A PLanG programozási nyelv kiterjesztése

Önálló laboratóriumi beszámoló

2015.

Készítette:
Scipiades Ármin

Konzulens:
Dr. Feldhoffer Gergely

Pázmány Péter Katolikus Egyetem
Információs Technológiai és Bionikai Kar

Önálló laboratórium beszámoló

Név: Scipiades Ármin	**Neptun kód:** A5OGRN	**Képzés:** MI-Bsc
Dolgozat címe: A PLanG programozási nyelv kiterjesztése		
Konzulens(ek) neve: Dr. Feldhoffer Gergely		

A hallgató a kitűzött feladatot megfelelő színvonalon és a kiírásnak megfelelően teljesítette.

Az írásbeli beszámoló javasolt érdemjegye (számmal és betűvel):

Konzulens aláírása

Budapest, 2014. május 18.

Tartalomjegyzék

1. Bevezetés

A PLanG[1] programozási nyelvet 2005 óta használják karunkon a programozás alapjainak oktatására: a nyelvet és a hozzá tartozó fejlesztői környezetet direkt erre a célra tervezte és implementálta Lövei László.

A PLanG oktatási célú nyelv, tervezése során az elsődleges szempont a procedurális programok működésének demonstrálása volt[1], ezért eszköztára meglehetősen szegényes: hiányoznak az alprogramok, a módszertan oktatását segítő eszközök, de még az alapvetőnek tekinthető *elsif* konstrukció is hiányzik a nyelvből. A nyelv Turing-teljes ugyan (elvégre rendelkezik elágazással és ciklussal), így elvben minden fejlettebb programozói eszköz szimulálható rajta, de ez – alprogramok híján – csak nagyon nehézkesen lehetséges, a programozással épp csak ismerkedő diákoktól semmiképp nem várható el. Oktatási célú programozási nyelvként a PLanGnak egyszerűen használható, kiterjedt programozói eszköztárat nyújtva inspirálnia kellene a diákokat, megnyitni előttük a programozás világát. Ehhez képest azt tapasztaljuk, hogy a PLanG gyakran gátolja a tanulót a programozás alapelveinek elsajátításában.

Önálló laboratóriumi feladatom az volt, hogy ezt az intuitív meglátást racionális elvekkel támasszam alá: formális rendszert keresve értékeljem a PLanG minőségét, illetve vázoljak fel egy lehetséges kiterjesztést, ami javítaná a nyelv minőségét.

A 2. részben minőségi mutatókat keresünk a programozási nyelvekhez, különös tekintettel az oktatási célú programozási nyelvekre.

A 3. részben futólag áttekintjük a fordítóprogramok általános működését.

A 4. részben a 2. részben ismertetett metrikák szerint vizsgálom a PLanGot és javítási lehetőségeit, és beszámolok a PLanG kiterjesztett változatát fordító program prototípusáról.

[1] A betűszó feloldása Lövei László szíves közlése szerint egyszerűen *Programming Language*, de érdemes megemlíteni az egyetemen elterjedt legendát, hogy a PLanG a *Pázmány Language* rövidítése.

2. Programozási nyelvek

A programozási nyelv algoritmusok leírására szolgáló formális jelölésrendszer, amelyet emberek és számítógépek is értelmezni tudnak. A programozási nyelv szolgálhat ember és gép közötti kommunikáció eszközeként, lehetővé téve a programozónak, hogy a feladatot megoldó algoritmust a számítógép számára érthető módon megfogalmazza. De a programozási nyelv kommunikációs eszköz lehet ember és ember között is, hiszen a megírt programot a programozón kívül mások is olvasni fogják majd, akár mert javítaniuk kell a programon, akár mert tovább akarják fejleszteni, akár mert tanulni akarnak belőle.

2.1. Történeti áttekintés

Az embernek és a gépnek a közös nyelvvel szemben támasztott igényei jelentősen eltérnek. Kezdetben, a számítógépek megjelenésekor egyértelműen a gépek igényei voltak előtérben: a programozónak ténylegesen a gép nyelvén kellett beszélnie, **gépi kód**ban kellett közölnie a gondolatait, vagyis ténylegesen bináris számok sorozataként kellett megírnia a programját, ahol minden szám egy-egy gépi utasítást, illetve az utasítás operandusaiként szolgáló memóriacímeket jelölte. Ez nagyon távol áll az emberi gondolkodástól, a természetes nyelven kifejezett algoritmustól: a program lefordítása a gép nyelvére a programozó igen nehéz és munkaigényes feladata volt, a program olvasása pedig hasonlóan nehéz.

Az ember-orientált nyelvek kialakulásának első lépése a csak embereknek szánt programleíró jelölésrendszerek megjelenése volt: Konrad Zuse *Plankalkül*je, Neumann János és Herman Goldstine programleíró folyamatábrái, Currie lambda-kalkulusa voltak az első olyan rendszerek, amelyek lehetővé tették algoritmusok formális leírását. Érdemes megjegyezni, hogy bár a matematika nagyon fejlett eszköztárat épített ki statikus struktúrák leírására, az algoritmusokat egészen addig természetes nyelven adták meg[2].

A negyvenes évek végén jelentek meg az első **assembly-szintű nyelvek**, ahol bináris számok helyett a programozó már rövid nevekkel, mnemonikokkal hivatkozhatott egy gépi utasításra, és konkrét memóriacímek helyett már azonosítókkal hivatkozhatott változókra. Az így leírt programot táblázatok segítségével könnyű átírni gépi kódra, és ez a fordítási folyamat könnyen automatizálható is: így jelentek meg az assemblerek, az első fordítóprogramok. Ez volt at első lépés az ember-orientált nyelvek kifejlődése felé: a programozóknak nem kellett többé numerikus utasításkódokat észben tartaniuk vagy memóriacímeket számolniuk, sőt, a fordítóprogram már képes volt bizonyos hibákat felderíteni a kapott programban. És bár az assembly-szintű nyelvekben eleinte a gépi utasítások és az assembly mnemonikok között egy-egy kapcsolat volt, de hamarosan megjelentek az első „makrók”, azaz makróutasítások (vagyis szó szeint „óriásutasítások”) is, amelyek több gépi utasításból álló sorozatokat helyettesítettek. Megjelentek az első szubrutinkönyvtárak is, amelyek lehetővé tették a programozó számára a magasabb absztrakciós szinten való gondolkodást.

Az ilyen korai *autocoding* technikák fejlődésével alakultak ki a **magasszintű nyel-**

vek, melyeket architektúrafüggetlenség, a magas absztrakciós szint biztosítása, az assemblyhez képest rövidebb, áttekinthetőbb szerkezetű programok írásának lehetősége jellemez, és az, hogy a nyelv közelebb áll az emberi nyelvhez. Fejlődésükben katalizátorként hatott Noam Chomsky generatív nyelvelmélete, melynek alapján viszonylag komplex nyelvtani struktúrákat hatékonyan felismerő és gépi nyelvre fordító algoritmusokat lehetett kidolgozni.

A magasszintű nyelvek kifejlesztésének motivációja a programozói termelékenység növelése volt[3], a programozók körében mégis sokáig érezhető volt egyfajta ellenérzés a magasszintű nyelvekkel szemben, mondván, hogy a magasszintű nyelvek emberorientáltsága a *hatékonyság* és a *kreativitás* rovására megy[4, 12. fejezet]. A számítógépek teljesítményének fejlődésével és az egyre szofisztikáltabb fordítóprogramok megjelenésével ezek a kritikák háttérbe szorultak ugyan, de ezzel párhuzamosan egyre magasabb szintű, egyre emberorientáltabb programozási nyelvek jelentek meg, melyekkel szemben a korábbi magasszintű nyelvek kedvelői fogalmaztak meg hasonló kritikákat. A *magasszintű nyelv* kifejezés így egyre magasabb szintű nyelvekre utal: az első magasszintű nyelveket egy mai programozó már jórészt alacsonyszintűnek tekinti. A ma programozója számára „túl magas” szintű programozási nyelvek például a programkódot automatikusan generáló szoftvertervezési eszközök; ezekkel szemben a mai programozók hasonlóan a hatékonyságot és a kreativitást féltik, mint a hatvanas évek programozói a magasszintű nyelvektől[5].

2.2. Fogalmak

A nyelv **szintaxis**a azon szabályok halmaza, amelyek megadják a nyelven írható összes formailag helyes programot. A szintaxis megadható formális környezetfüggetlen[2] nyelvtannal[6, p. 25], ekkor a nyelv szintaktikailag helyes programjainak halmaza a nyelvtan által generált mondatok halmaza, $\mathbb{M}_{\mathcal{L}}$. Egy ilyen mondat levezetési fáját szintaxisfának nevezzük. Megkülönböztethetjük a *felszíni* és *absztrakt* szintaxist: míg a felszíni szintaxis szabályai tényleges szövegekkel, karakterláncokkal operálnak, és ezekről döntik el, hogy alkothatnak-e formailag helyes programot; addig az absztrakt szintaxis szabályai a program „mélystruktúrájával” foglalkoznak, és nem törődnek azzal, hogy egy absztrakt nyelvi elemnek milyenek a felszíni megnyilvánulásai, lexikális tulajdonságai. Az absztrakt szintaxishoz **lexikális szabályok**at rendelve felszíni szintaxist kapunk.

A nyelv **szemantiká**ja a formálisan helyes mondatok jelentését megadó szabályok összessége. Megkülönböztetjük a *statikus szemantikát* és a *dinamikus szemantikát*: a statikus szemantika a programok futtatás előtti viselkedésével, míg a dinamikus szemantika a program futásidejű viselkedésével foglalkozik. A statikus szemantika szabályai a formailag helyes programok halmazát tovább szűkitik egy $\mathbb{P}_{\mathcal{L}} \subseteq \mathbb{M}_{\mathcal{L}}$ halmazra, amelyet az *érvényes programok* halmazának nevezünk: a gyakorlatban ezek azok a programok,

[2] Bizonyos programozási nyelvek szintaxisát környezetfüggőnek mondják, híres példa erre a C++, azonban a környezetfüggőség gyakran a lexikális sajátosságok eredménye. Definícióink mindenképp határozottabbak és egyértelműbbek lesznek, ha a szintaxist mindig környezetfüggetlennek tekintjük, és a környezetfüggő elemeket a statikus szemantika hatáskörébe soroljuk; ráadásul ez a megközelítés jól tükrözi a fordítóprogramok működését.

amelyeket a fordítóprogram hiba nélkül elfogad. A dinamikus szemantika szabályai minden érvényes programhoz megadják a program pontos futásidejű viselkedését.

Az absztrakt szintaxis elemeit **programkonstruktor**oknak, a programkonstruktorokat és a hozzájuk kapcsolódó szemantikai szabályokat együttesen pedig **nyelvi elemek**nek nevezzük. Ezek azok az alapvető építőelemek amelyeket használva a programozó felépítheti programját.

Bizonyos elemhalmazokról igazolható, hogy ha ezeket tartalmazza egy nyelv, akkor a nyelv **univerzális**, más néven **Turing-teljes**, vagyis a nyelvvel szimulálható egy Turing-gép, ami azt jelenti, hogy minden lehetséges algoritmus kifejezhető csak az adott elemeket használva. Procedurális nyelvek esetében történelmi és kényelmi okokból a *szekvencia*, az *elágazás* és a *ciklus* halmazának meglétét szokás a nyelv univerzalitását biztosítónak tekinteni: ha egy nyelvben ez a három nyelvi elem létezik, akkor az a nyelv biztosan univerzális[7].

A fentiek alapján formális definíciót is adhatunk a programozási nyelv fogalmára:

Definíció (Programozási nyelv[8, 3.1 definíció])**.** *Egy $\mathcal{L}$ programozási nyelv megadható az $\mathcal{L} = (\mathbb{M}_\mathcal{L}, \mathbb{P}_\mathcal{L}, eval_\mathcal{L})$ rendezett hármassal, ahol*

- *$\mathbb{M}_\mathcal{L}$ a nyelv mondatainak, a nyelv $\mathbb{K}_\mathcal{L} = \{K_1 \dots K_n\}$ programkonstruktoraiból szabadon generált absztrakt szintaxisfák halmaza.*
- *$\mathbb{P}_\mathcal{L} \subseteq \mathbb{M}_\mathcal{L}, \mathbb{P}_\mathcal{L} \neq \emptyset$, az $\mathcal{L}$-beli programok halmaza.*
- *$eval_\mathcal{L}$ a $\mathbb{P}_\mathcal{L}$ felett értelmezett predikátum, a nyelv dinamikus szemantikája, amely akkor igaz, ha a program elvégzi a feladatát*[3]*.*

Az így meghatározott absztrakt nyelv lexikális sajátosságait egy $R : \mathbb{K} \to \Sigma^*$, a nyelv programkonstruktoraihoz karaktersorozatokat rendelő függvény megadásával definiálhatjuk. Ha ez a leképezés nem injektív, a nyelv felszíni szintaxisa környezetfüggővé válhat.

Definíció (Konzervatív kiterjesztés[8, 3.2 definíció])**.** *Egy $\mathcal{L}$ nyelv $\{K_1 \dots K_n\}$ elemekkel való konzervatív kiterjesztése egy $\mathcal{L}'$ nyelvnek, ha $\{K_1 \dots K_n\} \cap \mathbb{K}_{\mathcal{L}'} = \emptyset$, $\mathbb{K}_\mathcal{L} = \mathbb{K}_{\mathcal{L}'} \cup \{K_1 \dots K_n\}$, $\mathbb{M}_{\mathcal{L}'} \subset \mathbb{M}_\mathcal{L}$, $\mathbb{P}_{\mathcal{L}'} \subset \mathbb{P}_\mathcal{L}$, és $\forall P \in \mathbb{P}_{\mathcal{L}'} : eval_{\mathcal{L}'}(P) \Rightarrow eval_\mathcal{L}(P)$. Ekkor $\mathcal{L}'$-t $\mathcal{L}$ leszűkítésének mondjuk.*

Vagyis $\mathcal{L}$-t úgy kapjuk, hogy $\mathcal{L}'$-höz hozzáadunk pár új nyelvi elemet, de minden, a régi nyelven írt program a kiterjesztett nyelvben is program, és ezek futásidejű viselkedése sem változik.

A konzervatív kiterjesztésre a $\mathcal{L} = \mathcal{L}' + \{K_1 \dots K_n\}$ jelölést, míg a leszűkítésre a $\mathcal{L}' = \mathcal{L} \setminus \{K_1 \dots K_n\}$ jelölést használjuk.

Definíció (Nyelvhasználat)**.** *Azt mondjuk, hogy egy K programkonstruktor nem része egy programozó nyelvhasználatának, ha programjaiban soha nem jelenik meg K. Ekkor a programozó által használt $\mathcal{L}'$ nyelv az ideális $\mathcal{L}$ nyelv leszűkítése, $\mathcal{L}' = \mathcal{L} \setminus \{K\}$.*

[3] Felleisen $eval_\mathcal{L}$-t olyan predikátumként definiálja, ami akkor igaz, ha a program terminál[8], ezzel definíciója a lehető legáltalánosabb. Ezt túl megengedőnek érzem, ezért választottam ezt a homályos, ám intuitív meghatározást. Ennek pontosabb definiálására egy lehetőség $eval_\mathcal{L}\ \langle feladat, program \rangle$ párok feletti értelmezése.

Egy programozó nyelvhasználatából egy nyelvi elem hiányozhat technikai megfontolásokból (mert „nem hatékony”), mert a használata nem illendő (jó példa erre a `goto` utasítás, ami ugyan a legtöbb nyelvben máig létezik, még olyan modern nyelvekben is, mint a Ruby, de használata a magasszintű programozásban teljesen eltűnt), vagy akár mert nem tud az adott nyelvi elem létezéséről.

Naivnak tűnhet a feltételezés, hogy egy képzett programozó nem tud egy nyelvi elemről, de gondoljunk bele, hogy egy nyelv standard könyvtárának függvényei is nyelvi elemek; másrészt a modern programozási nyelvek gyakran nagyon komplexek, emberek számára nehezen megismerhetőek a maguk teljességében.

2.3. Programozási nyelvek minőségi mutatói

Az univerzális programozási nyelvek számításelméleti szempontból teljesen azonosak: tetszőleges rendelkezésre álló idő és memória mellett bármely számítás elvégezhető velük. Intuitívan érezzük viszont, hogy a nyelvek között minőségi különbségek vannak, ez a különbség viszont nehezen megfogható: magas és alacsony szintű nyelvek között könnyű minőségi különbséget találni, de két magasszintű nyelv minőségét nehéz összehasonlítani.

2.3.1. Kifejezőerő

A programozók informálisan gyakran mondják, hogy ez vagy az a nyelv *nagyobb kifejezőerővel bír*: hogy egy algoritmus könnyebben, elegánsabban implementálható egy nyelvben, mint a másikban, hogy egy nyelvben könnyebben kifejezik magukat, mint egy másikban. Az az intuitív meglátásunk is hamar kialakul, hogy némely nyelvi elem fontosabb a kifejezőerő szempontjából, mint a többi: némelyik lényeges, némelyik meg csak *szintaktikus cukormáz*, amely könnyen kifejezhető más nyelvi elemekkel, így a nyelv kifejezőerejéhez nem ad hozzá, csak édesebbé teszi a nyelvet az emberek számára.[9]

A kifejezőerő szubjektív fogalom, de voltak kísérletek a definiálására: a hackerkultúrában[10] és a szoftvertechnológiában[4][11] is meghatározó nézet szerint egy nyelv kifejezőereje a *tömörségével* (*succintness*) arányos – egy nyelv annál kifejezőbb, minél kevesebb szóval tudok elmondani valamit. Ez jól illik a kifejezőerőről alkotott intuitív képünkhöz: a tökéletes programozási nyelv az volna, amelyben bármely feladat egyetlen utasítással megoldható.

Ha a kifejezőerő arányos a nyelv tömörségével, a kifejezőerő mérhető egy program sorokban, karakterekben, utasításokban mért hosszával, esetleg a tömörített forráskód méretével[4]: ezek statisztikák készítésére könnyen alkalmazható metrikák, de az eredményeket jelentősen torzítják a nyelv lexikális sajátosságai, névadási és tördelés konvenciói, melyekről érezzük, hogy bár fontosak, nem kellene igazán befolyásolniuk a kifejezőerőt. Ezt a problémát kiküszöbölhetjük, ha a tömörséget egy program absztrakt

[4] Az elterjedt tömörítő eljárások optimálisak, közelítik a Shannon forráskódolási tételéből ismert tömörítési határt, így a tömörített file mérete mond valamit a forrásszöveg entrópiájáról, információtartalmáról. Ez a megközelítés vonzóan tudományosnak tűnhet, amíg fel nem ismerjük, hogy a forrásszöveg entrópiája alapvetően a programot leíró szöveg karaktereinek eloszlásától függ, ami tisztán lexikális kérdés, és nincs nagyobb összefüggésben a nyelv kifejezőerőerejével, mint a karakterek száma. Ilyen megközelítésért lásd például a The Computer Language Benchmarks Game oldalt: `http://benchmarksgame.alioth.debian.org`.

szintaxisfájának elemszámával mérjük, de megfelelő elemzőt írni minden vizsgálandó nyelvhez fáradságos feladat. Akármilyen metrikát választunk is, a program méretének mérése, mint a statisztikai nyelvtechnológiai eszközök általában, csak akkor használható, ha nagy korpusszal dolgozunk, a nyelv ismerete önmagában tehát nem elégséges a nyelv kifejezőerejének vizsgálatához. Ráadásul a kifejezőerőt tisztán a tömörséggel magyarázó elmélet – formális definíció híján – nem ad lehetőséget, hogy bármit rigorózusan bizonyítsunk egy nyelvről. Nem igazán teszi lehetővé azt sem, hogy nyelvi elemekről érveljünk: szintaktikus cukormáz-e, vagy létfontosságú?

Mathias Felleisen a formális rendszerek elméletének eszköztárát felhasználva épített rendszert a kifejezőerő leírására és elemzésére[8]. Informálisan úgy foglalhatjuk össze, hogy $\mathcal{L}$ kifejezőbb $\mathcal{L}' = \mathcal{L} \setminus \{K_1 \dots K_n\}$-nél, ha egy $\mathcal{L}$-beli, valamely K_i konstruktort tartalmazó program lefordításához $\mathcal{L}'$-re a program globális átszervezése szükséges. Azokat a programkonstruktorokat pedig, amelyeknek fordításához csak lokális transzformációkra van szükség, kifejezhetőnek vagy *eliminálhatónak* mondjuk. A $\varphi : \mathcal{L} \to \mathcal{L}'$ fordítási függvény tulajdonságainak megkötésével definiálhatjuk a kifejezőerő különböző szintjeit.

Felleisen rendszerét használva a nyelvtervezés folyamán formálisan is bizonyítható, hogy egy nyelvi elem hozzáadása ténylegesen változtat-e a nyelv kifejezőerején.

2.3.2. Belső kiterjeszthetőség és makrókifejezhetőség

Egy kiterjeszthető nyelvben a programozó új nyelvi elemeket hozhat létre. A legtöbb nyelv kiterjeszthető valamilyen szinten, például lehetőséget biztosít alprogramok, új adattípusok létrehozására. Néhány nyelv arra is eszközt biztosít, hogy a programozó tetszőleges új nyelvi elemeket hozzon létre: a Lisp makró konstrukciói segítségével például a nyelv tetszőleges programkonstruktorokkal bővíthető. Ezeket az eszközöket a *szintaktikai absztrakció eszközei*nek nevezzük, és segítségükkel a kifejezőerő egy új szintjét definiálhatjuk.

Azt mondjuk, hogy egy K nyelvi elem *makrókifejezhető* $\mathcal{L}'$-ben, ha létezik egy megfelelő A szintaktikai absztrakció, amelyre $\varphi(K(e_1 \dots e_n)) = A(\varphi(e_1) \dots \varphi(e_n))$; vagyis a nyelvi elem eliminálása nem csak a program globális struktúráját őrzi meg, hanem az eliminált L-mondat alkotóelemeinek struktúráját is[8, 3.11 definíció]. Ez a definíció nagyon közel áll a szintaktikus cukormázról alkotott intuitív képünkhöz.

Látható, hogy a nyelv kifejezőerejét megsokszorozza egy szintaktikai absztrakciós eszköz; és minél megengedőbbek az absztrakciós eszközök, annál nagyobb a nyelv kifejezőereje. Így például egy alprogram-absztrakciót biztosító nyelv, amelynek alprogramjai elfogadnak paraméterként szekvenciát, nagyobb kifejezőerejű, mint amelyiknek alprogramjai nem tudnak szekvenciát kezelni paraméterként.

2.3.3. Hasznosság és használhatóság

A programozási nyelv az ember és gép közötti kommunikáció eszköze, ezért értelmezhetőek rá a felhasználói felületek minőségi mutatói[12]. Egy programozási nyelv **hasznos**

(*useful*) ha lehetővé teszi, hogy a programozó gyorsan és hatékonyan írja meg a feladatát megvalósító programot[6]. A hasznosság fő metrikái:

Kifejezőerő Két nyelv közül a nagyobb kifejezőerejű a hasznosabb. Ez ugyan reláció, és nem metrika, de a nyelv szintaktikai absztrakciós eszközeinek száma és minősége jó becslés egy abszolút kifejezőerőre.

Robosztusság A programozó hibája nem járhat katasztrofális következményekkel. A nyelv robusztus, ha nyelvi elemei lehetővé teszik a hatékony hibadetektálást.

Emberorientáltság A nyelv emberorientált, ha amikor csak lehet, a programozó igényeit a gép igényei elé helyezi.

Feladatorientáltság A nyelvnek csak azokat a nyelvi elemeket kell biztosítania, amelyeket a programozók feladataik során használnak; és nem szabad olyan nyelvi elemeket biztosítania, amelyeket nem használnak. Ugyanis minél több nyelvi elemet biztosít egy nyelv, annál komplexebb, nehezebben használható lesz, arányosan kisebb lesz a programozók nyelvhasználata.

Ezek a metrikák gyakran nehezen mérhetőek, de objektívek. Ezzel szemben a **használhatóság** (*usability*) metrikái eredendően szubjektívek, egy nyelv – vagy bármely rendszer – használhatósága a felhasználó demográfiai hátterétől és tapasztalatától függ[5]. A használhatóság analízisére Green adott 1989-ben objektív, racionális rendszert, amelyet a *jelölésrendszerek kognitív dimenziói*nak (*cognitive dimensions of notation*, CD) nevezett el[13]. A CD rendszere 14, páronként független dimenziót ad egy jelölésrendszer használhatóságának meghatározására[14]. Ezekből a programozási nyelvek szempontjából legfontosabbak a következők:

Absztrakció Milyen szintaktikai absztrakciós eszközöket biztosít a nyelv? Hány ilyen eszköz használatát kell elsajátítani a nyelv minimális használatához?

Hibák vonzása Mennyire növeli a nyelv a programozói hibák előfordulásának esélyét?

Konzisztensség Hasonló dolgok hasonló módon fejezhetőek ki a nyelvben? Hasonló szemantikájú nyelvi elemek hasonló szintaktikával és lexikális tulajdonságokkal rendelkeznek?

Leképezés közelisége Mennyire könnyen fejezhető ki a nyelven a programozó algoritmusa, hány nyelvi elem felel meg a programozó elvi megoldásának egy lépésének, milyen könnyen fejezhető ki a feladat állapotterének műveletei a nyelvben?

Szellemi erőfeszítés Mekkora erőfeszítést kíván a nyelv használata?

Szerepkifejezés Mennyire nyilvánvaló, hogy egy dolog mire jó? Milyen mértékben lehet következtetni egy nyelvi elem szemantikájára a szintaxisa és lexikális tulajdonságai alapján?

Terjengősség Milyen röviden lehet kifejezni egy algoritmust a nyelven? Milyen hosszúak a nyelv lexémái?

Viszkozitás Mekkora a kis változtatások ára? Mennyire változtatható egy program lokális struktúrája a globális struktúra változtatása nélkül?

Egyik dimenzió sem egyértelműen „jó” vagy „rossz”, a dimenziók megítélése a nyelv feladatától függ.

2.4. Oktatási célú programozási nyelvek

A programozás oktatása más követelményeket támaszt egy nyelvvel szemben, mint az ipari felhasználás[12]. A kognitív dimenziók optimális értékei a kifejezetten a programozás alapjainak oktatására szánt nyelvre a következőképpen alakulnak:

Absztrakció – alacsony Az oktatási nyelv *absztrakciós korlát*jának, vagyis a nyelv használatához minimálisan megtanulandó absztrakciós eszközök számának nagyon alacsonynak kell lennie, és egyszerű programok írásához nem is szükséges túl sok absztrakciós eszköz. A Java nyelvben például már a legegyszerűbb program írásához találkozni kell az objektumosztály fogalmával.

Hibák vonzása – nagyon alacsony A hiba frusztrálja és elbizonytalanítja a tanulót, a nyelvtervezés során minden áron törekedni kell a programozói hibalehetőségek minimalizálására. Az C++ utasításlezáró pontosvesszőjének elhagyása például gyakori hiba, amit a kezdők nehezen is azonosítanak, hiszen a `gcc` fordítóprogram hibaüzenete szót sem ejt pontosvesszőről.

Konzisztensség – közepes A konzisztensség segíthet a tanulónak, hiszen ha elsajátította egy nyelvi elem használatát, akkor automatikusan használni fogja tudni a hasonló nyelvi elemeket. Ugyanakkor nem biztos, hogy a kezdő képes felismerni egy szemantikai hasonlóságot két nyelvi elem között, illetve nem tudja kihasználni az elvi hasonlóságot. Jó példa a túlzásba vitt konzisztensségre a Turing és az Ada nyelvek tömbelem-lekérdező operátora, ami ugyanúgy sima zárójelként jelenik meg, mint a függvényhívás, mondván, hogy mindkettő értéklekérés; a diák ugyanakkor egy `A(1)` konstrukciót látva nem tudja eldönteni, hogy `A` változó-e vagy függvény.

Leképezés közelisége – nagyon magas A nyelvnek illeszkednie kell a tanulók feladatához és megoldásához, tükröznie kell a tanulók létező ismereteihez, világképéhez; de mutatnia kell az elsajátítandó világképet is. A Pascal oktatónyelv például jó leképezése egy ideális virtuális gépnek, de a kezdők világképétől távol áll, ami különösen a tömbök és a karakterláncok nehézkes kezelésénél szembetűnő.

Szellemi erőfeszítés – alacsony A programozás során elkerülhetetlen a szellemi erőfeszítés, de nem szabad, hogy a nyelv használata szükségtelenül nehezítse a feladat megoldását. A Scheme oktatónyelvben például a lambda-kalkulus igényeihez idomulás nehéz feladat a kezdő programozónak.

Szerepkifejezés – magas Jól megválasztott felszíni szintaxis mellett a tanuló sokkal könnyebben olvashat és érthet meg programokat. Egy jó szerepkifejezésű nyelvi elem nem igényel különösebb magyarázatot, használata azonnal értetődő. Például a C++ `cout « "Hello world";` utasítása sokkal kevésbé szerepkifejező, mint a Pascal-szerű `print "hello world"`: a `cout` szóval ellentétben a `print` felismerhető a laikus számára is, a `«` operátor megértése pedig abszolút fejlettebb tudást kíván.

Terjengősség – közepes A nyelvnek strukturálisan elég tömörnek kell lennie ahhoz, hogy egyszerű programokat nagyon röviden, redundancia nélkül ki lehessen fejezni, de a túlzott tömörség az olvashatóság rovására megy. Bőbeszédű nyelv például az AppleScript, amelynek a természetes nyelvhez való hasonlósága gyakran a programstruktúra átláthatatlanságához vezet.

Viszkozitás – alacsony A programozás tanulása gyakran jár kísérletező próbálgatással[12]; ha ez a próbálgatás, a programok változtatgatása nehéz és kényelmetlen feladat, a tanuló lelkesedése csökkenni fog, frusztrációhoz vezethet. Magas viszkozitású nyelv például az Ada: deklarációs blokkjai, erős típusossága, import-szabályai, nehézkes fordítási mechanizmusa mind nehezítik a gyors próbálgatást.

A kifejezetten a programozás alapjainak egyetemi szintű oktatására szánt nyelv esetében a hasznosság metrikái közé soroljuk a **módszertani helyesség** támogatását: a nyelvnek alakítania kell a diák gondolkodásmódját, jó programozói szokások, programozástechnikai és számítástudományi alapvetések elsajátítására kell késztetnie. Ez gyakran ellentétbe kerül a használhatóság metrikáival: módszertanilag helyes megkülönböztetést tenni például függvények és eljárások között, bár a tanulók ilyen megkülönböztetést nem tesznek[12].

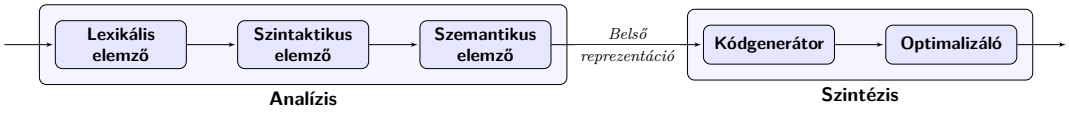
3. Fordítóprogramok

Formálisan a fordítóprogram egy $T : \mathbb{P}_{\mathcal{L}} \to \mathbb{P}_{\mathcal{L}'}$ transzformációs függvény, melyre $\forall P \in \mathbb{P}_{\mathcal{L}} : eval_{\mathcal{L}}(P) \Rightarrow eval_{\mathcal{L}'}(T(P))$. $\mathcal{L}$-t **forrásnyelv**nek, $\mathcal{L}'$-t pedig **tárgynyelv**nek nevezzük. A forrásnyelv általában magasszintű, a tárgynyelv pedig alacsonyabb szintű, például assembly-szintű, vagy gépi kódú nyelv: az ilyen átalakítást végző programot *compiler*nek nevezzük[6].

A compiler feladata a gyakorlatban az, hogy a forrásnyelvi programot az embereknek szánt, szöveges formából a gép számára futtatható programmá alakítsa.

3.1. A compiler általános felépítése

A compiler feladatát két jól elkülöníthető részfeladatra bontjuk: az **analízis** fázisában a forrásnyelvi programot elemzi, és az arról nyert tudást a program egy **belső reprezenzáció**nak nevezett struktúrában tárolja; majd a **szintézis** fázisában a belső reprezentáció alapján felépíti a tárgynyelvi programot[6][15].



1. ábra. A compiler felépítése.

Az analízis első feladata, hogy a forrásnyelvi programot reprezentáló karaktersorozatot a nyelv lexikális elemeit reprezentáló szimbólumok sorozatára bontsa. Ezt a feladatot a *lexikális elemző* végzi. A kapott szimbólumsorozatot a *szintaktikus elemző* vizsgálja tovább, a szekvenciális mondatból próbálja felállítani a program hierarchikus struktúráját: azt a kérdést próbálja megválaszolni, hogy egy formális nyelvtan milyen levezetése adja ki a kapott mondatot. Ha a program szintaktikailag helyesnek bizonyul, akkor tovább elemzi a *szemantikus elemző*, amelynek feladata, hogy ellenőrizze, megfelel-e a program a statikus szemantika szabályainak.

Az analízis kimenete a belső reprezenztáció, amely a forrásnyelvi programmal szemantikailag ekvivalens *köztesnyelv*i program. A belső reprezentáció a gyakorlatban sokféle lehet: valamilyen fastruktúra, valamilyen absztrakt utasítások szekvenciális sorozata.[15]

A szintézis során a *kódgenerátor* a belső reprezentáció minden egységéhez szemantikailag ekvivalens tárgynyelvi utasítást rendel. A kapott tárgynyelvi programon a *kódoptimalizáló* további transzformációkat végez, hogy növelje a kapott program hatékonyságát.

3.2. A környezetfüggetlen analízis eszközei

Miután a lexikális elemző a program szövegét a forrásnyelv felszíni szintaxisát leíró G környezetfüggetlen nyelvtan terminális szimbólumainak s sorozatává, a nyelv *mondat*ává, alakította, a szintaktikus elemző feladata konstruktívan bizonyítani, hogy s valóban levezethető G-ből. A bizonyítás konstruktív volta az jelenti, hogy az elemző algoritmus

ténylegesen megadja az s-t generáló levezetési fát. Egy nyelvtant *nemegyértelmű*nek mondunk, ha létezik hozzá olyan mondat, amely többféleképpen levezethető, ami avval jár, hogy különböző elemzésekhez különböző tárgyprogramok tartoznak – ez nem kívánatos tulajdonság.

A környezetfüggetlen analízis jól kutatott, megoldottnak tekinthető probléma[5]. Earley 1968-ban adott algoritmust, amely tetszőleges környezetfüggetlen nyelvtant $\mathcal{O}(n^3)$ időben elemez, azonban sok algoritmus létezik, amely a környezetfüggetlen nyelvek halmazának valamely részhalmazába tartozó nyelveket ennél hatékonyabban, vagy egyszerűbben implementálható módon elemzi. A legfontosabb ilyen nyelvosztályok az $LR(1)$ grammatikák, az $LL(k)$ és az $LL(1)$ grammatikák osztálya. Az ezen nyelvosztályokat meghatározó algoritmusokkal könnyen automatizálható módon *elemző táblázat*okat építhetünk a nyelvhez, amelyek lineáris időben képesek feldolgozni a bemeneti karaktersort[6]. Az ilyen elemző táblázatokat építő automatizáló eszközöket az iparban széles körűen használják. Ilyen például a híres *YACC*, illetve nyílt forráskódú változata, a *GNU bison* (módosított $LR(1)$ elemzők), és a modernebb, Java-alapú *ANTLR* (módosított $LL(k)$ elemző).

Az automatizált elemzőgenerátorok hasznos eszközök, de a fordítóprogramok tervezői gyakran mégis kézzel írják meg az elemzőt. Régi, elterjedt, bevált módszer a *rekurzív leszállásos elemzés* módszere, melynek lényege, hogy a nyelvtan minden nemterminális szimbólumához a szimbólum helyettesítési szabályát mgvalósító eljárást rendelve, egy vagy több előreolvasott szimbólum alapján döntést hozva, az implementációs programozási nyelv végrehajtási vermét használva szimulálunk veremautomatát.

A rekurzív leszállásos elemező előnye a táblázatos módszerekkel szemben, hogy nagyon könnyű megvalósítani, lehetőséget ad informatív hibaüzenetek generálására, és jól olvasható is, amennyiben az elemzőt végignézve megismerhetjük a nyelvtant. További előny, hogy az eljárásokban megkötés nélkül ki tudjuk használni a Turing-teljes implementációs nyelv teljes eszközkészletét, vagyis a szimulált veremautomatánk kifejezőerejét tetszés szerint növelhetjük – igaz, az implementációs komplexitás kárára. Rekurzív leszállásos elemzőt használ például a Lua programozási nyelv fordítóprogramja.

[5] Ami nem jelenti, hogy ne lenne fejlődés, ne születnének máig újabb, hatékonyabb algoritmusok, megközelítések: például Bryan Ford 2004-ben ismertette a Chomsky-nyelvosztályokból kilépő *Parsing Expression Grammar* nevű nyelvosztályt és az ezen alapuló szintaktikai elemzőt[16].

[6] Ezen és egyéb elemző algoritmusok leírásáért lásd [6, 4-6. fejezetek].

4. A PLanG programozási nyelv kiterjesztése

4.1. Követelményfeltárás

Mint minden szoftverfolyamatnak, a nyelvtervezés első fázisa is a követelmények feltárása: a nyelv feladatterének elemzése, a felhasználói igények felmérése. Célszerű lett volna kérdőíves felmérést végezni elsőéves diákok részvételével, komolyabban kutatni a nyelvek pedagógiai aspektusait, illetve elemezni lehetett volna az egyetem rendelkezésére álló nagyméretű korpuszt; erre erőforrások hiányában nem került sor.

Mivel létező nyelv kiterjesztéséről van szó, elemezni kellett a létező nyelvet is a 2.4. részben ismertetett metrikák szerint.

4.1.1. A PLanG programozási nyelv értékelése

A PLanG első pillantásra szembetűnő jellegzetessége, hogy kulcsszavai magyar nyelvűek, ami magyar diákok számára növelheti a szerepkifejező erőt. Mégis sok kulcsszó esetlen, furcsa megfogalmazású (`MEGNYIT`, `KI`, `KEREK`, `RND`), bár rövid (ami csökkenti a terjengősséget). Míg az angol programozási nyelvek kulcsszavai, függvénynevei rendszerint felszólító módú igék, addig a PLanG módszeresen főneveket és mellékneveket használ, még a függvénynevekre is. Az ilyen függvények, utasítások szerepkifejező ereje alacsony (vajon `KEREK 10.5` kerekíti az értéket, vagy azt mondja meg, kerek szám-e az érték? a `NAGY 'z'` azt mondja, nagybetű-e a paraméter vagy nagybetűvé alakít?). A kulcsszavak közül a `CIKLUS` nem feltétlen érthető egy olyan diáknak, aki még soha nem programozott; szerepkifejezőbb lehetett volna például az `*ISMÉTELD` kulcsszó.

Az operátorok között is vannak alacsony szerepkifejezésű elemek: az `@` infix operátor jelentése még gyakorlott programozóknak sem nyilvánvaló, de nem magától értetődő az sem, hogy a `/=` az áthúzott egyenlőségjel helyett áll, és a `DIV` és a `/` osztások közül sem evidens, melyik melyik. Hasonlóképp, bár kis programozói előképzettséggel „nyilvánvaló” a `:=` értékadó utasítás szemantikája, kezdőknek nem feltétlen az: anekdotikus bizonyíték szerint viszonylag gyakori hiba, hogy a diák nem tudja, melyik oldal adja, és melyik kapja az értéket. Valóban, a `:=` jó példája a *memetikus kompatibilitásnak*, vagyis hogy csak azért csinálunk valamit úgy, mert mások is úgy csinálták, és nem vesszük figyelembe az eltérő igényeket[12, 6.1.5.2. rész]. Szintén alacsony a tömbdeklarációk szerepkifejező ereje (`EGÉSZ[5]`), ez is a memetikus kompatibilitásra törekvésből fakad.

Az I/O utasítások szerepkifejező ereje viszonylag magas, használatuk kevés szellemi erőkifejtést kíván. A szöveg típusú változók I/O-kezelése nem idempotens[7], de mivel a beolvasás a sor végéig tart, a leképezés közeliségének szempontjából zavaró esetek száma kisebb, mint olyan nyelvekben, ahol a beolvasás az első szóközig tart.

A leképezés közelisége közepes-alacsony. Jól teljesítenek az operátorok, amelyek viselkedésükben és megjelenésükben is megfelelnek a diákok matematikai tudásának. Különösen szép, hogy az unáris függvényeket nem kell zárójelezni (`sin x`), és egyedi, ám nagyszerű az elemszám-lekérdezés cirkumfix operátora (`|tomb|`). Zavaró lehet a ma-

[7] vagyis létezik olyan x szöveg típusú érték, amelyet kiírva, majd a kiírt értéket beolvasva y-ba $x \neq y$

tematikai jelölésben megszokott *`a < b < c` jellegű asszociatív összehasonlítás hiánya. Nehézséget jelenthet az `EGÉSZ` és `VALÓS` típusok megkülönböztetése, közelebb állna a diákok gondolkodásához egy egyszerű *`SZÁM` típus.

A fix méretű tömbök valamelyest távol vannak a hallgatók gondolkodásától: ismerősebb lenne a tanulóknak egy, a matematikai halmazokra jobban emlékeztető típus. A tömböknek ráadásul nagyok kevés művelete van: lehet persze úgy érvelni, hogy műveletek megvalósítása a hallgató feladata, ebből tanulnak – azonban az absztrakciós eszközök teljes hiánya a hallgatót arra kényszeríti, hogy minden esetben külön, manuálisan, ciklussal végezze el a műveleteket, ami csökkenti a robosztusságot és növeli a szellemi erőfeszítést.

A nyelvtan szép, letisztult, előreolvasást nem igénylő $LL(1)$-es nyelvtan. Nagy erénye, hogy nincs szükség utasításlezáró jelre, ez jelentősen csökkenti a „becsúszó" hibák[ld. 12, 4.2.2 rész] valószínűségét. A legtöbb hibalehetőséget a változódeklarációs rész teremti, mert a PLanG nem deklarációs blokkot, hanem címkézett felsorolást használ, és a felsorolásból nagyon könnyű elhagyni a vesszőt. Az így keletkezett hibát viszonylag nehéz feladat detektálni, a referenciaimplementáció nem is teszi meg. A deklarációs rész a nyelv viszkozitását is növeli: új változó bevezetéséhez a használat helyétől távosli deklarációs részt kell szerkeszteni, amit nehezít a hibavonzó szintaxis. Általában, a deklarációs rész jó példája a gép- és nem emberorientált tervezésnek: az emberek számára csak csekély haszna van, elsősorban a gépek számára hasznos, megkönnyíti a feldolgozást és a kódgenerálást.

A PLanG egyáltalán nem nyújt absztrakciós eszközöket, a programozó semmilyen formában nem változtathat a nyelven. Az alprogramok hiánya nagyban növeli a terjengősséget és nagyobb szellemi erőfeszítést kíván, a felhasználó által definiált típusok hiánya csökkenti a robosztusságot és a módszertani helyességet.

Módszertani helyességet támogató eszközök nincsenek: dedikált hibakezelési eszköz hiányában a tanuló nem sajátíthatja el a megfelelő hibakezelést; a nyelv a megjegyzéseken kívül nem kínál strukturált, formális eszközt az előfeltételek és utófeltételek rögzítésére (holott a tárgy ezek használatára hangsúlyt helyez)

Ezen kívül hiányzik a nyelvből egy `elsif` konstrukció és a deklarációval egybekötött változóincializálás lehetősége. Bár mindkettő kifejezhető a PLanG programkonstruktoraival, az így kapott szerkezetek olvasása nehezebb, írása kevésbé hibatűrő.

4.1.2. A futtatókörnyezet értékelése

A PLanG használata a tanulók számára elválaszthatatlan a grafikus futtatókörnyezet használatától, így a nyelv elemzéséhez hozzátartozik a futtatókörnyezet hasznosságának és használhatóságának vizsgálata is.

Kétségkívül hasznos a kifejezésfát és a memóriamodellt kirajzoló modul, bár haszánalatuk nem intuitív. A programszerkesztő modul nagyon primitív, hiányzik a szintaxiskiemelés, zárójelpárosság-ellenőrző (ez növeli a hibák vonzását), undo funckionalitás, gyorsbillentyűk nincsenek (ez növeli a viszkozitást). Az alapvető editorfunkciók közül hiányzik a tabulátor méretének megadásának lehetősége, a legutóbb szerkesztett file-ok

	PLanG	ideális
absztrakció	nincs	alacsony
hibák vonzása	közepes	nagyon alacsony
konzisztensség	közepes	közepes
leképezés közelisége	közepes-alacsony	nagyon magas
szellemi erőfeszítés	közepes	alacsony
szerepkifejezés	közepes-magas	magas
terjengősség	alacsony-közepes	közepes
viszkozitás	közepes-magas	alacsony

1. táblázat. A PLanG kognitív dimenziói az ideális értékekkel összehasonlítva

megnyitásának lehetősége. A gombsorban a nagy zöld „Futtatás” gomb jó használhatóságú, de például a „Szerkesztés” és az „Értelmezett program szerkesztése” gombok közötti különbség egyáltalán nem világos.

A futtatókörnyezet nem interaktív, a programok előre bekészített bemenetekkel dolgoznak, ez ellentétes a tanulók előzetes várakozásával. A fordítás folyamata két lépésből áll, ez is meglepetést okozhat, és növeli a nyelv viszkozitását. A futtatókörnyezet által biztosított filekezelés nagyon zavaró, nem intuitív: nem valódi, hanem virtuális file-okkal dolgozik. Ez rendszerint megzavarja a tanulókat, magyarázatot igényel, és csökkenti a tanuló PLanGba vetett hitét, csökkenti motivációját.

4.1.3. Összegzés

A PLanG kifejlesztésének célja az volt, hogy az addig papíron írt pszeudokódot futtathatóvá tegye, illetve hogy eszközt adjon a procedurális programok működésének demonstrálására, elemzésére[1]. Bár ennek a célnak jól megfelelt, aktívan használt, a kurzus alapjaként szolgáló oktatási célú programozási nyelvként használhatósági mutatói meglehetősen rosszak (lásd az 1. táblázatot).

4.1.4. Ajánlások

Alapvető fontosságú alprogramok és összetett típusok definiálásának lehetősége, ez növelné a nyelv hasznosságát, és javítana több használhatósági dimenzión is.

A használhatóság tekintetében sokat nyernénk a deklarációs lista blokká alakításával, vagy akár a változók szabad, programtörzsön belüli deklarációjának engedésével. Engedni kellene a változó deklarációval egybekötött inicializálását.

A nyelv szerepkifejező ereje növelhető lenne a függvénynevek felszólító módú igévé átalakításával, még ha ezzel a programszöveg „gyerekesebbnek” is tűnik.

Kevés költséggel járna egy `assertion` konstrukció implementálása, amely eszközt biztosítana az előfeltételek, utófeltételek kezelésére. Szintén olcsó, de a hasznosságot növelő elem egy `hiba` konstrukció, amely lehetővé tenné a programozónak a hiba precíz jelzését.

Ajánlott lenne a tömbök kezelésének egyszerűsítése, például egy `foreach` konstrukció bevezetésével, de egy dinamikusabb tömb típus bevezetése is előnyös lehet.

4.2. Tervezési döntések

A tervezett nyelvet XPLanGnak (*eXtended PLanG*, azaz kiterjesztett PLanG) neveztem el, és úgy döntöttem, a PLanG konzervatív kiterjesztése lesz, vagyis minden érvényes PLanG program érvényes XPLanG program is lesz. Ennek az az előnye, hogy így az XPLanG könnyen kiválthatja a PLanGot; másrészt izgalmasnak tűnt a lehetőség, hogy az XPLanG a régi, elsősként írt PLanG programjaimat értelmezni tudja.

A megvalósításhoz használt nyelvnek a lefordított program korlátlan hordozhatósága miatt a Javát választottam. Legalacsonyabb verziószámú támogatott virtuális gépnek abszolút elterjedtsége miatt a hatos JVM-et választottam, bár felmerült az ötös JVM támogatása is, mivel az egyetem `turdus` szerverén csak ötös verziójú Java fut. Már a projekt ötletének felmerülésekor volt egy olyan hátsó szándékom, hogy népszerűsítsem a fordítóprogramokat hallgatótársaim körében, ezért minél hozzáférhetőbb szoftvert szerettem volna írni, amelynek megértése, fordítása, módosítása könnyű, ezért határoztam el, hogy minél kevesebb függőség felhasználásával fogok dolgozni. Azt is eldöntöttem, hogy automatizált parser generátor használata helyett kézzel fogok rekurzív leszállásos elemzőt írni, egyrészt demonstratív jellege miatt, másrészt mert megkönnyíti az informatív hibaüzenetek generálását.

Minél általánosabb fordítóprogramot akartam írni, egy keretrendszer-félét, amely felszíni szintaxissal paraméterezhető. Az volt az álmom, hogy több leszármazott nyelvet kezeljen a program: a PLanGot, a PLanG apró felhasználhatósági javításokkal ellátott kiterjesztését, majd még több, egyre jobban kiterjesztett PLanG változatokat; ráadásul azt is szerettem volna elérni, hogy a lexikális tulajdonságok minél könnyebben testreszabhatóak legyenek (erre [17] inspirált), például azért, hogy külföldi vendéghallgatók is tudják használni a nyelvet.

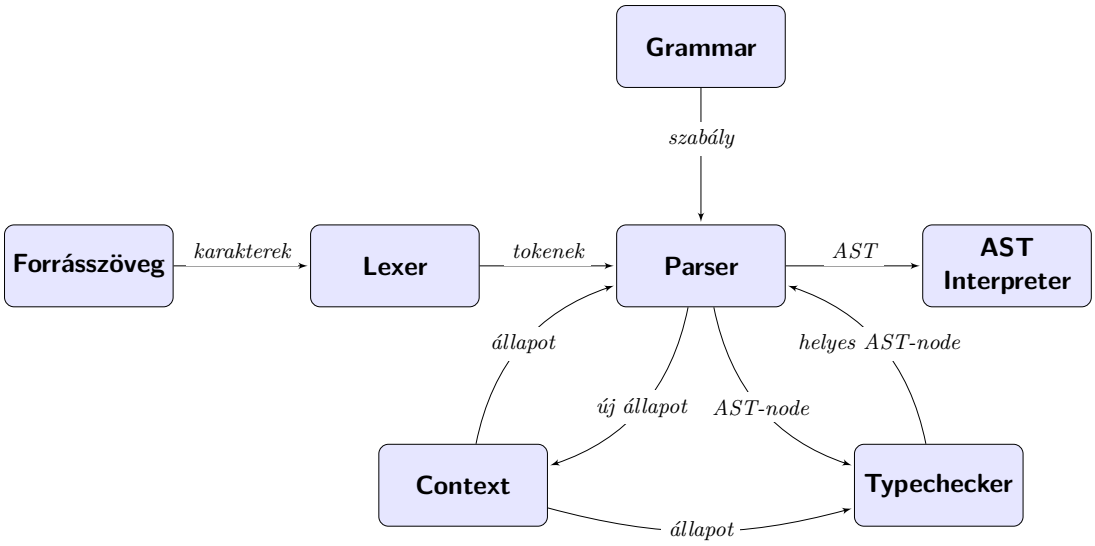
A fenti döntések utólagos értékeléséért lásd a 4.4 részt.

4.3. A prototípus implementációja

A prototípus központi osztálya a rosszul elnevezett `Parser`, amely létrehozásakor egy, valamely nyelv nyelvtanát enkapszuláló `Grammar` objektummal paraméterezhető. A `Parser` rendelkezik még egy `Lexer` objektummal, ami a programot leíró szimbólumsorozat forrása, és egy `Context` objektummal, ami a szimbólumtáblákat enkapszulálja.

4.3.1. A nyelvtan megadása

A fordítóprogram által használt nyelvtant egy `Grammar` objektumban adjuk meg: viszonylag szép, deklaratív szintaxissal sorolhatjuk meg a nyelv által használt lexikális elemeket a hozzájuk tartozó felszíni formákkal (ezek karaktersorozatok vagy reguláris kifejezések), beépített függvényeket, operátorokat. A nyelvtan helyettesítési szabályait egy rekurzív leszállásos elemző függvényeiként adjuk meg.



2. ábra. A prototípus idealizált adatfolyam diagramja.

4.3.2. Lexikális elemzés

A `Lexer` a forrásszöveget bontja fel szimbólumokra a `Context`ben tárolt szimbólumlista alapján. Implementációja nagyon egyszerű: a forrásszöveg bemeneti karakterfolyamára megpróbáljuk ráilleszteni a szimbólumokat leíró reguláris kifejezéseket, majd a leghosszabb egyezést tekintjük találatnak, és ezt adjuk fel a `Parser`nek. Ezt az implementációt eleinte ideiglenesnek szántam, de a feladathoz mérten hatékonynak és megbízhatónak bizonyult.

Fontos megjegyezni, hogy a `Lexer` valójában nem szimbólumokat ad vissza, hanem `Token`eket. A `Token` rekord tartalmazza a szimbólumot, a tulajdonképpeni karaktersorozatot, illetve a karaktersorozat előfordulásának helyét a szövegben.

4.3.3. Szintaktikus elemzés

A `Parser` a szintaktikus elemzést a `Grammar` `S` metódusának hívásával kezdi meg, ahonnan a nyelvtan szabályfüggvényeinek rekurzív hívásaival megy tovább. Ezen szabályfüggvények visszatérési értéke egy `Node`, azaz az absztrakt szintaxisfa egy csomópontja.

A megvalósított absztrakt szintaxisfa reguláris és heterogén: „heterogén” mert a csomópontok különféle típusúak (külön leszármazott osztály valósítja meg például a feltételes utasítást és a kifejezést), és „reguláris” mert a csomópontokat egységes, homogén interface-en keresztül is lehet kezelni.

4.3.4. Szimbólumtábla

A szimbólumtáblát a `Context` osztály valósítja meg. Minden nevesített szemantikai egységet (típusokat, változókat, függvényeket) egy közös LeBlanc-Cook szimbólumtáblában tárol[lásd 18, p. 30], de heterogén interface-t biztosít.

4.3.5. Szemantikus elemzés

A `Grammar` egy szabályfüggvénye kérheti egy `Node` szemantikus ellenőrzését a rosszul elnevezett `ASTTypechecker` osztálytól. Az `ASTTypechecker` ellenőrzi az adott `Node` és leszármazottai típushelyességét, és megkísérli rezolválni a függvényhívásokat. A prototípus kezeli a túlterhelt függvényeket: a PLanG operátorai mind túlterhelt függvényként vannak megvalósítva.

4.3.6. Interpreter

Az elkészült, típushelyes absztrakt szintaxisfát az `ASTInterpreter` osztály segítségével tudjuk futtatni. Az `ASTInterpreter` közvetlenül a szintaxisfát bejárva hajtja végre a programot.

4.4. A tervezési döntések utólagos értékelése

Ebben a részben a 4.2 részben ismertetett tervezési döntéseket fogom értékelni.

Szép cél volt ugyan az, hogy minden PLanG program legyen érvényes XPLanG program is, de emiatt az XPLanGban is megjelennek a PLanG legnagyobb felhasználhatósági problémái, az alacsony szerepkifejező erejű kulcsszavak és a deklarációs lista. A változtatható lexikális elemek a kulcsszavak problémájára megoldást nyújtanak, de a deklarációs lista súlyos problémái megmaradtak.

Nem volt egészen szerencsés választás a Java sem: a korlátlan hordozhatóság szép cél ugyan, de hasonló eredményt érünk el, ha a népszerű operációs rendszerekre fordított bináris állományokat terjesztjük, az egzotikusabb rendszerekhez pedig biztosítjuk a fordítás lehetőségét.

Ezzel szemben gondot okozott, hogy a Java virtuális gép indulása sok időt vesz igénybe, így a Javában írt parancssoros fordítóprogram növeli a nyelv effektív viszkozitását. A Java határozott objektumorientáltsága is gyakran nehezítette a tisztán imperatív XPLanG fejlesztését, többször objektumelvű gondolatok szivárogtak bele a tervezésbe; ez különösen a típusrendszer alakításánál jelentett gondot. De ha már Javát használtam, sok gondot megspórolhattam volna külső könyvtárak szabadabb használatával: függőségkezelő eszközök használatával a külső könyvtárak kezelése kezdőknek sem nehéz feladat. Szintén érdemes lett volna kihasználni a Java 8 nyújtotta lehetőségeket.

Nem volt rossz döntés viszont saját elemzőt készíteni: nem került sokkal időbe, mint egy parsergenerátor használatát tisztességesen elsajátítani, és valóban nagyobb kontrollom volt így a hibaüzenetek generálásában, az elemzés logikai folyamatát is egyszerűsíteni tudtam, és rengeteget tanultam.

Az, hogy egyszerű fordítóprogram helyett egyfajta keretrendszert írtam komolyan megnehezítette és lassította a fejlesztést. Egyértelműen a túlzott generalizáció csapdájába estem, minden tekintetben célszerűbb lett volna egy csak az XPLanGot értelmező fordítóprogramot írni.

Hivatkozások jegyzéke

[1] Lövei L., magánlevelezés, 2015. feb.

[2] D. Knuth és L. Trabb Prado, „The early development of programming languages", Stanford University, tudományos jelentés STAN-CS-76-562, 1976.

[3] J. Backus, „The history of FORTRAN I, II, and III", *ACM SIGPLAN Notices*, vol. 13, no. 8, pp. 165–180, 1978. aug.

[4] F. P. Brooks Jr., *The Mythical Man-month*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995., ISBN: 0-201-83595-9.

[5] Vető I., Szoftvertechnológia előadás, 2015.

[6] Csörnyei Z., *Fordítóprogramok*, 2. kiadás. Typotex, 2006., ISBN: 9639548839.

[7] Fóthi Á., *Bevezetés a programozáshoz*, 3. kiadás. 2012. [Online]. URL: `http://people.inf.elte.hu/fa/pdf/konyv.pdf` (utolsó elérés 2015. 05. 04.).

[8] M. Felleisen, „On the expressive power of programming languages", in *Science of Computer Programming*, Springer-Verlag, 1990., pp. 134–151.

[9] P. J. Landin, „The mechanical evaluation of expressions", *The Computer Journal*, vol. 6, no. 4, pp. 308–320, 1964. jan.

[10] P. Graham, „Succintness is power", 2002. máj. [Online]. URL: `http://www.paulgraham.com/power.html` (utolsó elérés 2015. 05. 14.).

[11] S. McConnell, *Code Complete*, 2. kiadás. Redmond, WA, USA: Microsoft Press, 2004., ISBN: 9780735619678.

[12] L. McIver, „Syntactic and semantic issues in introductory programming education", PhD disszertáció, Monash University, 2001. jan.

[13] T. R. G. Green, „Cognitive dimensions of notations", in *People and Computers V*, Cambridge University Press, 1989., pp. 443–460.

[14] T. R. G. Green és A. Blackwell, „Cognitive dimensions of notations and other information artifacts: a tutorial", 1998. okt. [Online]. URL: `http://www.cl.cam.ac.uk/~afb21/CognitiveDimensions/CDtutorial.pdf` (utolsó elérés 2015. 05. 14.).

[15] L. Torczon és K. Cooper, *Engineering A Compiler*, 2. kiadás. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011., ISBN: 012088478X.

[16] B. Ford, „Parsing expression grammars: a recognition-based syntactic foundation", *ACM SIGPLAN Notices*, vol. 39, no. 1, pp. 111–122, 2004. jan.

[17] Balogh Zs., *Testreszabható programozási nyelv implementálása*, önálló laboratóriumi beszámoló, PPKE ITK, 2012.

[18] M. L. Scott, *Programming Language Pragmatics*, 3. kiadás. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2009., ISBN: 978-0-12-374514-9.