



Eötvös Loránd Tudományegyetem
Informatikai Kar
Programozási Nyelvek és Fordítóprogramok Tanszék

Beágyazott logikai programozási nyelv tervezése és implementációja használhatósági elvek alapján

Tudományos diákköri dolgozat

Témavezető
Kitlei Róbert
tanársegéd

Hallgató
Kalló Bernát
Programtervező informatikus BSc

Budapest, 2013

Tartalomjegyzék

1. Bevezetés	2
1.1. A Ruby programozási nyelv	2
1.2. Logikai programozás	3
1.3. Logikai paradigmájú vendégnyelvek	4
1.4. Ruby nyelvbe ágyazott logikai nyelvek	5
2. A Rubylog nyelv tervezése	7
2.1. Adattípusok	8
2.1.1. Változók	9
2.2. Logikai kifejezések és állítások szintaxisa	9
2.2.1. Struktúrák	9
2.2.2. Tények és szabályok	10
2.3. Az átjárhatóság biztosítása	10
2.3.1. Kérdések	10
2.3.2. Gazdanyelvi predikátumok	11
2.3.3. Gazdanyelvi függvények	12
2.3.4. Változók behelyettesítése	12
2.4. Kvantor jellegű predikátumok	13
2.4.1. Prefix jelölés	13
2.5. Listák kezelése	14
2.5.1. Parciális listák	14
2.5.2. Általánosított parciális listák	15
2.6. Parciális sztringek	16
3. Az eredmény értékelése	18
3.1. Az érthetőség értékelése	18
3.1.1. Saját kísérletem	18
3.2. Konklúzió	19
3.2.1. Az olvashatóság haszna	19
3.2.2. Továbbfejlesztési lehetőségek	20
Irodalomjegyzék	22

1. fejezet

Bevezetés

A Prolog és a Ruby programozási nyelvek sok szempontból különlegesek a programozási nyelvek között. A Prolog nyelv a logikai programozási paradigmát követi, ahol egy program logikai állításokból áll, és ezekből következtetéseket lehet levezetni. A Ruby nyelv egy objektum-orientált, imperatív, de funkcionális elemeket is tartalmazó programozási nyelv, amelyet azzal a törekvéssel fejlesztettek ki, hogy minél könnyebben lehessen vele dolgozni.

Dolgozatomban megvizsgálom, hogy hogyan lehet a Ruby és a Prolog nyelvek előnyös tulajdonságait egyesíteni. Bemutatom az általam kifejlesztett *Rubylog* beágyazott nyelvet, amely lehetővé teszi a mind a Ruby, mind a Prolog nyelv módszereinek használatát akár egyetlen programrészen belül. Bemutatom, hogy milyen újításokat vittem a Rubylog nyelvbe, amelyekkel bizonyos algoritmusokat mind a Ruby, mind a Prolog nyelvnél tömörebben és átláthatóbban lehet leírni.

1.1. A Ruby programozási nyelv

A Ruby programozási nyelvet 1993-ban kezdte fejleszteni egy japán programozó, Yukihiro Matsumoto. A nyelv fejlesztésénél arra törekedett, hogy minél könnyebben lehessen vele fejleszteni, a programkódot természetesnek érezze a programozó, és élvezze a programozást [6]. A fő irányelve a „legkisebb meglepetés elve” volt, ami azt jelenti, hogy a (Ruby nyelvet jól ismerő) programozót a lehető legkevésbé ériék kellemetlen meglepetések [9].

Röviden megemlítem a Ruby nyelv néhány érdekes tulajdonságát. Ruby nyelvben minden adat objektum, és lehetnek metódusai. Például a számok `abs` metódusa visszaadja az abszolút értéküket:

```
1 -3.abs # => 3
```

A metódusok neve végződik kérdőjellel vagy felkiáltójellel, tipikusan kérdőjellel az igaz/hamis értéket visszaadó metódusok neve végződik.

```
1 0.zero? # => true
```

A Ruby nyelv egyik fontos eleme a **blokk**. Minden metódusnak át lehet adni egy blokkot, ami egy Ruby kódrészlet `do..end` vagy `{..}` jelek között. A metódus tetszőlegesen meghívhatja a blokkot. Például az alábbi program 10-szer írja ki a „Helló” szöveget:

```
1 10.times { puts "Helló" }
```

1.2. Logikai programozás

A logikai programozási módszertant és a Prolog logikai programozási nyelvet 1972-ben kezdte fejleszteni Alain Colmerauer. A logikai paradigma lényege, hogy egy egyszerű logikai bizonyító rendszer segítségével lehet programokat írni. Egy logikai programra kétféleképpen lehet tekinteni. Egyrészt tekinthetjük úgy, mint logikai állítások halmaza, amelyek alapján egy állítás bizonyítását szeretnénk megkeresni. Másrészt tekinthetjük úgy, mint utasítások sorozata, amelyek néhány speciális vezérlő lehetőséget felhasználnak: a sikeres és sikertelen végrehajtást, az egyesítést és a visszalépést. Mindezeket az 1.1. kódrészletben szereplő példaprogramon mutatom be, amelynek célja egy családfában megkeresni az összes nagyszülő-unoka kapcsolatot.

```
1 parent('Bob', 'Mary').
2 parent('Bob', 'Greg').
3 parent('Greg', 'John').
4 parent('Greg', 'Jane').
5
6 grandparent(A,B) if parent(A,X) and parent(X,B).
7
8 main if grandparent(A,B) and
9       write(B) and write(' nagyszuloje ') and write(A) and nl and
10      fail
11      or true.
```

1.1. kódrészlet: Családfa Prolog nyelven

A logikai program alapvető építőkövei a **logikai kifejezések**, amelyek értéke igaz vagy hamis lehet. A logikai kifejezések tartalmazhatnak **szimbólumokat** (pl. 'Bob'), **változókat** (nagybetűvel kezdődő azonosítók, pl. A), és számokat (pl. 3.141), ill. ezeket **struktúrákba** lehet rendezni (pl. parent('Bob', 'Greg')). A struktúrának van egy neve (parent) és egy vagy több argumentuma ('Bob' és 'Greg').

A logikai program mondatokból áll, melyeket pont zár le. Többféle mondat létezik, ebben a példában minden mondat **állítás**. Kétféle állítás létezik, a tény és a szabály. Az 1–4. sorokban négy **tényt** láthatunk, melyek szülő-gyermek kapcsolatokat mondanak ki (pl. Bob szülője Mary-nek). A tény egyetlen logikai kifejezésről kimondja, hogy igaz. A 6. sorban egy **szabály** van, amely definiálja a nagyszülő-unoka kapcsolatot: Egy A személy nagyszülője B-nek, ha A szülője valamely X-nek, és X szülője B-nek. Általában egy szabály egy logikai kifejezésről (a **fejrészből**) kimondja, hogy akkor igaz, ha egy másik logikai kifejezés (a **törzsrész**) igaz. Az azonos nevű és argumentumszámú állítások együtt egy-egy **predikátumot** alkotnak. A példában három predikátum van definiálva, a két argumentumú (bináris) parent, a szintén bináris grandparent és az argumentum nélküli (nulláris) main.

A 6. sort nem csak definícióként, hanem utasításként is fel lehet fogni: azt mondja ki, hogy úgy kell megkeresni egy nagyszülő-unoka kapcsolatot, hogy először keresünk egy szülő-gyermek kapcsolatot (pl. Bob–Mary), majd ehhez keresünk egy másikat. Amennyiben a második kapcsolat keresése sikertelen (Mary-nek nincs gyermeke), a program **visszalép** a legutóbbi helyre, ahol még több lehetőség volt (Bobnak még van gyermeke, Greg), és újra próbálja a második keresést (Gregnek van-e gyermeke) és ha ez sikeres (megtalálja Johnt), akkor a grandparent végrehajtása **sikeres**, az eredménye az {A=Greg, B=John} **változó-hozzárendelés**.

A 8–11. sorok egyetlen mondatot tartalmaznak, mely szintén egy szabály. Ezt utasításként célszerű értelmezni: A főprogramot úgy kell végrehajtani, hogy keresünk egy nagyszülő-unoka kapcsolatot, majd kiírjuk az unokát, a „nagyszülője” szót, a nagyszülőt, majd új sort kezdünk az nl utasítás segítségével. A 10–11. sorokban egy olyan módszer látható, amelyet gyakran használnak

logikai programokban az összes lehetséges megoldás megkeresésére. A `fail` parancs következtében a rendszer **sikertelennek** tekinti a jelenlegi végrehajtást, és visszalép a legutóbbi olyan helyre, ahol még volt több lehetőség (esetünkben ez a `grandparent` definíciójában lévő második `parent` parancsnál van, amikor `A=Bob` és `X=Greg`, és Greg-nek még hátravan egy gyermeke, Jane). Ekkor az új nagyszülő-unoka kapcsolatot (Bob–Jane) is az előzőhöz hasonlóan kiírjuk. A `fail` minden egyes kiértékelése után újra visszatér a program a `grandparent` definíciójához egészen addig, amíg talál nagyszülő-unoka kapcsolatokat. Ha nincs több kapcsolat (esetünkben nincs), akkor a `grandparent` kiértékelése sikertelen lesz. Ekkor a program a 11. sorban lévő `true` szóra lép, aminek következtében a `main` végrehajtása sikeres lesz.

```

1 | ?- parent('Mary', 'John').
2
3 no
4 | ?- parent('Greg',X).
5
6 X = 'John' ? ;
7
8 X = 'Jane'
9
10 yes
11 | ?- main.
12 John nagyszuloje Bob
13 Jane nagyszuloje Bob
14
15 yes

```

1.2. kódrészlet: A Családfa program futtatása

Az 1.2. kódrészletben láthatjuk a program futtatását: a `?-` jel után írhatjuk be a **kérdésünket**, amit a program megpróbál **bizonyítani**. Az 1. sorban lévő kérdéssel megkérdezzük, hogy Mary szülője-e Johnnak. A válasz „no”, vagyis nem. A 4. sorban Greg gyermekeit keressük meg, a rendszer kiírja az első választ (az `X=Greg` változó-hozzárendelést, 6. sor), majd ha egy pontosvesszőt beírunk, a következő választ is (8. sor). A 11. sorban lévő kérdéssel a `main` predikátumot futtatjuk le. A program a megfelelő módon kiírja a nagyszülő-unoka kapcsolatokat. A 15. sorban a „yes” azt jelenti, hogy a program végrehajtása sikeres volt.

A Prolog program végrehajtásának egyik alapvető lépése az **egyesítés**. Ez azt jelenti, hogy egy kifejezést egyenlővé teszünk egy másikkal úgy, hogy a bennük szereplő változóknak értéket adunk. Egyesítés történik akkor, amikor egy predikátumot meghívunk valamilyen paraméterekkel, például a `parent('Greg', X)` kérdés bizonyítása során egyesítjük a `parent('Bob', 'Mary')` ténnyel (ez sikertelen lesz), a `parent('Bob', 'Greg')` ténnyel (ez szintén sikertelen lesz), a `parent('Greg', 'John')` ténnyel (ekkor `X=John` eredménnyel sikeres lesz) és a `parent('Greg', 'Jane')` ténnyel (ekkor `X=Jane` eredménnyel sikeres lesz).

A példaprogramban a logikai kifejezésekben az `if`, `and` és `or` operátorokat használtam a Turbo Prolog nyelv alapján, hogy a Prolog nyelvet nem ismerő olvasó is könnyebben megértse [19]. A legtöbb Prolog rendszer az ISO Prolog dialektust követi, amelyben `if` helyett `:-` jelet, `and` helyett vesszőt és `or` helyett pontosvesszőt használnak [21].

1.3. Logikai paradigmájú vendégnyelvek

A logikai programozási módszereket kiválóan lehet alkalmazni számos területen, például mesterséges intelligencia programozására, adatbázis-kezeléshez, döntéstámogatáshoz, természe-

tes nyelvek értelmezéséhez, stb. [22]. Ugyanakkor van számos olyan feladat, amelyre a logikai programozás kevésbé alkalmas, ilyen például a felhasználói felületek készítése.

A logikai programozási nyelvek nehézségeinek megkerülésére egy gyakori megoldás, hogy a programunk fő részét egy hagyományos imperatív programozási nyelven készítjük el (a **gazdanyelven**), míg bizonyos algoritmusokra egy logikai programozási nyelvet használunk (a **vendégnyelvet**). Ennek a módszernek előnye, hogy az egyes részfeladatokra a nekik jobban megfelelő programozási paradigmát használhatjuk. További előnye, hogy a felhasznált vendégnyelvnek nem szükséges általános célú programozási nyelvnek lennie, hanem lehet egy speciálisan ilyen célra készült **szakterület-specifikus nyelv** (*domain specific language*, DSL). Egy vendégnyelv használatának hátrányai is vannak. Egyrészt két eltérő nyelvet is alaposan meg kell ismernie a programozónak. Másrészt meg kell oldani a két nyelven írt programrészek között az **átjárhatóságot**, vagyis az adatok és a vezérlés átadását, ami sokféle nehézséget okozhat.

Vendégnyelvek szintaxisának megvalósítására két módszer elterjedt: vagy **különálló nyelv-ként** valósítják meg, saját szintaxissal és fordítóprogrammal; vagy pedig a gazdanyelven belüli **beágyazott nyelv-ként**, a gazdanyelv szintaxisát és fordítóprogramját felhasználva [4]. Beágyazott nyelvek tervezésénél bizonyos kihívást jelent az, hogy csak a gazdanyelv szintaktikai eszközeit lehet felhasználni. Ugyanakkor új ötleteket is adhat, lásd 2.5, 2.6.

Ha logikai vendégnyelvet szeretnénk használni, többféle megvalósítási módszer közül is választhatunk. Egyik lehetőség egy általános célú Prolog rendszer felhasználása (pl. [5]). Egy másik gyakori megoldás egy olyan, a gazdanyelven írt értelmezőprogram használata, amely egy különálló, szakterület-specifikus logikai nyelvet értelmez [7, 8, 13, 17]. Létezik olyan fordítóprogram, ami egy különálló logikai nyelvet át tud fordítani a gazdanyelvre [18]. Léteznek olyan megoldások is, amelyek beágyazott logikai nyelveket biztosítanak, ilyen például C++ nyelvhez a *Castor* könyvtár [1], Scheme nyelvhez a *Schelog* [10] és *Kanren* [3] könyvtárak, Scala nyelvhez a *Prolog-in-Scala* könyvtár [11], Python nyelvhez a *PyLog* könyvtár [12].

1.4. Ruby nyelvbe ágyazott logikai nyelvek

A Ruby nyelvhez két olyan könyvtárat találtam, ami beágyazott logikai vendégnyelvet biztosít: a *Rulog* és a *Ruby-Prolog* (korábban *Tiny-Prolog*) könyvtárakat [16, 14]. Most a Ruby-Prolog könyvtárról fogok részletesebben beszélni. Az 1.3. példaprogramban az 1.2. szakaszban bemutatott program megfelelőjét láthatjuk Ruby-Prolog nyelven.

A Ruby-Prolog a Prolog nyelvnek csak egy szűk részhalmazát valósítja meg. A szabályok törzsrészét tömbként kell megadni (10. és 16. sor), amelynek elemeit „és” művelet kapcsolja össze. Ez a szintaxis másmilyen műveleteket nem is enged meg. A nyelv tervezője arra törekedett, hogy a Ruby-Prolog szintaxisa hasonlítson a Edinburgh Prolog szintaxisára (a <=> jel az Edinburgh Prolog-beli :- jel megfelelője, és az „és” műveletet az Edinburgh Prolog dialektushoz hasonlóan vessző jelöli)

Egy Ruby-programozó, Jamis Buck szerint a Ruby-Prolog szintaxisa a Ruby nyelv szintaxisához képest nehezen olvasható [20]. Javasolt egy olyan szintaxist, amely szerinte jobban olvasható. Az ötlete nekem is megtetszett, és ez alapján elkezdtem fejleszteni a saját beágyazott logikai nyelvemet, a Rubylog nyelvet.

```

1 require 'ruby-prolog'
2
3 c = RubyProlog::Core.new
4 c.instance_eval do
5   parent['Bob', 'Greg'].fact
6   parent['Bob', 'Mary'].fact
7   parent['Greg', 'John'].fact
8   parent['Greg', 'Jane'].fact
9
10  grandparent[:A, :B] <=<= [ parent[:A, :X], parent[:X, :B] ]
11
12  main[] <=<= [ grandparent[:A, :B],
13    write[:A], write[" unokája "], write[:B], write["\n"],
14    fail_[] ]
15  main[] <=<= [ ]
16
17  query(main[])
18 end

```

1.3. kódrészlet: Családfa program Ruby-Prolog nyelven

2. fejezet

A Rubylog nyelv tervezése

A Rubylog nyelvet 2011 őszén kezdtem el fejleszteni, Jamis Buck ötlete alapján. Az olvashatósági irányelvet és néhány ötletet tőle vettem át, de nagyon sok saját ötletet adtam hozzá, amellyel a Rubylog egy nagy kifejezőerejű, könnyen olvasható és jól használható logikai nyelv lett. Példaképpen bemutatom a Családfa program Rubylog nyelvbeli megfelelőjét (2.1. példaprogram).

A Rubylog nyelv tervezése során többféle irányelvet is figyelembe vettem. A fő törekvésem az volt, hogy egy jól használható nyelvet készítsek, amivel gyorsan lehet fejleszteni. A célközönségem azok a Ruby felhasználók, akik logikai programozási eszközöket szeretnének használni. Ezért arra törekedtem, hogy a Ruby nyelv előnyös tulajdonságaiból minél többet megőrizzek a Rubylog nyelvben is. Az alábbi tulajdonságokat tartottam különösen fontosnak:

Tömörség Tömörség alatt azt értem, hogy egy adott programot minél kevesebb nyelvi elemmel (vagy minél kevesebb karakterrel) lehessen leírni. A Ruby nyelv ebből a szempontból különösen jó [2]. A tömörséget azért tartom fontosnak, mert kevesebb kódot rövidebb időbe telik megírni, elolvasni és karbantartani. Azonban a nagyon tömör kód akár nehezen átláthatóvá is válhat, ezért fontos szempont az olvashatóság is.

Olvashatóság Olvashatóság alatt azt értem, hogy a programkódot elolvasva minél hamarabb meg lehessen érteni a program működését. Ez nagyon sok tényezőtől függ, a program tagolásától, a térközök elhelyezésétől, a változónevek megválasztásától, a megfelelő mennyiségű és típusú megjegyzésektől, a programozó előismereteitől stb.

A Ruby nyelv valamivel többet nyújt ebben a tekintetben a legtöbb programozási nyelvénél. A Ruby nyelv kialakításánál Matsumoto arra törekedett, hogy a nyelvet természetesnek érezze a programozó, tehát a gondolkodás és a természetes nyelv (konkrétan az angol

```
1 predicate_for String, ".parent_of() .grandparent_of()"
2
3 'Bob'.parent_of! 'Mary'
4 'Bob'.parent_of! 'Greg'
5 'Greg'.parent_of! 'John'
6 'Greg'.parent_of! 'Jane'
7
8 A.grandparent_of(B).if A.parent_of(X).and X.parent_of(B)
9
10 A.grandparent_of(B).each do
11   puts "#{A} unokaja #{B}"
12 end
```

2.1. kódrészlet: Családfa program Rubylog nyelven

nyelv) logikájához, szórendjéhez jól illeszkedjen. Sokszor a programkód szavait sorban kiolvasva azok egy angol mondatná állnak össze, és ez nagy segítség lehet a programkód értelmezésében. A Ruby nyelvnek ezt a jó tulajdonságát igyekeztem a Rubylog nyelvben is megvalósítani. A továbbiakban ezt az elvet „*reads like English*” elvnek fogom nevezni.

Legkisebb meglepetés elve Ez az elv azt mondja ki, hogy a nyelv ne okozzon kellemetlen meglepetést a programozónak. Ezt úgy értelmezem, hogy amennyiben a nyelvben szereplő funkciók szintaxisa között vannak analógiák, akkor a viselkedésük között is legyen analógia. Ez alapján a programozó akár ki is tudja következtetni egy-egy nyelvi elem jelentését, és jó, hogyha valóban úgy is működik, ahogyan elvárja. Természetesen ez nagyon szubjektív dolog, és nagyban függ attól, hogy ki a programozó.

A legkisebb meglepetés elvét a következő példával szemléltetem. A Ruby nyelvben a sztringek `sub` metódusa kicseréli egy részsstring első előfordulását egy másik sztringre, és visszaadja az eredményt. A `sub!` metódus hasonlóképpen kicseréli egy részsstring első előfordulását egy másik sztringre, de ezt destruktívan teszi, vagyis az eredeti sztring módosításával. A `gsub` metódus a `sub` metódushoz hasonlóan egy részsstringet cserél ki egy másik sztringre, de ő a részsstring minden előfordulását kicseréli. A programozó ezek alapján feltételezheti, hogy létezik egy `gsub!` metódus, mely a részsstring minden előfordulását kicseréli destruktívan. Valóban létezik is ilyen metódus. Továbbá, a `sub` metódus elfogad reguláris kifejezést is, ezért elvárhatja a programozó, hogy a `sub!`, `gsub` és `gsub!` metódusok is elfogadjanak.

A legkisebb meglepetés elve hozzásegít ahhoz, hogy kevesebb esetben kelljen a programozási nyelv kézikönyvéhez fordulni ahhoz, hogy programokat tudjunk írni az adott nyelven, vagy azokat értelmezni tudjuk.

Ezekén kívül fontosnak tartom az **átjárhatóságot** a gazdanyelv és a vendégnyelv között. Ahhoz, hogy a beágyazott nyelvvel való munka gördülékeny legyen, fontos, hogy egyszerűen lehessen a gazdanyelv és a beágyazott nyelv között adatokat átadni mindkét irányban.

Arra törekedtem, hogy a Prolog nyelv minden olyan elemét megvalósítsam Rubylog nyelvben, ami szükséges ahhoz, hogy tetszőlegs Prolog nyelven írt programot könnyen át lehessen írni Rubylog nyelvbe. Ugyanakkor arra is ügyeltem, hogy a Prolog nyelv azon funkcióit, melyeket a Ruby nyelven könnyen meg lehet valósítani, ne implementáljam Rubylog nyelven. Például a Prolog nyelv bemeneti/kimeneti predikátumainak nem hoztam létre a Rubylog nyelvben a megfelelőit, mert ezeket a funkciókat Ruby nyelven, gazdanyelvi predikátumok segítségével sokkal könnyebben meg lehet valósítani (lásd [2.3.2](#)).

A továbbiakban részletesen bemutatom azokat az újításokat, amelyeket a Rubylog nyelvbe bevezettem. Bemutatom, hogy milyen akadályokat jelentett egy beágyazott nyelv megvalósítása, hogy ezeket hogyan küzdöttem le, és hogyan segítettek abban, hogy a Rubylog nyelvbe a Prolog nyelv lehetőségeit meghaladó újítások is kerültek.

2.1. Adattípusok

A könnyű átjárhatóság és a sokoldalú felhasználás érdekében arra törekedtem, hogy minden Ruby nyelvbeli adatot lehessen használni Rubylog programokban is. A Ruby-Prolog nyelv ezt nem teljesíti, mert a szimbólumokat (`Symbol`) változóként felhasználja. Én úgy döntöttem, hogy a Rubylog értelmezőben mind a változókat, mind a struktúrákat saját osztályok példányaiként reprezentálom, így a beépített adattípusok használatát nem zavarom.

2.1.1. Változók

Prolog nyelvben a nagybetűvel (vagy aláhúzásjellel) kezdődő azonosítók jelölik a változókat (A), a kisbetűvel kezdődők pedig a szimbólumokat (parent). Érdekes módon a Ruby nyelv is megkülönbözteti a kis- és nagybetűvel kezdődő azonosítókat, csak itt a nagybetűvel kezdődő azonosítók konstansokat jelölnek (Class), a kisbetűvel kezdődők pedig változókat (x) vagy metódushívásokat (puts). Továbbá a Ruby nyelv is tartalmaz szimbólumokat, ezeket kettősponttal kezdve lehet jelölni (:abcd).

A Ruby-Prolog nyelvben a változókat a Ruby nyelv szimbólumaival jelölik (:A). A Rubylog nyelvben meg akartam oldani, hogy a Ruby nyelv konstansai jelöljék a változókat, így kevesebb billentyűlévétéssel, és a Prolog nyelvhez még inkább hasonló módon lehessen leírni a változókat (A). Ezt meg is oldottam a Ruby nyelv `const_missing` eseménykezelőjével, amely nemlétező konstans használata esetén fut le.

A Prolog nyelvben létezik egy ún. névtelen változó, ami egyetlen aláhúzásjelből (_) áll. Ezt akkor használják, ha egy értéket nem szeretnének felhasználni. Minden előfordulásának lehet más-más az értéke. A Rubylog nyelvben megvalósítottam ennek a megfelelőjét, azonban a jó olvashatóság érdekében itt az ANY nevű változó a névtelen változó.

2.2. Logikai kifejezések és állítások szintaxisa

Mivel a Rubylog nyelvet beágyazott nyelvként valósítottam meg, a logikai kifejezéseket a Ruby nyelvbéli kifejezéseként kell leírni. Ezek a kifejezések azonban nem igaz vagy hamis értéket adnak vissza, hanem magát a logikai kifejezést reprezentáló objektumot.

2.2.1. Struktúrák

Jamis Buck az alábbi szintaxist javasolta a struktúrák leírására [20]:

```
1 'Bob'.parent_of('Mary')
```

A szabályok létrehozására pedig:

```
1 A.grandparent_of(B).if(A.parent_of(X).and(X.parent_of(B)))
```

Bár ez a szintaxis jól követi a *reads like English* elvet, nehézséget okozhat a pontok és zárójelek megfelelő elhelyezése. Megpróbáltam a Turbo Prolog szintaxisát jobban megközelíteni, a Ruby nyelv `and` és `or` operátorait és az `if` módosító utasítást felhasználva, az alábbi módon:

```
1 A.grandparent_of(B) if A.parent_of(X) and X.parent_of(B)
```

Ennek a megoldásnak azonban több hátránya is volt. Egyrészt a Ruby nyelvben az `and` és `or` operátorok lusta kiértékelésűek, vagyis nem feltétlenül értékelik ki mindkét operandusukat. Ez nagyon megnehezítette volna a kód értelmezését. A másik probléma, hogy ezzel a szintaxissal nem lehet újfajta operátorokat bevezetni, csak a Ruby nyelvben meglévő operátorokat felhasználni. Mivel fontosnak tartottam, hogy a Rubylog nyelvet új logikai operátorokkal lehessen bővíteni, ezt a szintaxist elvettem.

Úgy találtam tehát, hogy a leginkább megfelelő szintaxis a Buck által javasolt, ahol metódushívásokkal építjük fel a logikai kifejezéseket és állításokat:

```
1 A.grandparent_of(B).if(A.parent_of(X).and(X.parent_of(B)))
```

Mivel Ruby nyelvben bizonyos esetekben el lehet hagyni a zárójeleket, a fenti sor az alábbi módon egyszerűsíthető:

```
1 A.grandparent_of(B).if A.parent_of(X).and X.parent_of(B)
```

2.2.2. Tények és szabályok

Jamis Buck javaslatában [20] a tények nincsenek megkülönböztetve a logikai kifejezésektől:

```
1 'Bob'.parent_of('Greg')
2 A.grandparent_of(B).if A.parent_of(X).and X.parent_of(B)
```

Ezt a szintaxist nem tudtam egyszerűen megvalósítani, mivel nem lehet egyszerűen eldönteni, hogy egy `.parent_of()` metódushívás tény vagy egy logikai kifejezés része. Ezért a tényekre valamilyen eltérő szintaxist kellett bevezetnem. Úgy döntöttem, hogy a metódusnév `!` jellel végződő változatát fogom használni:

```
1 'Bob'.parent_of!('Greg')
```

A Ruby nyelvben a `!` jelre vonatkozó konvenció az, hogy a `!` jellel végződő metódus „a `!` jel nélküli metódus veszélyes változata”, ami tipikusan több maradandó változtatást végez, mint a `!` jel nélküli megfelelője [15]. A tényt létrehozó metódus végez változtatást, a logikai kifejezést leíró metódus pedig nem, tehát ezzel követtem a Ruby konvencióját.

Ruby nyelvben minden olyan esetben, ahol `if` kulcsszót lehet használni egy feltétel igaz voltának vizsgálatára, lehet helyette az `unless` kulcsszót használni, ami éppen ellenkezőleg, a feltétel hamis voltát vizsgálja. Ezért Rubylog nyelvben bevezettem az `unless` metódust a szabályok létrehozására, az `if` metódushoz hasonlóan, így teljesítve a legkisebb meglepetés elvét. A Prolog nyelvben ilyen típusú szabály nem létezik. Egy példa:

```
1 A.good.unless A.bad
```

2.3. Az átjárhatóság biztosítása

Az átjárhatóság biztosításához szükség van arra, hogy mind a gazdanyelven írt kód meg tudjon hívni a vendégnyelven írt kódrészleteket, mind fordítva. Továbbá szükséges, hogy mindkét irányban lehessen adatokat átadni.

A vendégnyelvi kód meghívása **kérdésekkel** történik, a Prolog rendszerekhez hasonlóan. A vendégnyelvi kód gazdanyelvi kódot a **gazdanyelvi predikátumokkal** vagy **gazdanyelvi függvényekkel** tud meghívni. A vendégnyelv felé kérdésekkel vagy gazdanyelvi függvényekkel tudunk átadni adatokat, visszafelé pedig **változók behelyettesítésével** lehet adatokat átadni. Ezeket a technikákat mutatom be a következőkben.

2.3.1. Kérdések

Egy Prolog rendszernek fel tudunk tenni kérdéseket, ahogy ezt az 1.2. példaprogramban is láttuk. A Rubylog rendszert is úgy terveztem meg, hogy egy kérdésfeltevessel induljon a logikai program futása. Két csoportra osztottam a kérdéseket: amikor szükség van minden egyes megoldásra (például szeretnénk megkapni egy személy minden egyes gyermekét, ezeket nevezem megoldandó kérdéseknek), vagy amikor csak az fontos, hogy van-e megoldás (ezeket nevezem eldöntendő kérdéseknek).

A **megoldandó kérdések** feltevésére létrehoztam a logikai kifejezések `solve` metódusát. A `solve` metódus a logikai kifejezést kérdésként értelmezi, és megpróbálja bizonyítani. Minden egyes helyes megoldására meghívja a neki átadott blokkot, amelyen belül az eredményt tartalmazó változók változó-behelyettesítéssel elérhetők (lásd 2.3.4). Például az alábbi program kiírja Greg összes gyermekét:

```
1 'Greg'.parent_of(X).solve do
2   puts X
3 end
```

A megoldandó kérdések feltevésére egy másik módot is kidolgoztam. A Ruby nyelv `Enumerable` (felsorolható) felületét felhasználva különböző módokon fel lehet sorolni egy megoldandó kérdés megoldásait. Minden felsoroló metódus, amit pl. a tömböknél lehet használni, a logikai kifejezéseknél is elérhető. Az alábbi példa ennek néhány használatát szemlélteti:

```
1 'Greg'.parent_of(X).collect{X} # => ["John!", "Jane!"]
2 'Greg'.parent_of(X).count # => 2
3 'Greg'.parent_of(X).any?{X == 'Jane'} # => true
```

Az 1. sorban szereplő kód, ami egy listába gyűjti egy kifejezés megoldásait, a Prolog nyelvbeli `findall` predikátumot tudja helyettesíteni.

Az **eldöntendő kérdések** szintaxisát hasonló elven dolgoztam ki, mint a tények szintaxisát. A Ruby nyelv konvenciói szerint a `?` jelre végződő metódusok igaz vagy hamis értéket adnak vissza [15]. Ezért úgy döntöttem, hogy az eldöntendő kérdéseket a megfelelő metódus `?` jeles változatával lehessen feltenni:

```
1 'Greg'.parent_of?('Jane') # => true
2 'John'.parent_of?('Jane') # => false
```

2.3.2. Gazdanyelvi predikátumok

A gazdanyelvi predikátumokat azért fejlesztettem ki, hogy Rubylog programrészek meg tudjanak hívni Ruby programrészeket. A gazdanyelvi predikátum egy Ruby nyelvű blokk, ami predikátumként fut le. Ha igaz értéket ad vissza, a predikátum sikeres, ha pedig hamisat, a predikátum sikertelen. Az alábbi példa azt mondja ki, hogy `X` egy gyermeke Gregnek, és `X` nem John:

```
1 'Greg'.parent_of(X).and{X != John}
```

A kapcsos zárójelek közötti rész normál Ruby kód, azzal a különbséggel, hogy az ott elérhető Rubylog változók értékei behelyettesítődnek (lásd 2.3.4).

A gazdanyelvi predikátumok lehetősége miatt bizonyos Prolog nyelvbeli predikátumokat nem kellett megvalósítanom Rubylog nyelvben. Ilyen az összes összehasonlító predikátum, ugyanis ezeket könnyen helyettesíteni lehet gazdanyelvi predikátumokkal, ahogy a fenti példában is tettük. A `write` predikátumot sem vittem át Rubylog nyelvbe, hiszen azt az alábbi gazdanyelvi predikátummal lehet helyettesíteni:

```
1 A.is('Hello').and { puts A; true }
```

A `;` `true` azért szükséges a végére, hogy a predikátum sikeres legyen, mivel a `puts` metódus hamis értéket ad vissza.

2.3.3. Gazdanyelvi függvények

A gazdanyelvi függvények célja, hogy adatokat adjunk át a gazdanyelvből a vendégnyelvi programnak. Több olyan predikátum van, amely egyik argumentuma helyett blokkot is tud fogadni, és a blokk eredményét veszi értéknek. Két ilyen predikátumot fogok most bemutatni: az `.is()` és az `.in()` predikátumokat.

Az `.is()` Rubylog-predikátum az `=` és az `is` Prolog-predikátumok funkcióit valósítja meg. Egyesíti az első argumentumát a másik argumentumával. A második argumentum helyett egy blokkot is meg lehet neki adni, ez esetben a blokkot meghívja, és az eredményét egyesíti az első argumentummal. A blokkban tetszőleges Ruby kifejezés állhat, a benne lévő Rubylog változók behelyettesítődnek, lásd 2.3.4. Az alábbi példa két kifejezés Prolog és Rubylog nyelvbeli megfelelőjét mutatja be.

```
1 X = Y.                                1 X.is(Y)
2 X is 3*Y+5.                          2 X.is{3*Y+5}
```

Az `.is()` predikátum segítségével számos olyan funkciót, ami Prolog nyelvben külön predikátum valósít meg, a Rubylog nyelvben meg lehet oldani a Ruby nyelvbeli megfelelő lehetőség használatával. Az alábbi néhány példa ezt mutatja be GNU-Prolog, ill. Rubylog nyelven.

```
1 atom_length('hello', N).              1 N.is{'hello'.length}
2 char_code('a', Code).                  2 Code.is{'a'.ord}
3 findall(X, parent('Greg', X), L).      3 L.is{'Greg'.parent_of(X).collect{X}}
```

Egy másik hasonló célú predikátum az `.in()`. Ez a predikátum a második argumentumában megadott felsorolható típusú objektum minden elemével sorra egyesíti az első argumentumát. Tehát először az elsővel, azután visszalépéskor a másodikkal, és így tovább. Ha egy visszalépésnél már nincs több elem, sikertelen lesz.

2.3.4. Változók behelyettesítése

A változókat a Ruby `const_missing` eseménykezelőjével valósítottam meg, amely akkor kerül meghívásra, ha a programban egy ismeretlen konstans szerepel. Tehát minden Rubylog változó-név a Ruby szempontjából ismeretlen konstans. A `const_missing` eseménykezelő használatával lehetőségem nyílt arra, hogy a gazdanyelvi blokkokon belül speciálisan kezeljem a változókat. Ezekben a helyeken, amennyiben a változónak van értéke, a változó értékét fogja visszaadni a változónév. Az alábbi példán látható ennek működése:

```
1 A.is(B).and C.is('Hello').and {
2   A # => B változó
3   B # => B változó
4   C # => 'Hello'
5 }
```

Sok fejtörést okozott, hogy amennyiben nincs értéke a változónak, a Rubylog visszaadja-e magát a változó objektumot, vagy `nil`-t adjon vissza helyette. Végül úgy döntöttem, hogy a változó objektumot adja vissza. Egyrészt azért döntöttem így, mert ha `nil`-t adna vissza, nem lenne teljesen konzisztens a nyelv viselkedése (struktúrák belsejében a változók soha nem cserélődnek ki `nil`-re), tehát a legkisebb meglepetés elvét megsérteném. Másrészt azért is így döntöttem, mert ha `nil`-t adna vissza, a `nil` objektumot nem lehetne tetszőlegesen felhasználni a Rubylog nyelvbeli adatként. Ebben a döntésben sokat segített egyik tanárom, Golda Bence tanácsa.

2.4. Kvantor jellegű predikátumok

A Ruby nyelv felsorolható típusainak `all?`, `any?`, `one?` és `none?` metódusai megmondják, hogy egy adott kifejezés igaz-e a felsorolás mindegyik, valamelyik, pontosan egy elemére, vagy nem igaz semelyik elemére sem. Ezeket **kvantor jellegű felsoroló metódusoknak** hívom, mivel hasonlítanak a logikában szereplő kvantorokra.

A 2.3.1. szakaszban elmondottak alapján a Rubylog logikai kifejezései felsorolható típusúak, tehát az alábbi metódushívások megengedettek:

```
1 'Greg'.parent_of(X).all?{X == 'Jane'} # => false
2 'Greg'.parent_of(X).any?{X == 'Jane'} # => true
3 'Greg'.parent_of(X).one?{X == 'Jane'} # => true
4 'Greg'.parent_of(X).none?{X == 'Jane'} # => false
```

Ezek a sorok felsoroló metódusok hívásai, de szintaxisuk nagyon hasonlít az eldöntendő kérdésekre. A legkisebb meglepetés elve alapján megvalósítottam, hogy ezek valóban eldöntendő kérdések legyenek, és ennek megfelelően létezzenek az `.all()`, `.any()`, `.one()` és `.none()` **kvantor jellegű predikátumok**. A hozzájuk tartozó eldöntendő kérdések (egy jelentéktelen esettől eltekintve) most pontosan úgy viselkedjenek, mint előtte a kvantor jellegű felsoroló metódusok.

Az így kapott kvantor jellegű predikátumokkal az alábbihoz hasonló logikai kifejezéseket lehet leírni:

```
1 'Bob'.grandparent_of(X).all('Greg'.grandparent_of(X))
```

vagyis igaz-e, hogy Bob minden unokája Gregnek a gyermeke. Az alábbi mondat azt fejezi ki, hogy Bob minden gyermekének van gyermeke:

```
1 'Bob'.parent_of(X).all(X.parent_of(ANY))
```

Itt használtuk az `ANY` névtelen változót jelezve, hogy nem vagyunk kíváncsiak a gyermek kilétére.

Általánosítottam a kvantor jellegű predikátumokat úgy, hogy a második argumentumuk elhagyható, és ilyenkor helyette `:true` lesz. Ezzel az első argumentum megoldásainak számára lehet rákérdezni. Például

```
1 X.parent_of('Bob').any
```

azt fejezi ki, hogy van-e szülője Bobnak.

Bevezettem egy újfajta kvantor jellegű predikátumot, az `.iff()` predikátumot, amely akkor sikeres, ha az első argumentum minden megoldására a második argumentum igaz lesz, és fordítva is, például

```
1 'Greg'.parent_of(X).iff('Bob'.grandparent_of(X)) # => true
```

2.4.1. Prefix jelölés

A „minden ember halandó” mondat Rubylog nyelvbeli megfelelője a fenti kvantor jellegű predikátumok segítségével felírva az alábbi:

```
1 X.man.all X.mortal
```

Ez a mondat szintaktikailag és szemantikailag ugyan helyes, de nem követi jól a *reads like English* elvet, mivel az angolban az *all* szót nem ilyen szórendben használják. Ehelyett úgy döntöttem, hogy a kvantor jellegű predikátumokat lehessen prefix jelöléssel is használni:

```
1 all X.man, X.mortal
```

Ezt már fel lehet olvasni egy angol mondatként: „for all X where X is a man, X is mortal”, vagy röviden, „all men are mortal”. Hasonlóképpen lehet használni a prefix jelölést a többi kvantor jellegű predikátumra:

```
1 any X.child_of('Bill'), X.girl
2 one X.in(1..10), X.divides(49)
3 none(X.prime) { X < 2 }
```

Ezek kiolvasva így hangzanak: 1. „any child of Bill is a girl”, 2. „one number in 1..10 divides 49”, 3. „none of the primes is less than 2”.

2.5. Listák kezelése

A Rubylog nyelv fejlesztésénél az volt az egyik célom, hogy minden Prolog nyelven írt programot könnyen át lehessen írni Rubylog nyelvre, beleértve a listákat használó Prolog programokat is. Az egyik fontos probléma, ami itt felmerült, a parciális listák kezelése volt. A Prolog nyelvben a parciális listák olyan listák, melyeknek hossza nem definiált. Például a $[1, 2 | T]$ lista első eleme 1, a második 2, a hátralévő részlistája pedig a T változó. Ez az adatszerkezet, bár elsőre furcsának tűnhet, valójában nem bonyolult. A Prolog láncolt listaként tárolja a listákat, az előbbi példa valójában egy $f(1, f(2, T))$ szerkezetű struktúra. A teljes listák esetében pedig a lánc végén a $[]$ szimbólum áll, tehát az $[1, 2]$ lista megfelelője $f(1, f(2, []))$.

Egyik lehetőség lett volna Rubylog nyelvben is – a Prologhoz hasonlóan – struktúrákból láncolt listaként implementálni a listákat. A Ruby-Prolog nyelv is hasonló módon teszi. Ekkor nyilvánvaló lett volna a megfeleltetés a Prolog és Rubylog nyelvbeli listák között, ugyanakkor valahogy meg kellett volna oldani a Ruby tömbök és a Rubylog listák közötti konverziót. A Ruby-Prolog nyelvben explicit konverzió szükséges. Ugyan meg lehetett volna oldani az implicit konverziót is, de azt gondolom, hogy a programozónak fölösleges terhet jelentett volna, ha két különböző lista adattípust kell felváltva használnia. Nekem az volt a törekvésem, hogy a Rubylog nyelvben a Ruby tömb típusát használjam, hogy ne kelljen a konverzióval foglalkozni.

2.5.1. Parciális listák

A parciális listák kezelésére valamilyen megoldást kellett találnom. Két lehetőség kínálkozott: az egyik, hogy egyáltalán nem is implementálom a parciális listákat. A másik lehetőség, hogy a parciális listáknál valamilyen speciális módon tárolok egy referenciát a lista végét reprezentáló változóra, és ezt az egyesítő algoritmusnál figyelembe veszem.

Meglepő módon nagyon sok algoritmust sikerült megírnom Rubylog nyelven, anélkül, hogy parciális listákat használnék. Sok Prolog algoritmus csupán arra használja a parciális listák lehetőségét, hogy leválassza vagy beszúrja egy lista első elemét. Épp ezért úgy gondoltam, hogy ennek a lépésnek a biztosítása elegendő lesz arra, hogy a Rubylog minden Prolog algoritmust támogasson anélkül, hogy a parciális listák kezelését megvalósítanám. Bevezettem az `L.splits_to(H, T)` predikátumot, amely lényegében az $L=[H|T]$ Prolog kifejezés megfelelője. Így például a `member` Prolog predikátumot a következőképp lehetett implementálni Prolog és Rubylog nyelven:

```
1 member(A, [A]).
2 member(A, [_|T]) :- member(A, T).

1 A.member_of! [A]
2 A.member_of(L).if L.splits_to(ANY, T)
3 .and A.member_of(T)
```

Látható, hogy a Rubylog kód 2–3. sorában lévő állítás két predikátumhívásból áll, míg a Prolog nyelvbeli megfelelője csak egyből. Amit a Prologban írt program a `[_|T]` listával történő egyesítéssel végez, azt a Rubylog nyelv korábbi változatában expliciten, a `splits_to` predikátum meghívásával kellett elvégezni.

Sok program esetében az `L.splits_to(H,T)` a fentiekhez hasonlóan használhatónak bizonyult. Amennyiben `L` vagy `T` definiált és nem parciális, akkor ezzel a megoldással nem jöhet létre parciális lista, tehát ezekben az esetekben a program helyesen működött. Azonban amikor sem `L`, sem `T` nem volt definiálva, az `L` listának parciálisnak kellett volna lennie. Ezt úgy próbáltam megoldani, hogy ilyen esetekben az `L.splits_to(H,T)` predikátumnak tetszőlegesen sok megoldása volt, különböző hosszúságú listák, melyeknek az első eleme `H`, például `[H]`, `[H,A]`, `[H,A,B]`, stb. Ez további néhány program esetében jó megoldást adott, azonban bizonyos programokban végtelen ciklusba futott. Ezért végül mégis úgy döntöttem, hogy implementálom a parciális listákat.

A parciális listák szintaxisára a Ruby nyelv egy szintaktikai elemét, a tömb behelyettesítő szintaxist (*splat* szintaxis) használtam fel. Ezzel meg tudtam oldani, hogy a Prolog nyelvhez viszonylag hasonló, és a Ruby nyelvbe teljesen beleilleszkedő szintaxissal elérhető legyen a Prolog nyelv szemantikája. Például az `[1,2|T]` Prolog-lista megfelelője az `[1,2,*T]` Rubylog-lista lett, ahol `T` egy Rubylog változó. Ha az `[1,2,*T]` listát egyesítjük az `[1,2,3,4]` listával, akkor a `T` változó tartalma `[3,4]` lesz.

2.5.2. Általánosított parciális listák

Az `[1,2,*T]` szintaxis ötletét a legkisebb meglepetés elve alapján továbbgondolva általánosítottam, és ezzel egy olyan funkciót építettem be a Rubylog nyelvbe, amely messze túlnyúlik a Prolog lehetőségein. Először is, megengedtem, hogy máshol álljon a változó, nem csak a lista végén. Például Rubylog nyelvben a `[*L,2,3]` is egy érvényes parciális lista. Másrészt, megengedtem, hogy több változó is szerepeljen a listában, például `[1,*L,3,*M]`.

Ez az újítás egy alapvető változást jelent a Prolog működéséhez képest. Ugyanis Prolog nyelvben egy egyesítésnek csak egyféle sikeres kimenetele lehet. Így viszont Rubylog nyelvben egy egyesítésnek több megoldása is lehet. Például az `[1,2]` és `[*X,*Y]` listák egyesítésének háromféle eredménye lehet: `X=[],Y=[1,2]` vagy `X=[1],Y=[2]` vagy `X=[1,2],Y=[]`.

Az új szintaxissal és a hozzá tartozó egyesítő algoritmussal meg lehet valósítani több olyan listákat kezelő algoritmust, amit a Prolog nyelvben csak predikátumok segítségével lehet. Például két listát összefűzni az alábbi módon lehet Prolog, ill. Rubylog nyelven:

```
1 append(A,B,L)                                1 [L].is [*A,*B]
```

A Rubylog nyelvbeli `[L].is [*A,*B]` szintaxis könnyebben értelmezhető, mint az `append(A,B,L)`, mert nem kell ismerni az argumentumok helyes sorrendjét, és megfelel a *reads like English* elvnek.

A parciális listákat használhatjuk szabályok fejrészében, ezzel bizonyos esetekben tömörebb kódot kaphatunk. A következő két program egy lista megfordítottját definiálja, Prolog, ill. Rubylog nyelven (a Prolog program forrása: [23]).

```
1 reversed([], []).                                1 [].reversed!([])
2 reversed([A|X],L):- reversed(X,Y),              2 [A,*X].reversed([*Y,A]).if X.reversed(Y)
3                                     append(Y,[A],L).
```

Látható, hogy a Rubylog nyelven írt kódrészlet 2. sorában nem volt szükség az `append` predikátum használatára, mert a fejrészben használtuk a kibővített lista-szintaxist. Ezenkívül úgy

gondolom, hogy a Rubylog kódrészlet logikai kifejezései könnyebben értelmezhetők az olvasó számára.

Végül a szakasz elején bemutatott `member` predikátum most már így írható le Prolog, ill. Rubylog nyelven:

```
1 member(A, [A]).                                1 A.member_of! [*ANY, A, *ANY]
2 member(A, [_|T]) :- member(A, T).
```

Így a Rubylog kód már csak egyetlen tényből áll, és első elolvasásra könnyen értelmezhető. Ez is mutatja, hogy ezzel az újítással a Rubylog lényegesen kifejezőbb lett a Prolog nyelvnél.

2.6. Parciális sztringek

A Ruby nyelvben létezik egy sztring-behelyettesítő szintaxis is, a tömb-behelyettesítő szintaxishoz hasonlóan. Egy sztringben `#{}` jelek között bármilyen kifejezés előfordulhat, melynek értéke bekerül a sztringbe. A legkisebb meglepetés elve alapján kidolgoztam a parciális sztringeket a parciális tömbökhöz hasonlóan. Így a sztring-behelyettesítő szintaxissal beilleszthetünk sztringek belsejébe változókat részszttringként. Prolog nyelvben nincsenek sztringek, csak szimbólumok és listák, de egyik típus sem ad lehetőséget a parciális sztringeknek megfelelő adatszerkezet létrehozására. Tehát a Rubylog nyelv ezzel egy új fajta adatszerkezettel lett gazdagabb. Az új szintaxis nagyon hasznosnak bizonyult, és jóval kifejezőbbé tette a Rubylog nyelvet. Ezt egy példán keresztül fogom bemutatni.

Tekintsük a következő feladatot: a Képek mappában van néhány .jpg kiterjesztésű fájl, amit .png kiterjesztésűvé szeretnénk konvertálni. A megoldáshoz használjuk a `convert` külső programot, melyet a következőképp kell meghívni:

```
1 convert "Képek/valami.jpg" "Képek/valami.png"
```

A feladat egy megvalósítását megadtam Prolog nyelven (2.2. kódrészlet), Ruby nyelven (2.3. kódrészlet) és Rubylog nyelven (2.4. kódrészlet).

A Rubylog nyelvű kódrészlet 1. sorában látható a parciális sztring használata. A `.file_in()` predikátum sorra veszi a megadott mappában lévő fájlok elérési útjainak listáját, és az első argumentumot sorra egyesíti velük. Ha parciális sztringet teljes sztringgel egyesítünk, az egyesítő algoritmus megfelelően figyelembe veszi a sztringben lévő változókat, és szükség szerint értéket ad nekik. Tehát az `A` változó megkapja a .jpg kiterjesztésű fájlok nevének és elérési útjának a kiterjesztés előtti részét.

A 2. sorban lévő „`„` jelek közötti sztringet a Ruby értelmező végrehajta külső programként. A 2. sorban található `#{..}` sztring behelyettesítő szintaxisok a Ruby nyelvbeli eredeti szemantikájukkal működnek, hiszen a bennük lévő `A` kifejezés már nem változót jelöl, hanem a konkrét értéket, a változó-behelyettesítés miatt (2.3.4).

A 2.4 kódrészlet nagyon jól érthető, akár olyanok számára is, akik nem programozók. Ezzel kapcsolatos tapasztalataimról a 3.1.1. szakaszban fogok beszélni.

```

1 convert_all_jpgs :-
2     directory_files('Képek', Filenames),
3     member(Filename, Filenames),
4     atom_concat(A, '.jpg', Filename),
5     atom_concat(A, '.png', NewFilename),
6     atom_concat('convert "Képek/', Filename, S1),
7     atom_concat(S1, '" "Képek/', S2),
8     atom_concat(S2, NewFilename, S3),
9     atom_concat(S3, '"', S4),
10    shell(S4),
11    fail
12 or halt.
13 :- initialization(convert_all_jpgs).

```

2.2. kódrészlet: Konvertáló program Prolog nyelven

```

1 Dir["Képek/*.jpg"].each do |file|
2   `convert "#{file}" "#{file.sub(/\.jpg$/, ".png")}"`
3 end

```

2.3. kódrészlet: Konvertáló program Ruby nyelven

```

1 "#{A}.jpg".file_in("Képek").each do
2   `convert "#{A}.jpg" "#{A}.png"`
3 end

```

2.4. kódrészlet: Konvertáló program Rubylog nyelven

3. fejezet

Az eredmény értékelése

3.1. Az érthetőség értékelése

Ahogy eddig is beszéltem róla, a Rubylog nyelv fejlesztésénél fontos szempontnak tekintetem a kódérthetőséget. Azonban mindeddig csak kvalitatív megfigyeléseket tettem ezzel kapcsolatban.

Érdekes kérdés, hogy hogyan lehet egy programozási nyelv érthetőségét, olvashatóságát számszerűen vizsgálni. A *Central Washington University*-n végeztek egy kísérletet arra vonatkozóan, hogy bizonyos programozási fogalmak nevei mennyire jól írják le a fogalmakat. Programozó és nem programozó hallgatókat kérdeztek meg különböző programozási fogalomokról. Minden fogalomhoz fel volt sorolva több név vagy jelölés, amelyekről meg kellett mondani, hogy mennyire írják le a fogalmat. Számtottévő különbségek jelentkeztek a programozó és a nem programozó alanyok válaszaiban, például a nem programozók szerint a `for` szó sokkal kevésbé írja le a ciklus fogalmát, mint a programozók szerint. A kutatók azt a kérdést vizsgálták, hogy ha egy programozási nyelv elemeit az alapján tervezzük meg, hogy mennyire intuitív egy nem programozó számára, akkor jobb minőségű nyelvet kaphatunk-e [24]. A mérési eredmények alapján elkészítettek egy programozási nyelvet. Úgy találták, hogy az így létrehozott nyelvet kevesebb hibával tudják használni kezdő felhasználók, mint a Perl programozási nyelvet [25].

E kísérlet alapvető felvetését én is hasznosnak tartottam, miszerint érdemes laikusokat megkérdezni a programozási nyelvről. Elvégeztem én is egy kísérletet, jóval kevesebb alannyal és kevésbé céltudatosan, de sok hasznos információt nyertem belőle.

3.1.1. Saját kísérletem

Öt emberrel végeztem kísérletet, kettő közülük amatőr programozó, hárman nem programozók, de mindannyian írtak már programot. Mindnyájuknak néhány rövid programrészletet mutattam, és el kellett mondaniuk, hogy melyiknek mi a feladata. Nagy részük Rubylog nyelven volt, de volt közöttük más nyelven írt program, és volt olyan, aminek semmi értelme nem volt. Mindnyájuknak szintaxiskiemeléssel mutattam a kódot, ez segíthetett nekik a szerkezetek felismerésében.

Egyik érdekes megfigyelésem az volt, hogy a 2.4. kódrészletben látható képkonvertáló programot mindenki azonnal felismerte.

Érdekes volt megfigyelni, hogy a két programozó miket talált furcsának a nyelvben. Egyikőjük statikus típusrendszerű nyelvekhez volt szokva, neki furcsa volt, hogy nincs a változók típusa (szám vagy tömb) deklarálva. Másikuk furcsálta, hogy a tények esetében a `!` karakter nem logikai

```

1 predicate_for Integer, ".prime .divides()"
2 A.divides(B).if B.product_of(ANY,A)
3 P.prime.if all X.in{2..(P-1)}, X.divides(P).false
4 P.in(1..20).and(P.prime).each { puts P }

```

3.1. kódrészlet: Prímkereső program Rubylog nyelven

tagadást jelent.

A képkonvertáló program kapcsán felmerült a kérdés, hogy a `file_in` predikátum az almappokban is keres, vagy csak közvetlenül a megadott mappában. Ezért úgy döntöttem, hogy be fogok vezetni egy predikátumot, ami az almappokban is keres.

A 3.1. kódrészletben látható program célját az ötből négyen felismerték (kiírja 1-től 20-ig a prímszámokat). Hárman közülük matematika szakterületen tanulnak, ők részletesen el is magyarázták a működését, teljesen helyesen. Azonban egyikőjük sem vette észre, hogy a feladat megvalósítása valójában hibás. (Mi a hiba? Válasz a 21. oldalon!) Ebből azt a fontos következtetést vontam le, hogy lehet bármilyen egyszerű és jól olvasható a nyelv, a rejtett hibák lehetőségét ez nem küszöböli ki.

Az öt emberből háromnak mutattam be a dolgozat elején szereplő Családfa programot (2.1. kódrészlet), miután már néhány Rubylog nyelven írt programot elmagyaráztam nekik. Mindhárman pontosan le tudták írni a működését. Tehát azt tapasztaltam, hogy a Rubylog nyelvet nem ismerő emberek (egy programozó, kettő nem) kevesebb, mint egy óra alatt meg tudták annyira érteni a nyelvet, hogy ezt a példát tökéletesen megértették.

Egy ilyen kevés alannal végzett kísérletből nem lehet statisztikai következtetéseket levonni. Azonban néhány fontos dolgot megtanultam belőle. Megismerkedtem a kísérleti módszerrel, úgyhogy legközelebb céltudatosabban tudom tervezni a hasonló jellegű kísérleteimet. A kísérlet során megtapasztaltam, hogy milyen nehézségekkel kerül szembe valaki egy új programozási nyelv tanulása közben. Visszaemlékeztem arra, hogy amikor én tanultam programozni, hogyan próbáltam a példaprogramok jelentését megfejteni. Azt gondolom, hogy ez fontos tapasztalat volt nekem mint leendő informatika tanárnak, hogy jobban tudjam a programozást tanítani.

3.2. Konklúzió

Kifejlesztettem a Rubylog nyelvet, mely a Prolog nyelv számos nyelvi elemét biztosítja, és bizonyos elemeket a Prolog nyelvénél általánosabban is megvalósítottam. Ugyanakkor a Ruby nyelvben alkalmazott használhatósági irányelveket igyekeztem követni, és ez sok esetben jól sikerült. Azonban további vizsgálatra szorul, hogy a Rubylog nyelv valójában mennyire hasznos.

3.2.1. Az olvashatóság haszna

Miután nagy erőfeszítést tettem azért, hogy a Rubylog nyelvvel jól olvasható kódokat lehessen írni, jogos a kérdés, hogy érdemes volt-e. Igaz-e, hogy ha egy programozási nyelv jól olvasható, akkor valóban jól használható is? Valóban hasznos-e, ha egy programozási nyelv szórendje egy természetes nyelv szórendjével egyezik?

E kérdésekre nem tudok tényekkel alátámasztott válaszokat mondani, csupán feltételezéseket. Az a tapasztalatom, hogy a programozónak a fejlesztés során sokszor kell a megírt kódot elolvasnia, akár hosszú idővel a megírása után. Ekkor időt takaríthat meg, ha a programot könnyű értelmezni. Azt feltételezem, hogy ez elég sokszor történik meg ahhoz, hogy érdemes legyen jól olvasható kódot írni.

A másik feltételezésem a nyelvi relativizmus elméletén alapul. A nyelvi relativizmus azt mondja ki, hogy a nyelvünk befolyásolja a gondolkodásmódunkat, mivel a nyelv nem csupán a kommunikáció eszköze, hanem a gondolkodásé is [26]. Úgy gondolom, hogy a programozónak valamennyire az adott programozási nyelven kell gondolkodnia. Ha ez a nyelv hasonló a programozó anyanyelvéhez vagy egy általa beszélt nyelvhez, könnyebben tud benne gondolkodni, és ezáltal programokat írni.

Ezek alapján felmerül a kérdés, hogy vajon a programozó anyanyelvétől függ-e, hogy melyik programozási nyelv a megfelelő neki. Szintén érdekes kérdés, hogy a programozó által „beszélt” nyelvnek tekinthetjük-e a matematika nyelvét, vagyis hogy a matematikai képleteket érdemes-e programkódban használni, avagy egy szöveges leírás hasznosabb. Mindezek a kérdések még sok vizsgálatra szorulnak.

3.2.2. Továbbfejlesztési lehetőségek

Végül beszélek azokról a területekről, amelyeken a Rubylog nyelv még fejlesztésre szorul.

- Egy fontos újítás lenne a parciális asszociatív tömbök (Hash) kidolgozása, a parciális listákhoz és sztringekhez hasonlóan. A Ruby legújabb, 2.0-s verziójában bevezetett asszociatív-tömb-behelyettesítő szintaxis (**A) ebben sokat fog segíteni.
- A Rubylog nyelvben az `if`, `and` és `or` metódusneveket választottam a logikai kifejezések leírására, hogy könnyebben érthető programokat kapjunk. Feltételezem, hogy sok leendő felhasználó szívesebben használna rövidebb neveket, ezért tervezem a `«`, `&` és `|` operátorokat bevezetni az `if`, `and` és `or` metódusok mellett.
- Egyelőre nem készült Rubylog nyelvvel nagyobb méretű program. Éppen ezért még nincsenek a nyelvben kiforrott modularizációs lehetőségek. Az objektumorientált programozást valamilyen szinten támogatja a nyelv, de ez is még fejlesztésre szorul.
- Szeretném megoldani, hogy a Rubylog programok hibakeresése gördülékeny legyen. A legjobb megoldás az lenne, ha a Ruby hibakeresőjét tudnám úgy bővíteni, hogy Rubylog programokat is jól lehessen vele nyomon követni.
- Méréseim szerint a Rubylog programok futása a lefordított Prolog programokoz képest két nagyságrenddel lassabb. Ezen kb. egy nagyságrendet lehetne javítani, ha a Rubylog programokat a Yield Prolog programokhoz hasonlóan automatikusan át lennének fordítva a gazdanyelvre. Úgy gondolom, hogy egy ilyen fordító megvalósítása sok munkát igényel, de szükséges ahhoz, hogy a Rubylog nyelvet piaci alkalmazásokban érdemes legyen használni.

Ezekon kívül úgy gondolom, hogy arra kellene törekedni, hogy a Rubylog nyelvet többen kezdjék el aktívan használni. A felhasználók visszajelzései nagyon fontosak a nyelv fejlesztése szempontjából.

Köszönetnyilvánítás

Köszönöm barátomnak és tanáromnak, Golda Bencének, hogy átadta nekem a Ruby nyelv iránti lelkesedését. Köszönöm témavezetőmnek, Kitlei Róbertnek a sok segítséget a tudományos szaknyelv elsajátításában. Köszönöm csoporttársamnak, Szigeti Rékának a dolgozat nagyon alapos elolvasását. Köszönet Daninak, Daninak, Istinek, Kristófnak és Rékának a kísérletben való kitartó részvételért. És köszönet Istennek, a Nagy Programozónak mindenért.

A 3.1. kódrészlet hibája: az 1-et is prímszámnak tekinti.

Irodalomjegyzék

- [1] Castor : Logic paradigm for C++. <http://www.mpprogramming.com/Cpp/>. Elérés ideje: 2013. 06. 07.
- [2] Computer languages benchmark game. Code-used time-used shapes. <http://benchmarksgame.alioth.debian.org/u32/code-used-time-used-shapes.php>. Elérés ideje: 2013. 06. 17.
- [3] A declarative applicative logic programming system. <http://kanren.sourceforge.net/>. Elérés ideje: 2013. 06. 07.
- [4] Domain specific languages. <http://dsl2013.math.ubbcluj.ro/>. Elérés ideje: 2013. 06. 07.
- [5] Embedding SWI-Prolog in other applications, SWI-Prolog documentation. [http://www.swi-prolog.org/pldoc/doc_for?object=section\(3,'9.4.21',swi\('/doc/Manual/foreigninclude.html'\)\)](http://www.swi-prolog.org/pldoc/doc_for?object=section(3,'9.4.21',swi('/doc/Manual/foreigninclude.html'))). Elérés ideje: 2013. 06. 07.
- [6] An interview with the creator of Ruby. Elérés ideje: 2013. 07. 06.
- [7] JLog homepage. <http://jlogic.sourceforge.net/>. Elérés ideje: 2013. 06. 07.
- [8] jsprolog. <http://ioctl.org/logic/prolog-latest>. Elérés ideje: 2013. 06. 07.
- [9] The philosophy of Ruby, A conversation with Yukihiro Matsumoto, part I, p. 4. <http://www.artima.com/intv/ruby4.html>. Elérés ideje: 2013. 06. 17.
- [10] Programming in Schelog. <http://www.ccs.neu.edu/home/dorai/schelog/schelog.html>. Elérés ideje: 2013. 06. 07.
- [11] Prolog-in-Scala. <https://github.com/sonoisa/Prolog-in-Scala>.
- [12] Pylog. <http://cdsoft.fr/pylog/>. Elérés ideje: 2013. 06. 07.
- [13] Pyrolog. <https://bitbucket.org/cfbolz/pyrolog/>. Elérés ideje: 2013. 06. 07.
- [14] Ruby-Prolog. <https://github.com/preston/ruby-prolog>. Elérés ideje: 2013. 06. 07.
- [15] Ruby study notes, Writing own Ruby methods. http://rubylearning.com/satishtalim/writing_own_ruby_methods.html. Elérés ideje: 2013. 06. 17.
- [16] Rulog. <https://github.com/jimwise/rulog>. Elérés ideje: 2013. 06. 07.
- [17] Styla. <http://code.google.com/p/styla/>. Elérés ideje: 2013. 06. 07.
- [18] Yield Prolog. <http://yieldprolog.sourceforge.net/>. Elérés ideje: 2013. 06. 07.
- [19] Borland International, Inc. *Turbo Prolog Owner's Handbook*, 1986.

- [20] J. Buck. Prolog in ruby. <http://weblog.jamisbuck.org/2006/10/28/prolog-in-ruby>. Elérés ideje: 2013. 05. 15.
- [21] M. A. Covington. ISO Prolog: A summary of the draft proposed standard. <http://fsl.cs.uiuc.edu/images/9/9c/PrologStandard.pdf>, 1993. Elérés ideje: 2013. 06. 17.
- [22] U. Endriss. Lecture notes – An introduction to Prolog programming. <http://staff.science.uva.nl/~ulle/teaching/prolog/prolog.pdf>, Jul 2007.
- [23] M. Malita. Examples in Prolog. <http://www.anselm.edu/homepage/mmalita/culpro/index.html>. Elérés ideje: 2013. 06. 17.
- [24] Gellenbeck E. Stefik, A. Empirical studies on programming language stimuli. <http://link.springer.com/content/pdf/10.1007%2Fs11219-010-9106-7.pdf>, 2009.
- [25] Siebert S. et al. Stefik, A. An empirical comparison of the accuracy rates of novices using the Quorum, Perl, and Randomo programming languages. <http://ecs.victoria.ac.nz/foswiki/pub/Events/PLATEAU/Program/plateau2011-stefik.pdf>, 2011.
- [26] C. Swoyer. The linguistic relativity hypothesis, "relativism", the stanford encyclopedia of philosophy (winter 2010 edition). <http://plato.stanford.edu/entries/relativism/supplement2.html>.