egymásnak. Azt is leírtam, hogy a jelenleg működő program csak Prolog-programokat képes összehasonlítani, de a bővítési lehetőség adott, hogy más nyelveket is támogasson. Mivel a *Deklaratív programozás* tárgy keretében a Prolog mellett SML nyelvet is oktatnak, indokolt, hogy a hasonlóságvizsgáló program mihamarabb alkalmas legyen SML-programok ellenőrzésére is. Ehhez csak arra van szükség, hogy legyen egy olyan alprogram, amely képes feltérképezni egy SML-program hívási gráfját, és ezt megfelelő formátumban átadni. A *hívási gráf* nem más, mint az a gráf, amelyet a következőképpen kapunk:

1. minden SML-függvényre megállapítjuk, hogy mely más függvényeket hív;
2. a gráf csomópontjai a függvények;
3. a köztük futó *irányított* élek jelzik, hogy melyik melyiket hívja.

A diplomatervezés keretében elkészítettem egy SML nyelven írt programot, amely pontosan ezt a feladatot látja el. Mielőtt azonban rátérnék a konkrét megvalósítás ismertetésére és a hasonlóságvizsgáló programmal való összeillesztésének vizsgálatára, érdemes kicsit elidőzni a funkcionális nyelvek hívási gráfjai körüli általános érvényű kérdéseknél, amelyek a gráfépítő program írása közben merültek fel. Nem adok minden kérdésre teljes választ, de mindenütt igyekszem valamilyen módszert javasolni a probléma feloldására. Mivel az eddigiekben három programról volt szó, és a továbbiakban is szükséges lesz mindháromra hivatkozni, az alábbi logikus elnevezéseket vezetem be:

- *hasonlóságvizsgáló* programnak – röviden hasonlóságvizsgálónak – nevezem a Lukácsy Gergely készítette programot;
- *gráfépítő* programnak – röviden gráfépítőnek – nevezem azt a hipotetikus programot, amelynek készítéséhez tanácsokat adok, és amelynek egy egyszerűbb változatát magam is elkészítettem;
- *SML*-programnak – röviden programnak – nevezem azokat az SML nyelvű programokat, amelyek hívási gráfját fel kell építeni.

Többször szót fogok ejteni az *SML-értelmezőről* is. Amikor pedig úgy fogalmazok, hogy „$\mathcal{A}$ helyen *regisztráljuk* $\mathcal{B}$ függvény meghívását”, akkor arra gondolok, hogy a gráfépítő program az $\mathcal{A}$-hoz tartozó csomópontból a $\mathcal{B}$-hez tartozó csomópontba vezető élet vesz fel a gráfba; azaz feltételezi, hogy $\mathcal{B}$-t $\mathcal{A}$-ból meghívták.

6.2.1. A gráfépítés dilemmái

Hívási gráf építésekor funkcionális nyelveknél felmerül egy-két olyan kérdés, amely más – akár deklaratív, akár imperatív – nyelveknél nem jön elő. Ennek fő oka a funkcionális nyelvek alapvető jellegzetességeiben, a megalkotásuk alapjául szolgáló függvényelméletben keresendő. Azok számára, akik nem járatosak a funkcionális nyelvekben, egy igen rövid ismertető olvasható a D. függelékben. Az ebben foglaltak ismerete elengedhetetlen a fejezet további részének megértéséhez.

A 6.2.2.–6.2.4. szakaszokban felvetek néhány problémát, amely a hívási gráfok építését megnehezíti. Ezek egy része általában a funkcionális nyelvekre jellemző, mások SML-specifikusak. A problémákra rögtön elméleti megoldásokat is javasolok, de ezek egy része – mint látni fogjuk – a gyakorlatban csak nehezen kivitelezhető.

6.2.2. A függvényértékek problémaköre

Az általában a funkcionális nyelvekre érvényes problematikus kérdések jelentős része abból a jellegzetességből adódik, hogy a funkcionális programozásban a *függvényérték* (azaz olyan érték, amely egy függvényt képvisel) éppen olyan érték, mint az összes többi. Ebből adódóan sokkal rugalmasabban használható, ugyanakkor nehezebb a felhasználását nyomon követni. Az alábbi felvetések a D. függelék egy-egy témaköréhez kapcsolódnak.

Függvények „átnevezése”

Vegyük a függelékben is bemutatott egyszerű példát:

```
- val f = cos;
> val f = fn : real -> real
```

Ha most `f`-et alkalmazzuk egy értékre, akkor melyik függvényt hívjuk meg: `f`-et vagy `cos`-t? Ha pontos választ akarunk adni, akkor azt kell mondanunk, hogy voltaképpen egyiket sem, hanem azt a függvényt, amelyet mindkét név *jelöl*. Praktikus szempontból azonban talán egyszerűbb, ha azt mondjuk, hogy a `cos` függvényt hívtuk meg, mivel ezt definiáltuk elsőként (a könyvtár részeként). Ennek kiderítéséhez a nyelv szintaktikáján túl természetesen ismerni kell bizonyos mértékben a szemantikáját is, azaz a gráfépítőnek meg kell értenie, hogy mit jelent a `val f = cos` kifejezés, és meg kell jegyeznie, hogy az `f` név ettől a ponttól kezdve a `cos` függvény egy másik neve.

Mivel a már egyszer lekötött nevek átdefiniálhatóak, megtehetjük, hogy például a `cos` nevet egy újabb függvényértékhez kötjük. Noha egy ilyen átnevezésnek nem lenne túl sok értelme, és az SML-program átláthatóságát is nagy mértékben rontaná, „jó” eszköz lehet arra, hogy becsapjuk a hasonlóságvizsgáló programot, ha a gráfépítő erre a lehetőségre nem ügyel. A probléma az, hogy a hívási gráf – az SML-program állapotával ellentétben – statikus, egyszerre kell értelmezni az egészet. Ha egyszer használtuk a `cos` nevet, akkor attól kezdve mindig ugyanazt kell jelölnünk vele, az ilyen átdefiniálást tehát másképp kell kezelni. A megoldás az lehet, hogy a gráfépítő automatikusan átnevezi az újradefiniált függvényt (mondjuk `cos2`-re), és ettől kezdve ezen a néven hivatkozik rá (azaz minden további `cos` hívás helyett `cos2`-t regisztrál). Így olyan, mintha nem a `cos` nevet definiáltuk volna fölül az SML-programban, hanem egy teljesen új nevet hoztunk volna létre, és ettől a ponttól kezdve mindig azt használtuk volna az eredeti `cos` helyett.

Anonim függvények

A lambda-jelöléssel definiált anonim függvények létezése újabb problémát jelent. Vegyük a következő példát:

```
- (fn x => x*x) 12;
> val it = 144 : int
```

Ebben az esetben melyik függvényt hívtuk meg? Nyilvánvalóan nem a `*` (egész szorzás) függvényt, legalábbis nem közvetlenül. Amit meghívtunk, annak viszont nincs neve. Ilyenkor a gráfépítő programnak valamilyen módon el kell neveznie az anonim függvényt, és innentől kezdve ezzel a névvel hivatkoznia

rá. Az „innentől kezdve” fordulatot az előző mondatban az indokolja, hogy egy anonim függvény is kaphat nevet, ha például argumentumként adjuk át egy magasabb rendű függvénynek. (Ilyenkor nem szabad az adott argumentum nevét felhasználni, hiszen az általa jelölt érték mindig más és más.)

Részlegesen alkalmazható függvények

Ahogy azt a függelék D.4. fejezetében is leírtam, egy részlegesen alkalmazható függvény definiálásakor voltaképpen több függvény definíciójáról van szó. Vegyük például a következő függvénydefiníciót:

```
- fun add a b = a+b;
> val add = fn : int -> int -> int
- add 2;
> val it = fn : int -> int
```

A harmadik sorban meghívtuk az `add` függvényt? Szigorúan véve nem, hiszen a definíciójában megadott `a+b` kifejezést még nem értékeltük, nem értékelhettük ki. Ugyanakkor mégiscsak meghívtuk, hiszen az általa jelölt függvényértéket *alkalmaztuk* egy másik értékre, és mi ez, ha nem a függvény meghívása. A látszólagos ellentmondást talán úgy lehet a legjobban feloldani, hogy a hívási gráfba nem egy, hanem két függvény definícióját vesszük fel, mondjuk `addA`-t és `addB`-t. Amikor egy `add 2` jellegű kifejezéssel találkozunk, azt mondjuk, hogy meghívtuk `addA`-t (hiszen lekötöttük az első argumentumot), amikor pedig ezt a függvényértéket alkalmazzuk egy újabb értékre, akkor `addB` meghívásáról beszélünk. Ha egy részlegesen alkalmazható függvénynek nem két, hanem több ilyen argumentuma van, akkor azt természetesen nem két, hanem több függvény definíciójának kell tekintenünk.

Ha így döntünk, akkor felmerül egy újabb probléma. Az `add` függvényben törzsében lévő hívásokat melyik újonnan létrehozott függvényhez kell regisztrálnunk: `addA`-hoz vagy `addB`-hez? Általánosságban azt mondhatjuk, hogy mindig a legutolsóhoz, hiszen a részlegesen alkalmazható függvények törzse addig nem értékelődik ki, amíg az összes argumentumot meg nem határoztuk. Sajnos azonban lehetséges úgy definiálni egy függvényt, hogy az részlegesen legyen alkalmazható, és már az első argumentuma alapján végezzen számításokat (természetesen más függvényeket meghívva), például:

```
- val pyth = fn a => let val a2 = a*a
                     in fn b => sqrt(a2 + b*b) end;
> val pyth = fn : real -> real -> real
```

Azaz egyszerűen egymásba ágyazunk két anonim függvényt. A lényeg, hogy a `val a2 = a*a` definícióban szereplő kifejezés már az első argumentum meghatározása után kiértékelődik, és a visszaadott függvényértékben már a kiszámított érték szerepel. Ebben az esetben, ha precízek szeretnénk lenni, azt kell mondanunk, hogy a `*` (valós szorzás) függvényt `pyth2A` is meghívta (`pyth2B` is meghívja a `b*b` kifejezés kiértékelésekor). Ennek kiderítése viszont már meglehetősen bonyolult feladat.

Függvényérték mint argumentum

Tekintsük egy magasabbrendű függvény alkalmazásának a következő példáját:

```
- map (fn x => x+1) [1,4,9];
> val it = [2, 5, 10] : int list
```

Felmerül a következő probléma: az `(fn x => x+1)` anonim függvényt hol hívjuk meg? Tulajdonképpen nem a legkülső szinten, hiszen csak továbbadjuk a függvényértéket a `map`-nek, és nem alkalmazzuk egy másik értékre. Ugyanakkor azt is furcsa lenne állítani, hogy a `map` hívja meg, hiszen a `map` definíciójában elő sem fordul ez a kifejezés. Gondoljuk csak meg: ha azt mondjuk, hogy a `map` hívja a függvényt, akkor a hívási gráf `map`-hez tartozó csomópontjából *kiinduló* élek attól fognak függni, hogy a `map`-et hány helyen és hogyan hívjuk meg az SML-programban! Mivel első ránézésre egyik lehetőség sem tűnik jónak, vizsgáljuk meg alaposabban mindkettőt.

Tegyük fel, hogy a hívást ott regisztráljuk, ahol a függvény *neve* (anonim esetben a definíciója) szerepel (azaz a példa esetében a legkülső szinten). Ebben az esetben nagyon könnyű becsapni a hasonlóságvizsgáló programot, mert nem kell mást tenni, mint a függvény alkalmazásának helyéről a nevet

egy külsőbb hívási szintre vinni, és csak a függvényértéket továbbadni egy argumentumban. Az SML-program ettől kuszább lesz, de ha a cél a másolás leplezése, akkor ez nem akadály. Ez a fajta regisztráció tehát nem szerencsés, ugyanakkor meglehetősen egyszerű, mert nincs szükség hozzá a nyelv szemantikájának ismeretére, puszta szintaktikai elemzéssel megállapítható a hívás helye.

A másik lehetőség, hogy mégiscsak azon a helyen regisztráljuk a hívást, ahol a függvényértéket alkalmazzuk egy argumentumra. Ebben az esetben végig kell követni a függvényérték átadogatását (esetleg több szinten keresztül is), és figyelni, hogy hol hívódik meg. Noha ez a megoldás lényegesen bonyolultabb, leginkább azért, mert a már definiált függvényekhez tartozó részgráfot újra és újra módosítani kell, elméleti szempontból mégiscsak ez a jobb választás, hiszen ez felel meg jobban a funkcionális nyelvek logikájának. Az imént vázolt probléma, azaz a `map` és más könyvtári függvényekhez tartozó részgráfok változékonysága megszüntethető egy ügyes trükkel. Ez pedig az, hogy magasabb rendű *könyvtári* függvények hívása esetében mégiscsak a hívóhoz jegyezzük be az argumentumként átadott függvény meghívását. Ez ugyan torzítja a hívási fát, mégis több információt árul el a programról magáról, és jobban elkülöníti a könyvtári részeket, amelyek – lévén, hogy a legtöbb program használ könyvtári elemeket – kevésbé informatívak.

Függvényérték mint visszatérési érték

Vegyük a következő példát:

```
- nth([sin,ln,sqrt], 2) e;
> val it = 1.6487212707 : real
```

Ha a gráfépítés során következetesen azt a logikát követjük, hogy a függvények meghívását azon a helyen regisztráljuk, ahol alkalmazzuk őket, akkor itt nagy gondban vagyunk. E logika alapján az, hogy a `sin`, `ln` és `sqrt` függvények melyikét hívtuk meg, csak futási időben deríthető ki, hiszen a `2` helyén tetszőlegesen bonyolult kifejezés állhat. Ha ismét a hallgató fejével gondolkozunk, a fenti `nth` hívás remek eszköz az `sqrt` függvény meghívásának leplezésére. Ezen a ponton csődöt mond minden korábbi elképzelés arról, hogy a hívási gráf pontosan felépíthető anélkül, hogy futtatnunk kellene a programot.

6.2.3. Egyéb problémák

Van néhány kisebb probléma, amely ugyan nem olyan tulajdonságokból adódik, amelyek csak a funkcionális nyelvekre lennének jellemzőek, az előzőekkel kombinálva mégis érdekes helyzetek adódhatnak.

Lokális kifejezések és deklarációk

Természetesen semmi akadálya, hogy függvényt deklaráljunk lokálisan. Erre a függelékben is láthatunk példát a D.7. fejezetben. Ha ilyesmivel találkozik a gráfépítő, akkor ügyelnie kell arra, hogy a lokálisan deklarált név elfedhet korábban deklaráltakat, ám a blokkból kilépve ez a fedés megszűnik. A helyzet nem ugyanaz tehát, mint a nevek újradefiniálásánál, ahol a régi jelentést el lehetett dobni.

Kölcsönös rekurzió

Ahogy a C-ben lehetséges *kölcsönösen rekurzív*, azaz egymást hívó függvények definiálása (predeklarációval), úgy az SML-ben is lehetőség van erre. Az ilyen esetekre a kölcsönösen rekurzív függvények definiálásakor egy speciális kulcsszóval kell felhívni a fordító figyelmét. Mivel a funkcionális programok értelmezése alapvetően szekvenciális, általában nincs szükség rá, hogy gráfépítéskor előre tekintsünk, csak a korábban definiált függvényeket kell figyelembe vennünk. Itt más a helyzet, hiszen a kölcsönösen rekurzív függvények egymást hívják.

Szintén lehetséges kölcsönösen rekurzív függvények definiálása lokális kifejezéssel. Az ebben definiált függvények ui. meghívhatják a kifejezést magába foglaló külső függvényt.

Modulok használata

Az SML – modern programozási nyelvhez illően – moduláris nyelv. A könyvtári függvények különböző modulokban vannak definiálva, és nagy részüket csak a modulnév megadásával tudjuk elérni, pl. így: `Math.cos`. A nyelv azonban lehetőséget ad a modulok névterének megnyitására, ilyenkor az adott modulban definiált nevek bemásolódnak az aktuális névtérbe, és a rájuk való hivatkozáskor már nem kell megadni a modulnevet. Ilyenkor a gráfépítőnek észre kell vennie, hogy továbbra is ugyanarról a függvényről van szó. Egy modul megnyitása természetesen el is fedhet korábban definiált neveket.

A könyvtáriak mellett a saját magunk által írt függvényeket is modulokba szervezhetjük. Ily módon a programunkat több önálló részre oszthatjuk. Az ideális gráfépítőnek ilyen esetekben fel kell térképeznie, hogy a modulok hogyan hivatkoznak egymásra, és mindegyik modult fel kell dolgoznia.

A modulok megnyitása kombinálható a lokális kifejezésekkel és deklarációkkal. Ilyenkor a modul névtere csak az adott kifejezés vagy deklaráció erejéig nyílik meg, a blokkból kilépve elveszik. Ilyen esetekben ismét előfordulhat egy-egy név átmeneti elfedése.

Infix függvénydefiníciók

Az SML-program feldolgozása közben a gráfépítőnek nyilván kell tartania a már definiált függvényeket, és ehhez persze tudnia kell, hogy egy függvény neve honnan olvasható ki. Általában ez a közvetlenül a `fun` vagy `val` kulcsszó után következő név, kivéve, ha előtte a függvényt infixnek deklaráltuk, mert ilyenkor az első argumentum-mintát *követő* név az, hacsak nem az `op` kulcsszóval együtt adjuk meg, mely esetben mégiscsak az első név a függvény neve. Másként fogalmazva két eset lehetséges:

1. A függvény nem infix, vagy az, de a nevét az `op` kulcsszó előzi meg: ilyenkor a definícióban szereplő legelső név a függvény neve.

2. A függvény infixnek van deklarálva, és a nevét nem előzi meg az `op` kulcsszó: ilyenkor a második név a függvény neve.

Amint látható, ahhoz, hogy kideríthessük, mi a függvény neve, tudnunk kell róla, hogy infix-e, ehhez pedig tudnunk kell a nevét – ördögi kör jött létre. Szerencsére ez feloldható úgy, hogy először infixnek feltételezzük, és megvizsgáljuk, hogy a második helyen szereplő név valóban infix-e. Ha igen, megvagyunk. Ha nem, akkor prefixnek feltételezzük, és megnézzük, hogy valóban *nem* infix-e vagy az `op` kulcsszóval van-e definiálva. Ha igen, ismét minden rendben. Ellenkező esetben olyan helyzetet találtunk, ami nem fordulhat elő.

6.2.4. Gráfépítés futási időben

Az eddigiek alapján levonható az a következtetés, hogy a hívási gráf építése az SML-programok forráskódjának tanulmányozása alapján rengeteg buktatóval jár, sőt bizonyos esetekre nem is tud megoldást nyújtani. Érdemes elgondolkozni azon a lehetőségen, hogy a gráfot inkább futási időben építsük fel. Erre három mód kínálkozik.

1. Az SML-értelmező módosítása nem vezethet eredményre, mert az SML-ben a nevek nem tárolódnak az értelmező számára lefordított kódban. Ennek az a jótékony hatása, hogy egy függvény újradefiniálása – pontosabban: egy név új függvénydefinícióhoz rendelése – nem módosítja azon függvények viselkedését, amelyek meghívták, hiszen azok lefordított változatai nem a névre, hanem az általa jelölt függvényértékre hivatkoznak.

2. Érdekes, de nagy lélegzetvételű feladat lehet az SML-fordító módosítása úgy, hogy az általa lefordított függvények elejére odafordítsa azt a kódrészletet is, amely kibővíti a hívási gráfot a megfelelő éllel. Ehhez valamilyen módon azt is meg kell jegyezni, hogy melyik függvény hívta meg. Erre két megoldás kínálkozik:

   (a) A fordító minden függvényt egy további argumentummal lát el, amelyben automatikusan átadja a hívó nevét. Ennek a megoldásnak az a hátránya, hogy az ilyen módon lefordított függvények nem képesek együttműködni az extra információk nélkül fordítottakkal, így a könyvtári függvényekkel sem, hiszen más az argumentumok száma.

(b) A fordító vermet épít, amelyben a függvények neve szerepel.[2] Ennek a megoldásnak az az érdekessége, hogy a könyvtári függvények „átlátszóak” lesznek, azaz a hívási gráfból egyszerűen kimaradnak, hiszen a verembe sem kerülnek be, és a gráfot sem módosítják. Ezt akár előnynek is tekinthetjük, hiszen a könyvtári függvények használata minden programra jellemző, nem pedig egyedi jellegzetesség.

3. Módosíthatjuk magát az SML-forrást is úgy, hogy ugyanezeket a feladatokat elvégezze, tehát építsen hívási gráfot és vermet. Ez is kellemetlen és nehezen kivitelezhető megoldás lehet amiatt a szintaktikai sokszínűség miatt, amely annyiféleképpen engedi meg függvények definiálását, és amelyből már a statikus gráfépítés boncolgatásánál is annyi probléma adódott.

6.2.5. SML hívási gráf építésének megvalósítása

Az előző fejezetben elmondottakat alaposan átgondolva én a statikus, forráskód alapján történő gráfépítést választottam a megvalósításkor. Ennek oka a következő volt: a korábban ismertetett problémák – főként azok, amelyek megoldása igen körülményes vagy csaknem lehetetlen – jelentős része csak ritkán bukkan fel az átlagos SML-programokban, főként azokban, amelyeket olyan hallgatók írnak, akik csak fél éve ismerkednek a nyelvvel. Merem állítani, hogy itt is igaz a közismert „80-20 szabály”, azaz a problémák 80 százaléka csak az esetek 20 százalékában jön elő, sőt könnyen meglehet, hogy még ennél is kedvezőbb a helyzet. Ez alapján tehát kijelenthető, hogy egy viszonylag primitív, a felvetett problémákat nem vagy csak részben kezelő gráfépítő algoritmus is kellően hatékony lehet.

Az elkészült gráfépítő program ismertetését néhány technikai részlettel kezdem, majd kitérek arra is, hogy az előbbiekben vázolt problémák közül melyeket kezeli a program, és melyeket nem, és ezen döntéseimet meg is indoklom.

Technikai részletek

Az MOSML-interpreter – mint már említettem – szabadon hozzáférhető, ráadásul a forráskódja is tetszés szerint módosítható, részei szabadon felhasználhatóak. Ennek a forrásnak csupán a hatékonyság szempontjából legkritikusabb része, a byte-kód értelmező van C-ben írva, a többi, beleértve a *szintaktikai elemzőt* és az erre alapuló fordítót is, SML-ben. Ezt az elemzőt használtam fel a programomban, segítségével ugyanis beolvasható egy szintaktikai hibáktól mentes SML-program és átalakítható egy meglehetősen összetett SML-adatstruktúrává. Ebben a struktúrában rengeteg olyan adat van, amelyre a hívási gráf felépítéséhez nincs szükség, és szerkezetének kialakításakor inkább a hatékony kezelhetőség, mintsem az átláthatóság volt szempont, éppen ezért nem volt egészen triviális kinyerni a szükséges információkat. Egy helyen a fordítóba is „bele kellett nyúlnom”, mert nem látszott ki egy függvény[3], amelyet meg szerettem volna hívni.

Ezek után nem meglepő, hogy maga a gráfépítő is SML-ben készült. Ennek két kézenfekvő oka is volt: az egyik, hogy így fel tudtam használni a kész szintaktikai elemzőt, a másik, hogy az SML nyelvet meglehetősen jól ismerem és hatékonyan tudom használni. A deklaratív nyelvek tömörségére jellemző, hogy a teljes gráfépítő program (az elemzőt nem számítva) kevesebb, mint 500 sor. A program kimenete egy szöveges állomány, amely kódoltan, Prolog-kifejezések formájában – mivel a hasonlóságvizsgáló program Prologban készült, és ilyen kifejezéseket tud kényelmesen beolvasni – tartalmazza az adott SML-program hívási fáját. Szerencsére ez a megkötés nem okozott nagyobb gondot, mintha bármilyen más szöveges formában kellene kiírni az adatokat. A könyvtári függvények hívásának felismeréséhez természetesen rendelkezésre kell, hogy álljon a könyvtári függvények teljes listája. Ez szerencsére kinyerhető a könyvtári modulok forrásából, ill. a hozzájuk adott ún. szignatúra állományokból. Ehhez egy egyszerű szkriptet készítettem, amely előállítja a szükséges adatokat tartalmazó, a gráfépítővel összeszerkesztett SML-modult. Ennek a megoldásnak az is az előnye, hogyha bővül az MOSML könyvtár, akkor pillanatok alatt kényelmesen adaptálni lehet a változásokhoz a gráfépítő programot is.

[2] Az SML-programok futása során természetesen hagyományosan is épül verem, ezt azonban közvetlenül az értelmező és nem a fordító által generált kód építi.

[3] Az SML is moduláris nyelv, támogatja az adatok elrejtését.

Hiányosságok

Alapvető döntést kellett hoznom a tekintetben, hogy a gráfépítő foglalkozzon-e szemantikai kérdésekkel is, vagy csak az elemző által kinyert nyers szintaktikai információkat használja fel. Sajnos egy alapos szemantikai elemzés lényegesen bonyolultabb, ezért úgy döntöttem, hogy a legtöbb kérdésben csak a szintaktikát veszem figyelembe. Mint azt már írtam, megfigyeléseim szerint egy relatíve egyszerű program is kellően pontos hívási gráfot tud építeni, ezért a szemantikai elemzés „ára" meglehetősen nagy lenne a haszonhoz képest. E döntésemből adódóan a *függvényértékek problémakörére* javasolt megoldásaimat nem alkalmaztam a programomban. A gráfépítő a hívást mindig ott regisztrálja, ahol az egyes függvények neve szerepel, emiatt az anonim függvények meghívását sem kezeli (hiszen azok mindig pontosan egy helyen szerepelnének a gráfban). A szemantika megfelelő szintű kezelése egy következő fejlesztési fázis részeként kerülhet ismét szóba.

Ugyanebből az okból a program nem ismeri fel a `val` kulcsszóval definiált függvényeket. Ilyen esetekben ugyanis a függvény törzsét vagy a már ismertetett lambda-jelöléssel kell megadni, vagy valamely másik függvény részleges alkalmazásával (mint pl. `val addTwo = add 2`). Ennek felismeréséhez viszont a gráfépítőnek ismernie kellene, hogy mely függvények alkalmazhatóak részlegesen, és fel kellene ismernie az alkalmazásukat is. Jelenleg csak a `fun` kulcsszavas definíciókat kezeli, ezek ugyanis pusztán szintaktikai információk alapján felismerhetők. Gyakorlati szempontból azért nem kritikus ez a hiányosság, mert az SML-programok átláthatósága kedvéért nem szokás a `val` kulcsszó segítségével függvényeket definiálni.

Egy függvényt természetesen egy nem-függvény értékkel is elfedhetünk. A program jelenlegi változata nem veszi észre, hogy egy név nem függvény többé, ilyenkor tévesen az előző függvény meghívásaként értékeli az adott név egy-egy előfordulását. Ez a hiányosság kellemetlen: gyakran előfordul, hogy egy-egy hallgató nem tud róla (vagy nem jut eszébe), hogy létezik pl. `length` nevű könyvtári függvény, és a névhez egy egész értéket köt (mondjuk egy füzér hosszát). Szerencsére a hívási gráf ebből adódó torzulása sem túl jelentős.

A program egyelőre nem kezeli jól a több modulból álló programokat sem, azaz nem képes felismerni, hogy egy állomány gráfjának felépítéséhez először egy másik állományt kell feldolgoznia. Mivel a beadott házi feladatok az esetek elhanyagolhatóan kis részében készülnek több modullal, ez a hiányosság sem túl kritikus, de a közeljövőben célszerű kiküszöbölni.

Kezelt problémák

A 6.2.3. fejezetben ismertetett „egyéb" problémák többségét megfelelően kezeli a gráfépítő. Jól kezeli a lokális kifejezések és deklarációk kérdését, például az olyan eseteket, amikor egy lokális kifejezésben átmenetileg felüldefiniáljuk a könyvtári `sin` függvényt. Ha ilyenkor a kifejezésben szerepel a `sin` hívás, a gráfépítő észreveszi, hogy az nem a könyvtári függvényre, hanem a lokálisan definiáltra vonatkozik. Ehhez a lokálisan definiált függvényeknek egyedi nevet generál.

Ugyancsak jól működik a könyvtári modulok kezelése. A gráfépítő megjegyzi a modulok megnyitását, és a modulbeli függvények meghívásakor továbbra is az adott függvény teljes nevét (tehát modulnévvel együtt) regisztrálja a hívási gráfban.

A gráfépítő program jól kezeli a *kölcsönösen rekurzív* függvényeket is, tehát a függvényként felismert nevek listája nem teljesen lineárisan bővül, hanem szükség esetén blokkokban.

Szintén jól működik az infixnek deklarált operátorok definíciójának kezelése. Ehhez nyilvántartom az infix deklarációkat és az ezek hatását megszüntető nonfix deklarációkat is.

Futási példa

Tekintsük az alábbi rövid – kissé értelmetlen – példaprogramot, amely a gráfépítő által kezelt problémák egy részét jól demonstrálja.

```
structure Pro = struct

fun f x = x+1

local fun f (x,y) = x+y
```

```
in fun g (SOME xy) = f xy
     | g NONE = let fun f () = 0 in f() end
end

fun h s = implode (i (map (chr o f o ord) (explode s)))
and i (c::cs) = c :: explode (h (implode cs))
  | i [] = []

end
```

A példában az f függvény három változata szerepel: egy globális, egy lokális deklarációban előforduló, és egy lokális kifejezésben előforduló. A h és az i függvények kölcsönösen egymást hívják a számos könyvtári függvény mellett. Most pedig lássuk, mit produkál belőle a gráfépítő!

```
match:call_graph([
  'Pro.f'-['General.+'],
  'Pro.$local1.f'-['General.+'],
  'Pro.g.$2.f'-[],
  'Pro.g'-['Pro.$local1.f','Pro.g.$2.f'],
  'Pro.h'-['Misc.implode','Pro.i','Misc.map','Misc.chr','General.o',
           'Pro.f','General.o','Misc.ord','Misc.explode'],
  'Pro.i'-['Misc.explode','Pro.h','Misc.implode']]).

match:non_called_preds([]).

match:useless_preds(['General.+','General.o','Misc.chr','Misc.explode',
                     'Misc.implode','Misc.map','Misc.ord']).
```

A hívási gráf (a call_graph/1 funktorú kifejezés argumentuma) a három f függvénnyel kezdődik, látható, hogy mindháromnak más a neve, amely a definíciójuk helyétől függ. Ezután következik g törzse, amelyben két f változat is meg van hívva. Végül a h és i függvények következnek, amelyek több könyvtári függvény mellett egymást is hívják. A useless_preds/1 funktorú kifejezés argumentumában a könyvtári függvények vannak pontosan egyszer felsorolva, mivel ezek kevesebb információt adnak a programról, és így lehetővé válik, hogy a hasonlóságvizsgáló program a gráfból kihagyja őket. Az is látszik, hogy a General és Misc könyvtári modulok alapértelmezés szerint meg vannak nyitva, azaz egy sima explode hívást a gráfépítő – helyesen – Misc.explode-ként értelmez. A non_called_preds/1 funktorú kifejezés argumentuma mindig üres lista – ide akkor lehetne értelmes adatot írni, ha az SML-programot futtatnánk is, és futás közben figyelnénk, hogy mely függvények nem hívódnak meg. Erre a gráfépítő jelenleg nem képes, és nem tartom valószínűnek, hogy egyhamar képes lesz. Ehhez ugyanis az SML-programok fordítási folyamatába kell beavatkozni oly módon, hogy az egyes függvények neve megőrződjék. (Erről arra a 6.2.4. szakaszban már esett szó.)

6.2.6. Értékelés

Összességében elmondható, hogy a program egyszerűsége ellenére megbízhatóan és hatékonyan képes egy egyetlen modulból álló SML-program hívási gráfjának elfogadható pontosságú feltérképezésére. A „rosszindulatú támadásoknak" ugyan nem tud igazán ellenállni, de ezen a szinten elfogadható az a védelem is, hogy az érdekeltek egyszerűen nem tudják meg a program hiányosságait.

A programot természetesen a gyakorlatban is kipróbáltam, méghozzá úgy, ahogyan a jövőben használni fogjuk: az általa előállított kimenet alapján SML-programokat hasonlítottam össze Lukácsy Gergely programjával. A tapasztalatok kedvezőek: több tanév házi feladataira küldött megoldásokhoz generáltam hívási gráfokat, és minden esetben (mintegy 150 programra) értelmes adatokat szolgáltatott. Az összehasonlítás kimutatta, hogy mindhárom vizsgált tanévben volt másolás, és ezt a manuális vizsgálat meg is erősítette.

Megállapítható tehát, hogy a gráfépítő program alapvetően megfelel elsődleges céljának, noha számos ponton fejlesztésre szorul.

7. fejezet

Tapasztalatok

A szakdolgozat jelen fejezetének első felében az ETS rendszer tervezése során összegyűjtött tapasztalataimat ismertetem. Ennek során először értékelem a rendszert, majd beszámolok a továbbfejlesztési lehetőségekről.

7.1. Az ETS értékelése

Tekintettel arra, hogy az elkészült munka jelentős részében elméleti jellegű, tesztelésről, gyakorlati alkalmazásról nem lehet szó. De meg lehet vizsgálni azt, hogy honnan indultam el és meddig jutottam a kitűzött célhoz vezető úton. Vagyis – konkrétabban megfogalmazva – mennyiben léptem túl a meglévő programok keretein, és mennyiben értem el azt a célt, hogy általános, összefüggő rendszert hozzak létre.

Ennek vizsgálatához érdemes még egyszer végigvenni a megalkotott rendszer komponenseit, és külön-külön vizsgálni őket.

Az **adatbázis** terve merőben új. A meglévő programok mind külön adattárakat használnak, amelyek csak a számukra fontos adatokat tárolják. Ráadásul az adatok formátuma is minden esetben más és más, főként mivel a megvalósításuk ad-hoc jellegű és nem tervezett volt. Ebből adódóan az adatok nehezen fellelhetőek, egy részük többszörösen van tárolva, és a különböző reprezentációk közötti konverzió is sok bonyodalmat okoz. Az elkészült terv, úgy hiszem, kellően pontos ahhoz, hogy ennek alapján könnyen meg lehessen valósítani az adatbázist. A felvázolt szerkezet a céloknak megfelelően nem csak a már azonosított adatcsoportokat képes tárolni, hanem kellően rugalmas ahhoz, hogy a jövőbeni fejlesztéseknek, bővítéseknek is teret adjon. Egységességének köszönhetően biztosítja a komponensek közötti kapcsolattartást, megszünteti a konverziós problémákat. Struktúráltságának köszönhetően minden szükséges adat gyorsan megtalálható. Természetesen előfordulhat, hogy bizonyos előre nem látott változásokhoz mégiscsak szükség van a terv módosítására, például a reprezentált események körét kell bővíteni, vagy a hallgatók nyilvántartott adatai bizonyulnak hiányosnak. Ugyanakkor ennek az ellenkezője is előfordulhat: kiderülhet bizonyos adatokról, hogy teljesen feleslegesen vannak tárolva, és soha nincs rájuk szükség.

A **házi feladatokat feldolgozó komponens** ötlete a legrégibb, ebből adódóan a legtöbb tapasztalat is itt áll rendelkezésre és itt van a legjobban kialakult kép arról, hogy pontosan mire is van szükség. Az erre vonatkozó tervek a jelenleg használt program ismeretében nem tartalmaznak „korszakalkotó” ötleteket, csupán pontosítják, tisztázzák a feladatokat és az ezek megoldásához szükséges struktúrát. Ugyanakkor meg kell említenem, hogy a meglévő program kifejlesztésében is meghatározó szerepem volt, tehát a felhasznált ötletek is javarészt a sajátjaim, ha nem is ebben a félévben születtek. A szakdolgozatban elkészült tervekkel sikerült egyesítenem a házi feladatok különböző fajtáinak ellenőrzését, és további kategóriák bevezetése is lehetővé vált. A rugalmas struktúrának köszönhetően a tesztelendő programok és a felhasznált tesztadatok mérete tág határok között mozoghat, az ellenőrzés során mindig a méretekhez igazodó reprezentációt választhatunk. A komponens legfőbb erénye, hogy teljesen nyelvtől független, így akár más programozási tárgyaknál is felhasználható.

A **gyakoroltató komponens** tervének elkészítéséhez szintén sok ötletet merítettem a meglévő programból, főleg a megjelenést illetően. Alaposan átterveztem ugyanakkor a belső struktúrát, amely ez-

által reményeim szerint rugalmasabbá és könnyebben programozhatóvá vált. Ebben a szerkezetben technikailag nem jelent gondot egy újabb témakör vagy séma bevezetése, igaz, a feladatbázis feltöltése mindenképpen időigényes munka. A tervben nem tértem ki arra, hogy az adatbázisban a gyakorlás eredményeként mit jelenítünk meg. Ez azért van, mert még az oktatóknak sincs határozott elképzelése arról, hogy kötelező legyen-e a gyakorlás, ill. hogy milyen módon számítson bele a vizsgajegybe. Azt azonban számon tartjuk a terv szerint is, hogy melyik témakörből hány feladatot oldott meg jól egy-egy hallgató, ebből pedig egyértelműen lehet számítani az összeredményt, akármilyen is az algoritmus.

A további két komponens alapvetően adminisztratív feladatokat lát el. Kidolgozásuk nem olyan részletes mint az előző három esetben. Mivel azonban mindkettő az adatbázison végez műveleteket, tervüket az adatbázis tervével együtt kell értékelni. Így vizsgálva őket már finomabban cizellált képet kapunk. A megvalósítandó programok feladatait definiáltam, megvalósítási tanácsokat pedig kivételesen azért nem adtam, mert egy-egy adatbázis-lekérdezés megvalósítása rutinszerű feladatnak tekinthető.

A **teljes ETS** terve az adatbázissal a középpontban véleményem szerint kellően átfogó lett, érinti egy egyetemi tárgy minden adminisztratív feladatát. Az egyes komponensei kellően függetlenek egymástól ahhoz, hogy akár külön-külön is lehessen őket alkalmazni vagy cserélni, módosítani, ugyanakkor az egységes adatbázisnak köszönhetően mégis összeállnak egy közös egésszé. A kitűzött cél azon pontjainak is megfelel, hogy programozási nyelv- és tárgyfüggetlen legyen. Noha a gyakorlatban nem volt lehetőségem kipróbálni a terv minden részletét, úgy érzem, hogy működőképes, jól alkalmazható rendszert alkottam.

A megvalósított komponenseket az őket ismertető fejezetek végén egy-egy szakaszban már külön értékeltem. Itt csak annyit mondanék el, hogy elkészítésükkel bemutattam, hogy a tervek követésével elég programozói oldalról átgondolni a megvalósítás lépéseit, a strukturális részletek már készen állnak. Az elkészült programok működőképessége részben igazolja a terv helyességét is.

7.2. Továbbfejlesztési lehetőségek

A terv kivitelezésekor elsőként az adatbázist kellene megvalósítani, mivel ez köti össze a komponenseket. Erre remélhetőleg hamarosan sor kerül demonstrátorok, önálló laborosok bevonásával. Ezután következhet a többi komponens fokozatos megvalósítása és gyakorlati alkalmazása. Természetesen az egyes komponensek terve is továbbfejleszthető.

Az adatbázis maga bővíthető lenne úgy, hogy több félév adatait legyen képes egyszerre tárolni, az összefüggéseket – pl. közös hallgatók – feltüntetve. A félévvel kapcsolatos számos dokumentum – feladatkiírás, követelményrendszer stb. – vagy egy-egy rájuk mutató hivatkozás is tárolható lehetne benne.

A házi feladatok értékelésénél például érdekes lehet egy névre szóló feladatkiosztó és -ellenőrző rendszer kidolgozása, mert ezzel jelentősen csökkenthető lenne a másolások száma. Ehhez a terveket alaposan át kellene dolgozni, de bizonyos elemek megőrizhetőek. Az ellenőrzést gyorsítani lehetne több számítógépből álló rendszerrel is. Ilyenkor persze szükség lenne egy koordinátor-programra, amely egyenletesen osztja el a munkát a gépek között. Egy-egy levél beérkezésekor becsülni lehetne a várakozási időt is a korábban kapott tesztidők alapján, így a hallgatók nem csak azt tudnák meg, hogy hányan állnak előttük a sorban, hanem arról is tájékoztatást kapnának, hogy ez várhatóan mennyi időt jelent. A becsült idő leteltével újabb értesítésben finomítani lehetne a becslést.

A gyakoroltató rendszer bővíthető lenne például egy ZH-sor szerű teszt összeállításával. Ez a szolgáltatás a meglévő gyakorló feladatokból egy olyan tesztsort állítana össze, amelyet a hallgatók interaktívan kitöltve azonnal megkapnák az „osztályzatot", és így jobban fel tudnák mérni saját tudásszintjüket. El lehetne készíteni a komponens önállóan telepíthető programváltozatát is, amelyet ki lehetne adni a hallgatóknak otthoni gyakorlásra.

Az adatbázis-hozzáférést biztosító komponenst számtalan lekérdezés-fajtával lehetne bővíteni. Különösen igaz ez, ha maga az adatbázis is bővül.

Mindent összevetve megállapítható, hogy a szakdolgozatom nemhogy lezár egy témát, hanem éppen ellenkezőleg, lehetőségek százait nyitja meg. Az álmok valóra váltásához több év munka is kevés lenne, de úgy vélem, hogy a realitásoknál maradva már egy év alatt is jól használható rendszert lehet felépíteni.

Összefoglalás

A cél egy olyan összefüggő, általános rendszer kidolgozása volt, amely hatékonyan támogatja az egyetemi oktatók munkáját, légyen szó bármilyen tárgyról, ezen belül azonban főként a programozási nyelveket oktató tárgyakra kellett koncentrálnom.

Ehhez elsőként megvizsgáltam az oktatást támogató rendszerek világát, és megállapítottam, hogy az összkép igen sokszínű, az alkalmazástól függően egészen különböző rendszerek léteznek.

Ezután bemutattam az ötletadó és esettanulmányként szolgáló *Deklaratív programozás* című tárgyat, és kiemeltem azon tulajdonságait, amelyek számunkra az ETS megtervezése szempontjából érdekessé teszik.

Ezt követően rátértem a már létező, de egységes rendszert nem alkotó programok ismertetésére. Elmondtam, hogy hogyan jöttek létre, milyen feladatokat oldanak meg, és mik a korlátaik.

A dolgozat magját az ETS tervének ismertetése adta. Ebben a fejezetben egy átfogó képtől kezdve közelítettem a részletek felé, az elméleti alapokat lefektetve, praktikus megvalósítási tanácsokat is adva. A tervezés során végig szem előtt tartottam azt az elképzelést, hogy a kapott rendszernek tudnia kell támogatni *bármilyen programozási nyelv*, tágabban *bármilyen tárgy* oktatását.

A kiírásnak megfelelően a rendszer egy komponensének két részletét meg is valósítottam, így vizsgálva meg a terv életképességét. Az eredményt részletesen bemutattam, szót ejtve a megvalósítás hiányosságairól is.

Végezetül az előző fejezet az első szakaszában röviden értékeltem a szakdolgozat eredményeit, rámutatva egy-két hiányosságára, és felhívtam a figyelmet a továbbfejlesztés lehetőségeire is.

Köszönetnyilvánítás

Köszönet illeti elsődlegesen *Szeredi Pétert*, a konzulensemet és a tárgy egyik oktatóját a rengeteg hasznos tanácsért és javaslatért, amellyel a szakdolgozatom megírását támogatta. Ugyanezért jár a köszönet *Hanák Péternek*, a tárgy másik oktatójának és *Benkő Tamás* doktorandusz hallgatónak is.

Köszönöm *Péter Lászlónak*, *Gefferth Andrásnak* és *Rozmán Tamásnak* a házi feladatokat feldolgozó rendszer első változatainak kidolgozását, amely az alapját jelentette az egész ETS rendszernek.

Köszönöm *Berki Lukács Tamásnak* és *Békés András Györgynek*, a 2001-es tavaszi tanév demonstrátorainak a gyakorló rendszer első változatának kidolgozását, ahonnan rengeteg hasznos ötletet leshettem el.

Köszönöm *Lukácsy Gergelynek* a hasonlóságvizsgáló program elkészítését, amely nélkül nem jutott volna eszembe funkcionális nyelven írt programok hívási gráfjának építésével foglalkozni.

Köszönöm *Tarján Péternek*, aki szintén 2001-es tavaszi félévben volt a tárgy demonstrátora, hogy a házi feladatokat ellenőrző rendszer működtetését ebben a minőségében nagyrészt átvállalta tőlem, és így több időt fordíthattam a szakdolgozatom megírására.

A. Függelék

A webes adatbázis-hozzáférés egyes lapjai

Ebben a függelékben az 5.8. fejezetben bemutatott webes adatbázis-hozzáférést biztosító komponens néhány weboldalának látványterve látható.

A.1. ábra. Belépés

Netscape: Egyéni beállítások

File Edit View Go Communicator Help

Egyéni beállítások

Beállítások

kérek Prolog jegyzetet

kérek SML jegyzetet

E-mail cím: dhanak@freemail.hu

feliratkozom a levelezési listára

És/vagy erre a címre kérek értesítést:

az egyes határidők közeledtéről

a házi feladataim teszteredményéről

a házi feladataim végeredményéről

ZH időpontokról és az engem érintő vizsgaidőpont(ok)ról

Mehet

Jelszó megváltoztatása

Új jelszó:

Új jelszó (ismét):

Mehet

DP Administrator

Open the address book

A.2. ábra. Egyéni beállítások

Netscape: Házi feladat beadása

File Edit View Go Communicator Help

Házi feladat beadása

A programod beadásához kérlek add meg a szükséges adatokat! Ha valamelyik állomány **nem áll rendelkezésre**, a megfelelő mezőt hagy kitöltetlenül! Ha valamelyik ponthoz **több állomány tartozik**, zip-pel tömörítsd, és az állomány kiterjesztése legyen `.zip`.

Neptun kód: e6070b

Jelszó: *********

A feladat kategóriája: nagy házi feladat

A Prolog-program: satrak.pl Browse...

Az SML-program: Satrak.sml Browse...

A dokumentáció (text vagy html): doksi.html Browse...

Mehet!

DP Administrator

100%

A.3. ábra. A házi feladat webes beadása

B. Függelék

ZH eredmények listaállományának elkészítése

Az 5.7.1. szakaszban ismertettem a ZH eredmények offline adminisztrálásának egy lehetséges módját. Ugyanitt megemlítettem egy programot, amely az ehhez szükséges állomány előállításához nyújt segítséget. Ebben a függelékben ezt mutatom be.

A „program" a Unixos világban ismert *GNU Emacs* szövegszerkesztő *ELisp* programozási nyelvén készült, amely a LISP egy nyelvjárása. Azért írtam a program szót idézőjelben, mert az Emacs-hoz jár egy forms névre hallgató csomag, amely segítségével kevés programozás árán egyszerű szöveges űrlapokat lehet létrehozni. Az ilyen űrlapok adataikat egy szöveges adatbázisból veszik és oda is írják vissza. Az adatállomány formátuma ugyan rögzített, de definiálhatók beolvasó- és kiíró-filterek, amelyekkel tetszőleges formátumú állomány kezelhetővé tehető. Az űrlap létrehozásához egy rövid ELisp programot kell írni, amely beállítja a paramétereket és definiál bizonyos függvényeket.

A rövid ismertető után következzék a ZH eredmények beírásához készített program forrása.

```
(defun zhform-grep (file str)
  (= 0 (call-process "grep" file nil nil str)))

(setq forms-file (read-file-name "Az adatállomány neve: ")
      zhform-languages '(("prolog" . pl)
                         ("sml" . ml))
      zhform-lang (or
                   (and (zhform-grep forms-file "pl") 'pl)
                   (and (zhform-grep forms-file "ml") 'ml)
                   (cdr (assoc (completing-read
                                "Új állomány! A javított nyelv: "
                                zhform-languages nil t)
                               zhform-languages))
                   'pl)
      zhform-firstex (cdr (assoc zhform-lang '((pl . 1) (ml . 6)))))

(setq forms-number-of-fields
      (forms-enumerate
       '(neptun                ; 1
         p1                    ; 2
         p2                    ; 3
         p3                    ; 4
         p4                    ; 5
         p5)))                 ; 6
```

```
(defun zhform-read-filter ()
  (beginning-of-buffer)
  (while (re-search-forward (format"^%s('\\(.*\\)', \\[\\(.*\\),\\(.*\\),\
\\(.*\\),\\(.*\\),\\(.*\\)\\])\\.$" zhform-lang) nil t)
    (replace-match "\\1\t\\2\t\\3\t\\4\t\\5\t\\6" t)))

(defun zhform-write-filter ()
  (beginning-of-buffer)
  (while (re-search-forward "^\\(.*\\)\t\\(.*\\)\t\\(.*\\)\t\\(.*\\)\t\
\\(.*\\)\t\\(.*\\)$" nil t)
    (replace-match (format "%s('\\1', [\\2,\\3,\\4,\\5,\\6])."
                           zhform-lang) t)))

(defun zhform-new-record (record)
  (aset record neptun "??????")
  (aset record p1 "-")
  (aset record p2 "-")
  (aset record p3 "-")
  (aset record p4 "-")
  (aset record p5 "-")
  record)

(setq forms-read-file-filter  'zhform-read-filter
      forms-write-file-filter 'zhform-write-filter
      forms-new-record-filter 'zhform-new-record)

(setq forms-format-list
      (list
       "=================== ZH eredmények ===================\n\n"
       "PgUp:    előző rekord       PgDn:    következő rekord\n"
       "C-c <:   első rekord        C-c >:   utolsó rekord\n"
       "C-c C-o: rekord beszúrása   C-c C-d: rekord törlése\n"
       "S-TAB:   előző mező         TAB:     következő mező\n"
       "C-c C-x: kilépés            C-h m:   segítség\n\n"
       "===================================================\n\n"
       "   Neptun kód: " neptun "\n"
       (format "  %2d. feladat: " zhform-firstex)       p1 "\n"
       (format "  %2d. feladat: " (+ zhform-firstex 1)) p2 "\n"
       (format "  %2d. feladat: " (+ zhform-firstex 2)) p3 "\n"
       (format "  %2d. feladat: " (+ zhform-firstex 3)) p4 "\n"
       (format "  %2d. feladat: " (+ zhform-firstex 4)) p5))

(setq forms-read-only nil
      forms-forms-scroll t        ; pgup/pgdn = next/prev record
      forms-forms-jump t          ; home/end = first/last record
      forms-insert-after t)       ; always insert *after* current record
```

Ezt töltsük be az Emacs szövegszerkesztőbe és adjuk ki a M-x forms-mode parancsot. Ekkor a program megkérdezi a szerkeszteni kívánt adatállomány nevét. Ha ilyen állomány létezik, felismeri, hogy abban a ZH melyik fele van pontozva (Prolog: 1–5 feladat, vagy SML: 6–10 feladat). Ha az állomány még nem létezik bekéri a nyelv nevét. Végül megjelenik az űrlap, amellyel kényelmesen feltölthetjük az „adatbázist”.

C. Függelék

Egy naplóállomány

Ebben a függelékben egy a 6.1. fejezetben bemutatott *GUTS* program által előállított naplóállományt mutatok meg. A tesztelt program valódi, a 2001-es tavaszi félév egyik beadott házi feladata. A szöveges részek mind a sablonokból jönnek, módosításuk könnyű.

```
                              Tesztnapló
                              ----------

A hallgató neve és azonosítója: LaposElemer.lapose
A program verziószáma:          2
A teszt időpontja:              május 15. 16:43
A tesztet futtató hoszt:        gaia.hanak.hu

A Prolog-program tesztelése
---------------------------

A házi feladat fordítása folyamatban...
{loading /home/guts/hwks/01s/bhw/LaposElemer.lapose/esatrak.ql...}
SICStus 3.8.1 (x86-linux-glibc2.1): Tue Dec 21 16:50:07 CET 1999
A fordítás sikeres volt.

 1. teszteset, időlimit =  10 sec
---------------------------------
{restoring /home/guts/hwks/01s/bhw/LaposElemer.lapose/satrak...}
{Warning: [P] - singleton variables in user:egyezik/2 in lines 14-18}

>>>>> A program lefutott 0.88 sec alatt, a megoldás HELYES.

 2. teszteset, időlimit =  10 sec
---------------------------------
{restoring /home/guts/hwks/01s/bhw/LaposElemer.lapose/satrak...}
{Warning: [P] - singleton variables in user:egyezik/2 in lines 14-18}

>>>>> A program lefutott 0.88 sec alatt, a megoldás HELYES.

 3. teszteset, időlimit =  20 sec
---------------------------------
{restoring /home/guts/hwks/01s/bhw/LaposElemer.lapose/satrak...}
{Warning: [P] - singleton variables in user:egyezik/2 in lines 14-18}

>>>>> A program lefutott 0.88 sec alatt, a megoldás HELYES.
```

```
 4. teszteset, időlimit =  20 sec
---------------------------------
{restoring /home/guts/hwks/01s/bhw/LaposElemer.lapose/satrak...}
{Warning: [P] - singleton variables in user:egyezik/2 in lines 14-18}

>>>>> A program lefutott 0.84 sec alatt, a megoldás HELYES.

 5. teszteset, időlimit =  30 sec
---------------------------------
{restoring /home/guts/hwks/01s/bhw/LaposElemer.lapose/satrak...}
{Warning: [P] - singleton variables in user:egyezik/2 in lines 14-18}

>>>>> A program lefutott 0.89 sec alatt, a megoldás HELYES.

 6. teszteset, időlimit =  30 sec
---------------------------------
{restoring /home/guts/hwks/01s/bhw/LaposElemer.lapose/satrak...}
{Warning: [P] - singleton variables in user:egyezik/2 in lines 14-18}

>>>>> A program lefutott 0.94 sec alatt, a megoldás HELYES.

 7. teszteset, időlimit =  40 sec
---------------------------------
{restoring /home/guts/hwks/01s/bhw/LaposElemer.lapose/satrak...}
{Warning: [P] - singleton variables in user:egyezik/2 in lines 14-18}

>>>>> A program lefutott 2.4 sec alatt, a megoldás HELYES.

 8. teszteset, időlimit =  40 sec
---------------------------------
{restoring /home/guts/hwks/01s/bhw/LaposElemer.lapose/satrak...}
{Warning: [P] - singleton variables in user:egyezik/2 in lines 14-18}

>>>>> A program lefutott 17.07 sec alatt, a megoldás HELYES.

 9. teszteset, időlimit =  50 sec
---------------------------------
{restoring /home/guts/hwks/01s/bhw/LaposElemer.lapose/satrak...}
{Warning: [P] - singleton variables in user:egyezik/2 in lines 14-18}

>>>>> Túllépte az időlimitet.

10. teszteset, időlimit =  50 sec
---------------------------------
{restoring /home/guts/hwks/01s/bhw/LaposElemer.lapose/satrak...}
{Warning: [P] - singleton variables in user:egyezik/2 in lines 14-18}

>>>>> A program lefutott 26.13 sec alatt, a megoldás HELYES.

------------------------
 9 megoldás jó a 10-ből.
------------------------
```

```
================================================================
################################################################
================================================================

Az SML-program tesztelése
-------------------------

A házi feladat fordítása és szerkesztése folyamatban...
/usr/local/lib/mosml-2.0.0/bin/mosmlc -c Satrak.sml
/usr/local/lib/mosml-2.0.0/bin/mosmlc -o ESatrak ESatrak.uo
A fordítás sikeres volt.

 1. teszteset, időlimit =  10 sec
----------------------------------

>>>>> A program lefutott 0.01 sec alatt, a megoldás HELYES.

 2. teszteset, időlimit =  10 sec
----------------------------------

>>>>> A program lefutott 0.01 sec alatt, a megoldás HELYES.

 3. teszteset, időlimit =  20 sec
----------------------------------

>>>>> A program lefutott 0.02 sec alatt, a megoldás HELYES.

 4. teszteset, időlimit =  20 sec
----------------------------------

>>>>> A program lefutott 0.02 sec alatt, a megoldás HELYES.

 5. teszteset, időlimit =  30 sec
----------------------------------

>>>>> A program lefutott 0.03 sec alatt, a megoldás HELYES.

 6. teszteset, időlimit =  30 sec
----------------------------------

>>>>> A program lefutott 0.04 sec alatt, a megoldás HELYES.

 7. teszteset, időlimit =  40 sec
----------------------------------

>>>>> A program lefutott 33.21 sec alatt, a megoldás HELYES.

 8. teszteset, időlimit =  40 sec
----------------------------------

>>>>> A program lefutott 1.9 sec alatt, a megoldás HELYES.
```

```
 9. teszteset, időlimit =  50 sec
---------------------------------

>>>>> A program lefutott 5.4 sec alatt, a megoldás HELYES.

10. teszteset, időlimit =  50 sec
---------------------------------

>>>>> Túllépte az időlimitet.

------------------------
 9 megoldás jó a 10-ből.
------------------------
```

D. Függelék

A funkcionális nyelvek néhány sajátossága

A 6.2. fejezetben a hívási gráfok építésének speciálisan funkcionális nyelveknél adódó problémáit és az ezekre javasolt megoldásaimat ismertetem. A fejezetben foglaltak megértéséhez elengedhetetlen a funkcionális nyelvek néhány fontos tulajdonságának ismerete. Azok számára, akik nem járatosak a funkcionális nyelvekben, egy igen rövid ismertető olvasható ebben függelékben.

A 3.1. fejezetben röviden elmondtam, amit a funkcionális nyelvekről minimálisan tudni érdemes, legfőképpen, hogy a funkcionális nyelvek két fő jellemzője az *érték* és a *függvényalkalmazás*. Azt is megemlítettem, hogy – ellentétben más nyelvekkel – a *függvényérték* is éppen olyan érték, mint a többi, azzal a többlettel, hogy egy függvényértéket *alkalmazhatunk* egy másik értékre: ennek eredményeként egy harmadik értéket kapunk (természetesen ez is lehet függvényérték). A továbbiakban ismertetendő tulajdonságok mind erre a két jellemzőre vezethetők vissza.

D.1. Értékek és nevek

Az értékekhez a funkcionális nyelvekben *nevet* köthetünk. SML-ben ezt az alábbi deklarációval tehetjük meg:

```
- val myName = "Hanák Dávid";
> val myName = "Hanák Dávid" : string
```

Az első sorban a '`-`' jel jelzi, hogy az értelmező bemenetre vár. A `val` kezdetű paranccsal nevet adunk a `"Hanák Dávid"` füzérnek, a rendszer ezt a következő sorral nyugtázza, és jelzi, hogy az érték típusa füzér, azaz `string`. Fontos megérteni, hogy ez nem ugyanaz, mint az imperatív nyelvek *változó*-deklarációi: itt a név és az általa jelölt érték teljesen ekvivalensek egymással. Ha valahol a nevet használjuk, maga az érték épül be a kifejezésbe, nem pedig hivatkozások láncolata jön létre:

```
- val HD = myName;
> val HD = "Hanák Dávid" : string
```

Lehetőség van arra is, hogy a nevet „átdefiniáljuk”, azaz egy újabb értékhez kössük. Ennek az újabb értéknek természetesen a típusa is más lehet.

```
- val myName = pi;
> val myName = 3.14159265359 : real
```

A `pi` is egy név, amely a `Math` könyvtárban van definiálva. Ha most megvizsgáljuk a `HD` név által jelölt értéket, azt tapasztaljuk, hogy az még mindig a `"Hanák Dávid"` füzér, tehát valóban az *értéket* „jegyezte meg”, nem a `myName` nevet:

```
- HD;
> val it = "Hanák Dávid" : string
```

D.2. Függvények „átnevezése”

Mivel a függvényértékek éppen úgy kezelhetők, mint bármely más érték, a következő teljesen szabályos:

```
- val f = cos;
> val f = fn : real -> real
- f pi;
> val it = ~1.0 : real
```

Az első sor hatására az `f` *név* ugyanazt a függvényértéket jelöli, mint a `cos` név (ez utóbbi a `Math` könyvtárban definiált függvény). A második sor annyit jelent, hogy `f` egy olyan függvényt jelöl, amely valósból képez valós értéket. A harmadik sorban az `f` által jelölt függvényértéket alkalmazzuk a már ismert `pi` értékre, az eredményt a negyedik sor mutatja (az SML '~'-vel jelöli a negatív előjelet). A példa alapján is megállapítható, hogy ez a viselkedés teljesen megfelel a matematikában megszokottnak, csupán a „szintaktika” különbözik. (Ott valahogy így fogalmaznánk: „Jelölje f azt a függvényt, amely. . . ”.)

Hasonló dolgot persze imperatív nyelvek esetében is tehetünk: teljesen szabályos egy olyan C függvény definiálása, amelynek visszatérési értéke és argumentumai megegyeznek valamely más függvényével, és amelynek törzse nem áll másból, mint eme másik függvény meghívásából. Például:

```
float   f (float x)   { return sin (x); }
```

A különbség az, hogy ebben az esetben a lefordított változatban egy `call` utasítás helyett kettő van, és a veremben is létrejön egy újabb hívási szint; hacsak a fordító ki nem optimalizálja. A fenti SML-példában bemutatott eszköz ennél hatékonyabb: `f` meghívásakor közvetlenül a megfelelő függvényre adódik a vezérlés. A működés kicsit olyan, mint a C-ben a makrók használata, csak nyelvi szinten megoldott, és emiatt sokkal biztonságosabb.

A helyzetet némiképpen bonyolítja, hogy a már egyszer lekötött nevek átdefiniálhatóak. Ha tehát a fenti példa egyenes folytatásaként a következőt adom be az SML-értelmezőnek:

```
- val f = e;
> val f = 2.71828182846 : real
```

akkor ettől a ponttól kezdve `f` nem a `cos` függvényértéket, hanem az e valós értéket jelöli. Ha megpróbálnánk ismét alkalmazni `f`-et `pi`-re, akkor az SML-értelmező hibát jelezne.

D.3. Anonim függvények

A funkcionális nyelvek alapja a λ-kalkulus (lambda-kalkulus), ennek megfelelően az ún. *lambda-jelöléssel* úgy is definiálhatunk függvényértékeket, hogy nem adunk nekik nevet! (Ezen a módon természetesen nem definiálhatunk rekurzív függvényt, hiszen nem tudunk saját magára hivatkozni.) Ez SML-ben a következőképpen írható le:

```
- (fn x => x*x) 12;
> val it = 144 : int
```

A zárójelben lévő kifejezés azt a függvényértéket jelöli, amely az argumentumát négyzetre emeli. Ha ezt alkalmazzuk a 12 értékre, nem meglepő módon 144-et kapunk. Egy ilyen anonim függvénynek (mivel éppen olyan érték, mint a többi) utólagosan nevet is adhatunk:

```
- val f = fn x => x*x;
> val f = fn : int -> int
```

Voltaképpen ez a függvények definiálásának kanonikus alakja. Az áttekinthetőbb és ezért gyakrabban használt jelölés csak szintaktikus édesítőszer:

```
- fun f x = x*x;
> val f = fn : int -> int
```

D.4. Részlegesen alkalmazható függvények

Az első funkcionális nyelvben, a LISP-ben ez a fogalom még nem létezett. Ott, mint minden mást, egy függvényhívást is egy listával írunk le, amelynek fejében a függvényérték áll, farkában pedig a függvény argumentumai.[1] Természetesen módunkban áll ezt a listát adatként kezelni, és fokozatosan újabb tagokkal (leendő argumentumokkal) bővíteni, végül a megfelelő helyen az erre alkalmas beépített függvénnyel az egészet meghívni. Az SML – és a többi, a LISP-nél modernebb funkcionális nyelv – erre a „fokozatos bővítésre” sokkal elegánsabb és kényelmesebb lehetőséget nyújt.

Az SML-ben szigorúan véve egy függvénynek pontosan egy argumentuma lehet. Ez az argumentum természetesen tetszőlegesen bonyolult SML-kifejezés lehet, leggyakrabban egy ún. *ennes* (*tuple*), amely nem más, mint egy n db rögzített helyű mezőből álló rekord, ha úgy tetszik, felsorolás (önmaga is egyetlen érték). Az SML-ben ezt a következőképpen jelöljük:

```
- val pair = (pi, "pi");
> val pair = (3.14159265359, "pi") : real * string
```

Itt `pair` egy olyan párt jelöl, amelynek első tagja a π közelítése, második tagja pedig egy füzér. Az ilyen ennesek segítségével lehet egyféleképpen megoldani, hogy egy függvény több argumentumot kapjon. Egy másik lehetőség az ún. *részlegesen alkalmazható függvények* írása. Vegyük például a következő függvénydefiníciót:

```
- fun add a b = a+b;
> val add = fn : int -> int -> int
- add 2 3;
> val it = 5 : int
```

Pontatlanul megfogalmazva mondhatjuk, hogy az `add` függvénynek két argumentuma van (és a továbbiakban is ezt a fordulatot fogom használni), amelyeket összead, az alkalmazásából ez jól látszik. Valójában azonban a következőről van szó: az `add` egy részlegesen alkalmazható függvény, amelynek első (pontosabban egyetlen) argumentumát meghatározva (lekötve) egy *függvényértéket* kapunk eredményül, amelyet egy újabb (ismét egyetlen) argumentumra alkalmazva megkapjuk a végeredményt. Talán könnyebb megérteni, miről is van szó, ha a példa folytatásával rávilágítok a lényegre:

```
- val addTwo = add 2;
> val addTwo = fn : int -> int
- addTwo 3;
> val it = 5 : int
- addTwo 134;
> val it = 136 : int
```

Mi is történt itt? Az első sor hatására `addTwo` azt a függvényt jelöli, amely az argumentumához kettőt ad. Ha ezt háromra alkalmazzuk, ötöt kapunk, ha 134-re, akkor 136-ot. Ha jobban megvizsgáljuk a helyzetet, azt tapasztaljuk, hogy az `add` név valójában nem is egy, hanem *két* függvényértéket takar, sőt voltaképpen nagyon sokat, hiszen a részleges alkalmazás eredményeként kapott függvényérték függ `add` első argumentumától.

Előfordulhatnak olyan esetek is, hogy nem állapítható meg egy függvényről, hogy hányszorosan alkalmazható részlegesen. Pontosabban szólva az argumentumainak száma függhet maguktól az argumentumoktól is. Erre példát és némi magyarázatot a D.6. szakaszban láthatunk.

D.5. Függvényérték mint argumentum

Mivel a függvényérték is érték, semmi akadálya nincs, hogy egy másik függvény argumentumaként szerepeljen. Azokat a függvényeket, amelyek argumentumként egy függvényt várnak, *magasabb rendű*

[1] Innen a LISP neve: *LISt Processing*, azaz lista-feldolgozás.

függvényeknek nevezik. Ilyen például a könyvtári `map` függvény, amely két argumentumot vár, egy függvényt és egy listát. Az eredményül adott lista hossza megegyezik a második argumentumban átadottéval, és minden eleme olyan érték, amelyet úgy kapunk, hogy az eredeti lista megfelelő elemére alkalmazzuk az argumentumként átadott függvényt. Ezt szemlélteti a következő példa:

```
- map (fn x => x+1) [1,4,9];
> val it = [2, 5, 10] : int list
```

D.6. Függvényérték mint visszatérési érték

Arról már volt szó, hogy ha egy függvényt részlegesen alkalmazunk, akkor a visszatérési értéke (eredménye) egy újabb függvény. Ilyen helyzet azonban úgy is előfordulhat, hogy az adott függvény összes argumentumát meghatározzuk. Vegyük a következő példát:

```
- val f = nth([sin,ln,sqrt], 2);
> val f = fn : real -> real
- f e;
> val it = 1.6487212707 : real
```

Mi is történt itt? Az `nth` függvény nem tesz mást, mint visszaadja az első argumentumában átadott lista n-edik elemét, ahol n a második argumentumban megadott érték. Csakhogy itt most olyan listáról van szó, amelynek elemei a `Math` könyvtár bizonyos függvényei. (Mivel a függvényérték is érték, ennek nincs semmi akadálya.) Az `f` érték tehát maga is függvényérték, amelyet alkalmazhatunk egy valós értékre; az így kapott eredmény történetesen e négyzetgyöke.

D.7. Lokális kifejezés és deklaráció

Ahogy az imperatív nyelvekben is van lehetőség lokális változók létrehozására és használatára, úgy a funkcionális nyelvek is adnak eszközöket lokális *érték-deklarációk* használatára. Az értéket eredményező kifejezésnek (pl. `pi/2.0`) és a deklarációnak (pl. `val two = 2`) is megvan a lokális érték-deklarációval kiegészített változata. Az előbbit *lokális kifejezésnek*, az utóbbit *lokális deklarációnak* nevezik. Lássunk mindkettőre egy-egy példát:

```
- let val halfPi = pi/2.0 in halfPi end;
> val it = 1.57079632679 : real
- local fun half x = x/2.0 in val halfPi = half pi end;
> val halfPi = 1.57079632679 : real
```

Az első egy lokális kifejezés, deklarálja a `halfPi` nevet, amit aztán a kifejezésben felhasznál. A lokális kifejezés értéke természetesen $\pi/2$, a `halfPi` név azonban érvényét veszti az `end` kulcsszó után. A második egy lokális deklaráció, amely deklarálja a `half` függvényt, amelyet aztán felhasznál a `halfPi` név deklarálásakor. Az `end` kulcsszó után a `half` függvény nem használható, de a `halfPi` név igen.[2]

D.8. Infix jelölés

Az SML lehetőséget ad arra, hogy egyes kétoperandusú függvényeket infix alakban használjunk, ha ezt kifejezőbbnek érezzük. Ehhez deklarálni kell, hogy az adott operátor – így nevezzük az infix pozíciójú függvényeket – jobbra vagy balra köt-e, ill. hogy mi a precedenciája. Erre szolgál az `infix` és az `infixr` kulcsszó. Ha az infix deklaráció a függvény definiálása *előtt* helyezkedik el, akkor maga a definíció is infix formájú kell, hogy legyen. Példaként nézzük meg a bináris léptető operátor definícióját!

[2] Sajnos az elnevezések kicsit megtévesztőek, hiszen nem maga a kifejezés vagy a deklaráció lokális, hanem a bennük felhasznált értékek deklarációja.

```
- infixr 8 <<;
> infixr 8 <<
- fun a << 0 = a
    | a << b = if b > 0 then (2*a) << (b-1)
               else raise Subscript;
> val << = fn : int * int -> int
```

A 8-as precedenciaszint a szorzásnál is erősebben kötővé teszi az operátort (ezért a zárójel a definícióban), az `infixr` kulcsszó pedig azt jelzi, hogy jobbra köt. A feltétel `else` ága hibajelzésre szolgál arra az esetre, ha az operandus negatív lenne.

Az infixként deklarált operátorok továbbra is használhatóak prefix alakban, ha az `op` kulcsszóval előzzük meg őket, pl. az `op«(3,2)` jelölés teljesen ekvivalens a `3 « 2` alakkal. Az `op` kulcsszó használatával azt is megtehetjük, hogy először infixnek deklaráljuk az operátort, majd az ezt követő függvénydefiníciót mégis prefix alakban adjuk meg.

Az infix deklarációk hatását véglegesen is megszüntethetjük a `nonfix` kulcsszó használatával, ez azonban kerülendő, mert könnyen zavart okozhat.

Irodalomjegyzék

Cumming, A. (1999), *A Gentle Introduction to ML*, Napier University, Edinburgh. Web address: http://www.dcs.napier.ac.uk/course-notes/sml/manual.html.

Hanák, P. (2001), 'Deklaratív programozás – bevezetés a funkcionális programozásba', oktatási segédlet, BME Irányítástechnika és Informatika Tanszék.

Hanák, P., Szeredi, P., Benkő, T. & Hanák, D. (2001), *Magad, uram, ha szolgád nincs!* NetworkShop'2001 konferencia.

Lukácsy, G. (2000), Forráskódú programok hasonlóságvizsgálata, Országos Tudományos Diákköri Konferencia 2000.

Országh, L., ed. (1992), *Angol-magyar kéziszótár*, 12th edn, Akadémiai Kiadó, Budapest.

Patel, A. & Kinshuk (1997), Intelligent tutoring tools on the internet – extending the scope of distance education, *in* 'International Conference on Distance Education – ICDE 97', State College, PA, USA.

Stern, M. K. (1997), 'Web-based intelligent tutors derived from lecture based courses', Dissertation Proposal.

Szeredi, P. & Benkő, T. (2001), 'Deklaratív programozás – bevezetés a logikai programozásba', oktatási segédlet, BME Számítástudományi és Információelméleti Tanszék.