Írta:
GYIMÓTHY TIBOR
HAVASI FERENC
KISS ÁKOS

FORDÍTÓPROGRAMOK

Egyetemi tananyag

2011

LEKTORÁLTA: Dr. Aszalós László, Debreceni Egyetem Informatikai Kar Számítógéptudományi Tanszék

TÁMOGATÁS:

Készült a TÁMOP-4.1.2-08/1/A-2009-0008 számú, „Tananyagfejlesztés mérnök informatikus, programtervező informatikus és gazdaságinformatikus képzésekhez” című projekt keretében.

ISBN 978-963-279-503-4

KÉSZÜLT: a Typotex Kiadó gondozásában

FELELŐS VEZETŐ: Votisky Zsuzsa

AZ ELEKTRONIKUS KIADÁST ELŐKÉSZÍTETTE: Csépány Gergely László

KULCSSZAVAK:

fordítóprogramok, attribútum nyelvtanok, interpreterek, fordítóprogram-generáló rendszerek, röpfordítás.

ÖSSZEFOGLALÁS:
A fordítóprogramok feladata, hogy a különböző programozási nyelven írt programokat végrehajtható gépi kódú utasításokká transzformálja. A fordítási folyamat nagyon összetett, hiszen sok programozási nyelv létezik és a különböző processzorok gépi utasítás készlete is jelentősen eltérhet egymástól. Figyelembe véve a rendelkezésre álló magyar nyelvű szakirodalmat, ebben a jegyzetben nem törekedtünk egy átfogó, a fordítás minden fázisát érintő anyag elkészítésére, hanem három területet érintünk részletesebben.
A fordítási folyamat legjobban kidolgozott fázisainak a lexikális és a szintaktikus elemzés tekinthető. Hatékony algoritmusok és ezek megvalósítását támogató rendszerek készültek a lexikális és szintaktikus elemzők automatikus előállítására. A jegyzetben példákon keresztül bemutatjuk az ANTLR rendszert, amely segítségével formális nyelvtan alapú definíciók alapján gyakorlatban is használható minőségű elemzők generálhatók.
A szintaktikus elemzést követő fordítási fázis a szemantikus elemzés. Ennek feladata az olyan fordítási időben felderíthető problémák megoldása, amelyek a hagyományos szintaktikus elemzőkkel nehezen valósíthatók meg (ilyen például a típus kompatibilitás vagy a változók láthatósági kérdésének kezelése). A szemantikus elemzés elterjedt modellje az attribútum nyelvtan alapú fordítási modell. A jegyzetben ismertetjük az attribútum nyelvtanokat és a kapcsolódó attribútum kiértékelő stratégiákat, valamint példákat mutatunk be arra, hogyan használhatók az attribútum nyelvtanok fordítás idejű szemantikus problémák megoldására.

Tartalomjegyzék

1. Bevezetés **5**

2. A fordítóprogramok alapjai **7**
2.1. Áttekintés 7
2.1.1. Formális nyelvtanok és jelölésük 7
2.1.2. A FI halmaz 8
2.1.3. Az LL(k) elemzés 9
2.1.4. Feladatok 10
2.2. A fordítóprogramok szerkezete 10
2.3. Lexikális elemző 11
2.4. Szintaktikus elemző 11

3. Lexikális és szintaktikus elemző a gyakorlatban - ANTLR **13**
3.1. Jelölések használata 13
3.2. Az ANTLR telepítése 13
3.3. Fejlesztés ANTLRWorks környezetben 14
3.4. Az ANTLR nyelvtanfájl felépítése 16
3.5. Lexikális elemző megvalósítása 16
3.5.1. Feladatok 17
3.6. Szintaktikus elemző megvalósítása 17
3.6.1. Feladatok 20
3.7. Akciók beépítése 20
3.8. Paraméterek kezelése 21
3.9. Kifejezés értékének kiszámítása szemantikus akciókkal 21
3.9.1. Feladatok 22
3.9.2. ANTLR által generált kód 23
3.10. AST építés 25
3.10.1. Feladatok 30

4. Attribútumos fordítás **32**
4.1. Attribútum nyelvtanok definíciója 32
4.2. Attribútum kiértékelés 36
4.3. Attribútum nyelvtan bináris törtszám értékének kiszámítására 38
4.4. Többmenetes attribútum kiértékelő 41

4.5. Típus kompatibilitás ellenőrzés . . . 46
4.6. Rendezett attribútum nyelvtanok . . . 49
4.6.1. OAG teszt . . . 49
4.6.2. OAG vizit-sorozat számítása . . . 51
4.7. Attribútum nyelvtanok osztályozása . . . 53
4.8. Közbülső kód generálása . . . 55
4.9. Szimbólumtábla kezelése . . . 56
4.10. Feladatok . . . 58

5. Interpretált kód és röpfordítás **60**
5.1. Fa alapú interpreter . . . 61
5.2. Bájtkód alapú interpreter . . . 61
5.3. Szálvezérelt interpreterek . . . 63
5.3.1. Tokenvezérelt interpreter . . . 64
5.3.2. Direkt vezérelt interpreter . . . 65
5.3.3. Környezetvezérelt interpreter . . . 66
5.4. Röpfordítás . . . 68
5.5. Példamegvalósítás . . . 72
5.5.1. Bájtkódgenerálás és -interpretálás megvalósítása Java nyelven . . . 72
5.5.2. Bájtkód alapú és szálvezérelt interpreterek megvalósítása C nyelven . . . 77
5.5.3. Röpfordító megvalósítása C nyelven Intel Linux platformra . . . 83
5.6. Összegzés . . . 86
5.7. Feladatok . . . 86

Irodalomjegyzék **86**

1. fejezet

Bevezetés

A fordítóprogramok feladata, hogy a különböző programozási nyelven írt programokat végrehajtható gépi kódú utasításokká transzformálja. A fordítási folyamat nagyon összetett, hiszen sok programozási nyelv létezik és a különböző processzorok gépi utasítás készlete is jelentősen eltérhet egymástól. Nem véletlen, hogy a számítógépek elterjedésének időszakában (50-es évek) a fordítóprogramok hatékony megvalósítását tekintették az egyik legbonyolultabb számítástudományi és szoftvertechnológiai problémának. Sok kiváló informatikai szakember kezdett el dolgozni ezen a területen, aminek eredményeként létrejöttek olyan algoritmusok, amelyek felhasználásával nagyméretű programokra is sikerült végrehajtási időben és tárméretben is elfogadható kódot generálni. Ezt nyilvánvalóan az is elősegítette, hogy a számítógépek műszaki teljesítménye rohamosan növekedett, pl. nem jelentett problémát a megnövekedett tárigény.

A beágyazott rendszerek terjedésével ez a kényelmes helyzet megváltozott. Noha az ezekben az eszközökben alkalmazott processzorok teljesítménye is rohamosan növekszik, de jelentősen elmarad az asztali gépek nyújtotta lehetőségektől.

Ez a trend újból óriási kihívást jelent a fordítóprogramokkal foglalkozó szakemberek számára. Meg kell találni azokat a megoldásokat, amelyekkel a gyakran nagyon összetett alkalmazások is hatékonyan végrehajthatók ezeken a beágyazott eszközökön. Ez általában nagyon agresszív optimalizálási megoldásokat igényel mind végrehajtási idő, mind pedig tárméret területén. A beágyazott rendszerek esetében egy sajátos problémaként megjelent egy harmadik optimalizálási szempont is, nevezetesen az energia-felhasználás kérdése. Mivel ezek az eszközök általában korlátos energiaforrással rendelkeznek, ezért a generált kód energia-felhasználását is optimalizálni kell.

A fordítóprogramokkal kapcsolatos eredményekről számos kiváló szakkönyv készült. Az angol nyelvű szakkönyvek közül mindenképpen ki kell emelni az A. V. Aho, R. Sethi és J. D. Ullman által írt alap könyvet, amit szokás „Dragon” könyvként is említeni [2]. Ez a könyv szisztematikusan tárgyalja a fordítás különböző fázisait, érinti a kapcsolódó elméleti eredményeket, de alapvetően gyakorlati megközelítéssel dolgozza fel a területet. Szintén kiváló angol nyelvű szakkönyv S.S. Muchnick munkája [8]. Ez a könyv a fordítóprogramok legösszetettebb problémájának tekinthető kódoptimalizálás alapjait képező programanalizálási módszerekkel foglalkozik.

Magyar nyelven is elérhető néhány kiváló munka a fordítóprogramok területén. A

fordítóprogramok szintaktikus elemzésének elméleti hátterét ismerteti Fülöp Z. egyetemi jegyzete [5]. Csörnyei Z. 2006-ban publikált Fordítóprogramok című könyve [4] áttekintést ad a teljes fordítási folyamatról sikeresen ötvözve a szükséges elméleti háttér és a gyakorlati problémák bemutatását. A szintaktikus elemzés gyakorlati oktatásához készített hasznos példatárat Aszalós L. és Herendi T. [3].

Figyelembe véve a rendelkezésre álló magyar nyelvű szakirodalmat, ebben a jegyzetben nem törekedtünk egy átfogó, a fordítás minden fázisát érintő anyag elkészítésére. Erre a jegyzet méret korlátai miatt sem lett volna lehetőség. A jegyzetben egy rövid bevezető fejezet (2. fejezet) után három területet érintünk részletesebben.

A fordítási folyamat legjobban kidolgozott fázisainak a lexikális és a szintaktikus elemzés tekinthető. Hatékony algoritmusok és ezek megvalósítását támogató rendszerek készültek a lexikális és szintaktikus elemzők automatikus előállítására. A jegyzet 3. fejezetében példákon keresztül bemutatjuk az ANTLR rendszert [1], amely segítségével formális nyelvtan alapú definíciók alapján gyakorlatban is használható minőségű elemzők generálhatók.

A szintaktikus elemzést követő fordítási fázis a szemantikus elemzés. Ennek feladata az olyan fordítási időben felderíthető problémák megoldása, amelyek a hagyományos szintaktikus elemzőkkel nehezen valósíthatók meg. Ilyen például a típus kompatibilitás vagy a változók láthatósági kérdésének kezelése. A szemantikus elemzés elterjedt modellje az attribútum nyelvtan alapú fordítási modell. A 4. fejezetben ismertetjük az attribútum nyelvtanokat és a kapcsolódó attribútum kiértékelő stratégiákat, valamint példákat mutatunk be arra, hogyan használhatók az attribútum nyelvtanok fordítás idejű szemantikus problémák megoldására.

A beágyazott rendszerek elterjedésével egyre nagyobb teret kaptak az értelmező (interpreter) alapú program végrehajtási megoldások. Ezek lényege, hogy a forráskódból nem generálunk gépi kódú utasításokat, hanem egy értelmező program segítségével hajtjuk végre az utasításokat. (Megjegyezzük, hogy általában a forrásprogramból készül egy közbülső kódnak nevezett reprezentáció és az értelmező program ezt a kódot hajtja végre.) Az 5. fejezetben ismertetjük a különböző értelmezési technikákat és bemutatunk optimalizálási megoldásokat.

Ez a jegyzet az egyetemi informatikai képzés BSc és MSc szakjain is használható a fordítóprogramokkal kapcsolatos kurzusokban. A jegyzet elsajátításához alapfokú ismeretek szükségesek a formális nyelvek elemzéséről és programozási tapasztalat C és Java nyelveken.

2. fejezet

A fordítóprogramok alapjai

2.1. Áttekintés

Ebben a jegyzetben az un. szintaxis-vezérelt fordítási megoldással foglalkozunk. Ennek lényege, hogy a fordítandó programozási nyelv szerkezete megadható egy *környezet-független nyelvtannal*. Ezután az adott nyelv szerint, minden programra a nyelvtan alapján készíthető egy *elemzési fa* (derivációs fa). A formális nyelvtanokkal és elemzésükkel ebben a jegyzetben csak áttekintés szintjén foglalkozunk. A témába mélyebb betekintést a [4] és [5] jegyzetek adnak, gyakorlási lehetőségre pedig [3] jegyzet nyújt kiváló lehetőséget.

2.1.1. Formális nyelvtanok és jelölésük

Egy G formális nyelvtan egy olyan $G = (N,T,S,P)$ négyes, ahol N a nemterminális szimbólumok, T a terminális szimbólumok halmaza, amelyeket szoktak *tokeneknek* is nevezni. P az $\alpha \to \beta$ alakú átírási szabályok halmaza, az $S \in N$ pedig egy kitüntetett nemterminális, a kezdőszimbólum. A továbbiakban a jelölésekben a következő konvenciókat követjük:

a terminális szimbólumokat vagy a token nevével jelöljük kis betűkkel megnevezve (pl. szam), vagy az általa reprezentált karaktersorozatot egyszeres idézőjelek között (pl. '+') Ezek lényegében az adott nyelv speciális karakterei (pl. ',' ':' ';'), kulcsszavai (pl. 'if', 'for') illetve logikailag összetartozó karakter sorozatai (pl. azonosítók, konstansok).

a nemterminális szimbólumokat nagy kezdőbetűkkel nevezzük el (pl. Tag).

ezek tetszőleges sorozatát pedig a görög ABC betűivel jelöljük (pl. α).

Egy formális nyelvtan *környezetfüggetlen*, ha a szabályai $A \to \beta$ alakúak, azaz a bal oldalon pontosan egy darab nemterminális található. A legtöbb elemző algoritmus eleve ilyen nyelvtanokkal dolgozik, mert elemzésük általánosságban véve sokkal könnyebb feladat, mint a környezetfüggő nyelvtanoké.

Egy környezetfüggetlen formális nyelvtant *balrekurzívnak* nevezünk, ha van benne olyan A nemterminális, amelyből levezethető (valamennyi szabály egymás utáni alkalmazásával

megkapható) egy $A\alpha$ alakú kifejezés. Az ilyen típusú szabályokat az általunk használt elemzők nem bírják elemezni, viszont tetszőleges nyelvtan transzformálható vele ekvivalens, nem balrekurzív nyelvtanná.

Az elemzés során általában felépülő *elemzési fa* gyökerében a kezdőszimbólum van, a leveleit (amelyek terminális szimbólumok) összeolvasva megkapjuk az elemezendő inputot, a csúcspontokban pedig nemterminálisok ülnek a levezetésnek megfelelően „összekötve".

Példa: a 2.1 ábrán látható egy egyszerű értékadó utasítás formális nyelvtana.

```
1.  Utasitas → azonosito ':=' Kifejezes
2.  Kifejezes → Kifejezes '+' Tag
3.  Kifejezes → Kifejezes '−' Tag
4.  Kifejezes → Tag
5.  Tag → Tag '*' Tenyezo
6.  Tag → Tag '/' Tenyezo
7.  Tag → Tenyezo
8.  Tenyezo → '(' Kifejezes ')'
9.  Tenyezo → szam
10. Tenyezo → azonosito
```

2.1. ábra. Értékadó utasítás környezet-független nyelvtana

A nyelvtan

- nemterminális szimbólumai: *Utasitas, Kifejezes, Tag, Tenyezo*
- terminális szimbólumai: *':=', '+' , '-', '*' , '/ ', '(' , ')' , azonosito, szam*
- kezdő szimbóluma: Utasitas

Ezt a nyelvtant felhasználva elkészíthetjük az alábbi értékadó utasítás

*alfa := beta + 8 * (gamma - delta)*

elemzési fáját.

A 2.2. ábrán láthatjuk, hogy az elemzési fában az *alfa, beta, gamma, delta* szimbólumok helyett mindenütt az *azonosito* a 8-as konstans helyett pedig a *szam* terminális (token) szimbólum szerepel. A *lexikális elemző* feladata, hogy ezeket a transzformációkat megvalósítsa.

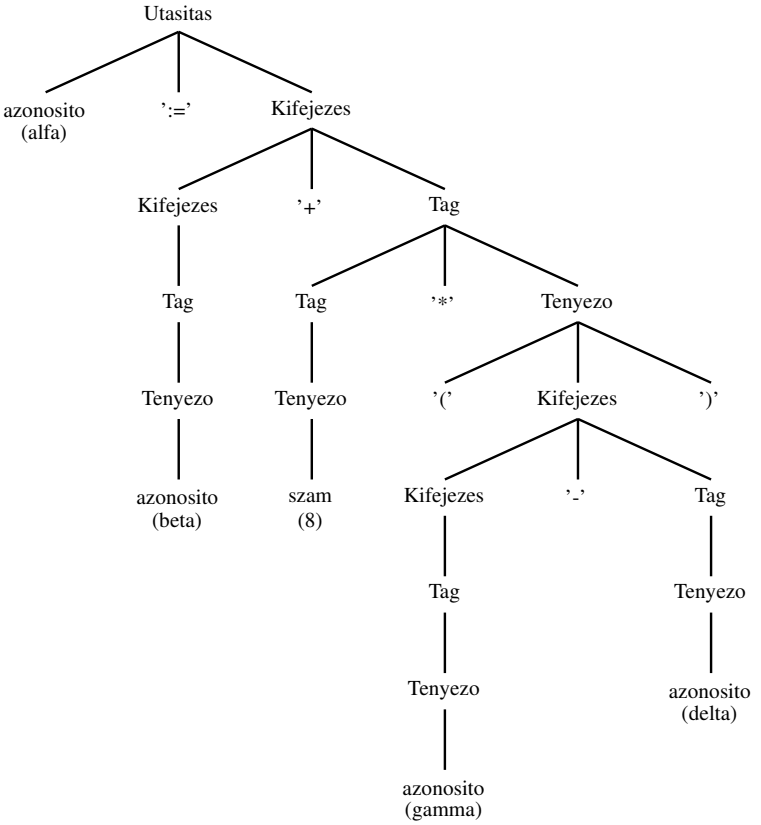
2.1.2. A FI halmaz

Az elemzéshez szükséges egyik segédfogalom a $FI_k()$ halmaz, amely értelmezhető tetszőleges terminálisokat és/vagy nemterminálosakat tartalmazó kifejezésekre, és belőle levezethető terminális szimbólumok első (first) k szimbólumainak halmazát jelenti, azaz:

terminális szavaknál ez a halmaz az első k szimbólumuk – rövidebb esetén a teljes szó.

nemterminális szavaknál azon terminális szavak $FI_k()$-jainak halmaza, amelyek ebből levezethetőek.

A 2.1 nyelvtan esetében a FI_1(Tenyezo) halmaz: '(',szam,azonosito.



2.2. ábra. Az alfa := beta + 8 * (gamma - delta) *utasítás elemzési fája*

2.1.3. Az LL(k) elemzés

A LL(k) elemzés során a levezetés a kezdőszimbólumból indul. Bal oldali levezetés használunk (jelölése $\Rightarrow_l$), azaz minden egyes lépésnél kicseréljük a legbaloldalibb nemterminálist – legyen most ez A – valamelyik szabály alapján. Ha egyetlen $A \to \beta$ alakú szabályunk van, akkor ez a csere egyértelmű. Ha több ilyen van, akkor pedig az inputon lévő következő k darab szimbólum alapján döntünk, hogy melyik szabályt alkalmazzuk. A LL(k) nyelvtanok olyan nyelvtanok, amelyeknél ez az információ bármilyen input esetén is elegendő az egyértelmű döntéshez.

Formálisan megfogalmazva: legyen k egy pozitív egész szám, $\alpha \Rightarrow^* \beta$ jelentse azt, hogy α-ból β levezethető, $\alpha \Rightarrow_l^* \beta$ pedig azt, hogy α-ból β baloldali levezetéssel (mindig csak a bal oldali nemterminális kicserélésével) levezethető. Ekkor egy nem balrekurzív nyelvtan akkor LL(k) nyelvtan, ha valahányszor teljesülnek a

- $S \Rightarrow_l^* \omega A \alpha \Rightarrow \omega \beta \alpha \Rightarrow^* \omega x$
- $S \Rightarrow_l^* \omega A \alpha \Rightarrow \omega \gamma \alpha \Rightarrow^* \omega y$
- $FI_k(x) = FI_k(y)$

feltételek, akkor $\beta = \gamma$-nak is teljesülnie kell.

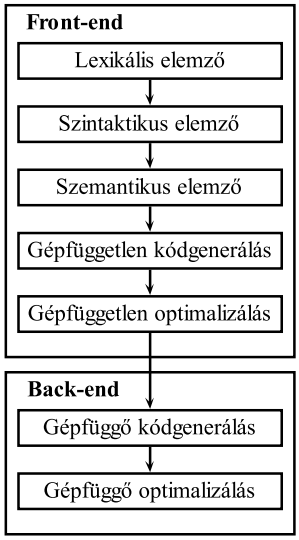
2.1.4. Feladatok

Tekintsük az alábbi nyelvtant:

```
S –> + A A | * A A

A –> '1' | '2' | S
```

- Számoljuk ki a $FI_1(S)$ és $FI_1(A)$ halmazokat!
- LL(k) nyelvtan-e a fenti nyelvtan, és ha igen, mekkora k-ra?
- Rajzoljuk föl az elemzési fáját az $++1\ 2*2\ 2$ inputra!
- Melyik állítás igaz?
  1. Minden LL(2) nyelvtan egyben LL(1) nyelvtan is.
  2. Minden LL(1) nyelvtan egyben LL(2) nyelvtan is.

2.2. A fordítóprogramok szerkezete



2.3. ábra. Fordítóprogramok fázisai

A fordítóprogramok főbb fázisai a 2.3. ábrán látható. A gép független fázisokat szokás *front-endnek* a gép függő részeket pedig *back-endnek* nevezni. A fordítóprogram front-end része analizálja a forrásprogramot, felépíti az elemzési fát, elvégzi a szükséges szemantikus ellenőrzéseket és egy gép független *közbülső kódot* generál. A közbülső kód assembly jellegű

kód, ami azonban független a konkrét gépi architektúráktól. Nagy előnye, hogy közbülső kódból már könnyű előállítani gépi kódú utasításokat konkrét architektúrákra. Lehetőség van a generált közbülső kód optimalizálására is pl. redundáns számítások kiszűrésével. A fordítóprogram back-end része a konkrét gépi architektúrára készíti el az áthelyezhető gépi kódú vagy pedig assembly szintű programot. Itt már például fontos szempont a regiszter allokáció optimalizálása. A front-end és back-end részek szétválasztása nagyon előnyös, mivel így a back-end részek cseréjével egyszerűen készíthető gépi kód különböző platformokra.

A fordítóprogramok fontos része a *szimbólumtábla kezelés* illetve a fordítás különböző fázisaiban észlelt *hibák kezelése*. A szimbólumtáblában a program azonosítóiról tárolunk információkat pl. egy változóról a típusát, az allokált memória címét, érvényességi tartományát (scope). A szimbólumtáblát a fordítás minden fázisában használjuk. Fordítási hibák leggyakrabban a fordítóprogram analízis fázisaiban (lexikális, szintaktikus, szemantikus elemzés) keletkeznek. Tipikus szintaktikus hiba, ha egy utasítás szerkezetét elrontjuk, például elhagyunk egy zárójelt. Szemantikus hiba lehet például az, ha egy értékadó utasítás baloldali változójának típusa nem kompatibilis a jobboldali kifejezés típusával. A fordítási hibák kezelésénél fontos szempont, hogy minél több hibát detektáljunk egy fordítás során.

2.3. Lexikális elemző

A fordítóprogram bemenete egy karakter sorozat, például egy program forráskódja. A karakter sorozat feldolgozásának első lépése a *lexikális elemző*, amely a következő feladatokat hivatott ellátni:

- Az összetartozó karaktereket egy *tokenné* összevonja: tokenek lehetnek egy adott programozási nyelv kulcsszavai (pl. 'if', 'while', ...), vagy számok ('123', '1.23', ...), stb. Az olyan karakterek, amelyek önállóan bírnak jelentéssel, (pl. egy '+' jel) önálló tokenek lesznek.
- Kiszűri azokat a karaktereket, amelyek az elemzés szempontjából lényegtelenek: tipikusan ilyenek a kommentek, újsor vagy white space karakterek (szóköz, tabulátor, ...).
- Felépít egy táblát (sztring tábla) a felismert tokenekre, amelyben bizonyos információkat tárol róluk (pl. az azonosító neve).

A lexikális elemző tehát előfeldolgozást végez az inputon, és egy token sorozatot ad át a szintaktikus elemzőnek. A token saját típusán kívül tárolja az általa képviselt input szövegrészletet (egy egész szám esetében pl. '12'), a forráskódban elfoglalt helyét (hányadik sorban és oszlopban volt volt) is. A szintaktikus elemző a további feldolgozáshoz a tokeneket és a sztring táblát kapja meg.

2.4. Szintaktikus elemző

A feldolgozás következő lépése a *szintaktikus elemző*, melynek feladata:

- Ellenőrzi az input szintaktikáját, hogy megfelel-e az elvártaknak - amelyet környezetfüggetlen nyelvtan formájában fogalmazunk meg számára.
- A szintaktika (környezetfüggetlen nyelvtan) alapján készítse elő a további feldolgozást egy új belső reprezentáció felépítésével, amely elméleti szempontból a elemzési fával mutat rokonságot, sok fordítóprogram esetében AST-nek, azaz absztrakt szintaxis fának hívnak.

A fordítóprogram következő lépcsője, a *szemantikus elemző* ezen a belső reprezentáción fog majd tovább dolgozni.

Látni fogjuk, hogy egyszerűbb esetekben az AST felépítése nélkül, már az elemzés közben elvégezhető a kívánt feladat.

3. fejezet

Lexikális és szintaktikus elemző a gyakorlatban - ANTLR

Egy fordítóprogram implementálása sok olyan feladattal jár, amely gépies és igen időigényes. Ez az oka annak, hogy a gyakorlatban ritkán fordul elő, hogy valaki közvetlenül írjon elemzőt, sokkal inkább használ erre a célra egy fordítóprogram generáló rendszert, amely a gépies részfeladatok elvégzése mellett általában számos kényelmi szolgáltatással teszi még hatékonyabbá a fejlesztést.

Egyik ilyen szabadon felhasználható, nyílt forráskódú, platformfüggetlen LL(k) fordítóprogram generátor az ANTLR [1] (ANother Tool for Language Recognition). Kifejezetten ehhez a rendszerhez fejlesztették az ANTLWorks[1] környezetet, amely még kényelmesebbé és hatékonyabbá teszi a fejlesztést, különösen a fordítóprogramok világával újonnan ismerkedők számára.

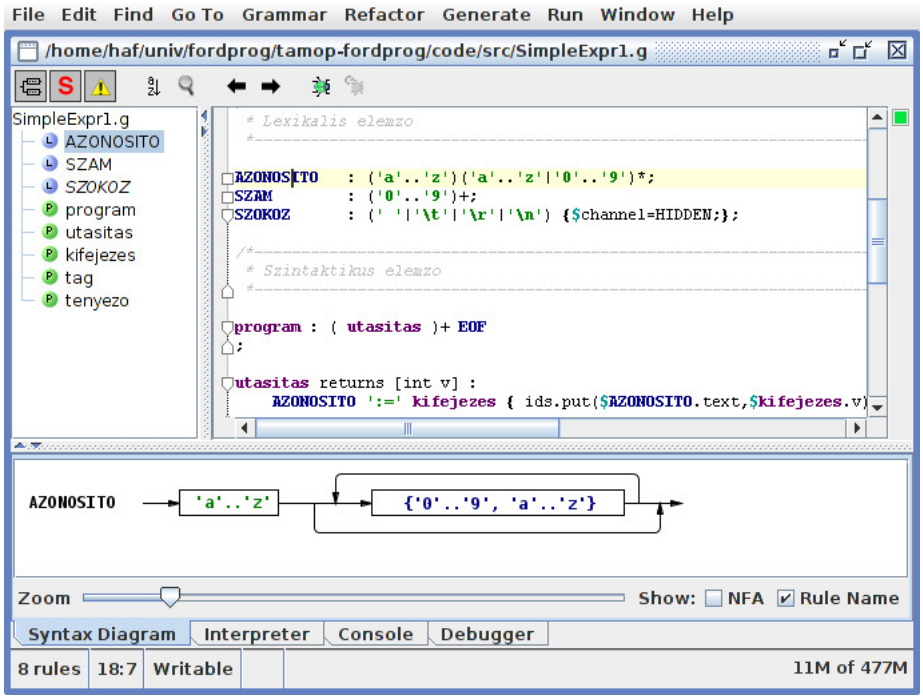
3.1. Jelölések használata

Az előző fejezetben megismertük az a fordítóprogramok elméletnél megszokottól jelölési konvenciót. Ebben a fejezetben egy ettől különböző konvenciót, az ANTLR konvencióját fogjuk használni: a tokenek avagy terminálisok nevének nagybetűvel kell kezdődnie, a nemterminálisokénak pedig kisbetűvel, és egyikben sem használható ékezetes karakter.

3.2. Az ANTLR telepítése

Az ANTLR Java alapú rendszer, így használatának előfeltétele, hogy telepítve legyen Java futtató környezet. Az ANTLR és az ANTLRWorks szabadon letölthető a következő weboldalról (http://www.antlr.org/works/index.html). Ez utóbbi egy jar fájlba van csomagolva, amely magába foglalja az ANTLR-t és az ANTLRWorks-öt is. A jegyzet írásának idejében a fájl neve anlrworks-1.4.2.jar, amely a következő paranccsal indítható:

```
java -jar anlrworks-1.4.2.jar
```



3.1. ábra. Az ANTLRWorks működés közben: bal oldalt a terminális és nemterminális szimbólumok listája, jobb oldalt a nyelvtanfájl, lent pedig az aktuális szabály szintaxis diagramja

Az ANTLR számára a generálandó fordítóprogramot egy (tipikusan .g kiterjesztésű) nyelvtanfájl formájában kell megadnunk, amelyből alapértelmezésben Java[2] forráskódot gyárt, legenerálva a lexikális (Lexer) és szintaktikus elemzőt (Parser).

Parancssort használva a nyelvtanfájlból a következő módon indíthatjuk a generálást:

```
java -cp antlrworks-1.4.2.jar org.antlr.Tool SajatNyelvtan.g
```

A parancssoros generálás további részleteit nem tárgyaljuk, példánkban az ANTLWorks-szel fogunk dolgozni, ahol a fenti parancsot a „Generate code” menüpontra kattintással hívhatjuk meg.

3.3. Fejlesztés ANTLRWorks környezetben

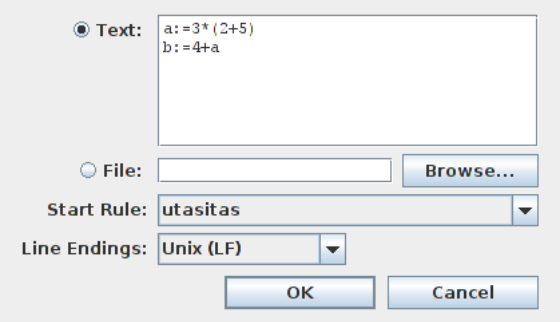
Az ANTLWorks környezet a nyelvtanfájlok szerkesztését több oldalról támogatja. Egyrészt a szövegszerkesztője beírás közben elemzi a nyelvtan fájlt, jelezve a hibás részeket, és a beírt szabályokat azonnal mutatja diagram formájában is.

Tartalmaz egy beépített hibakereső részt is, amelyet a „Run” menüpont „Debug” almenüjével indíthatunk - 3.2 ábra. Itt adhatjuk meg azt az inputot, amire futtatni szeretnénk, és itt

[1] Eclipse-be beépülő modul is elérhető hozzá.

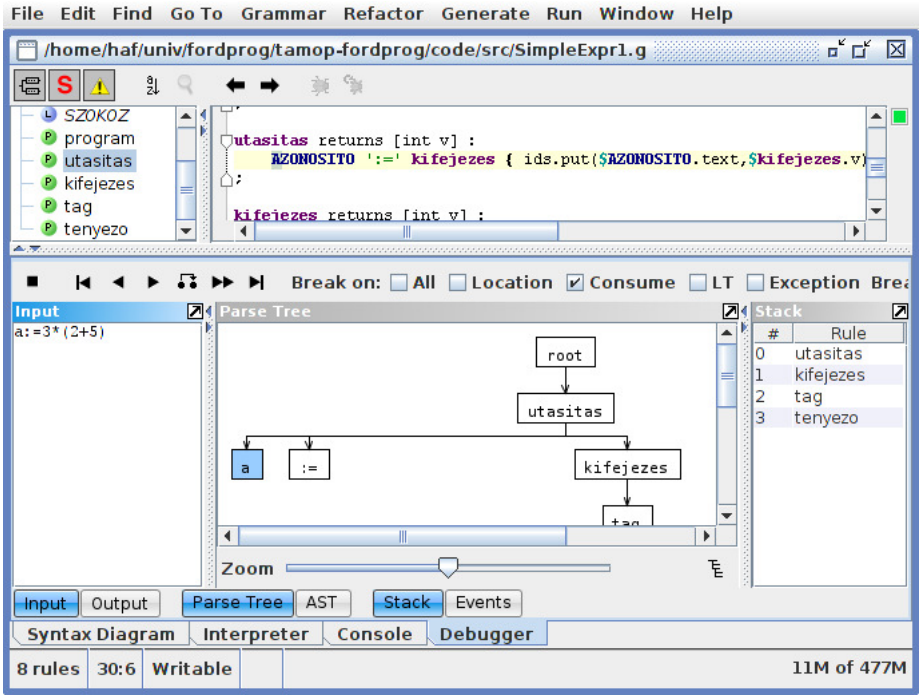
[2] Támogat más célnyelveket is, pl. C/C++, C#, Python, Ruby.

állíthatjuk be a kezdőszimbólumot, amivel az ANTLRWorks tesztelő része kezdeni fogja az elemzést.



3.2. ábra. A hibakeresés előtt meg kell adnunk a kezdőszimbólumot és az elemzendő bemenetet

Amint a hibakeresés elindult – 3.3 ábra – , egyesével lépkedhetünk a tokeneken, nézve melyik szabály hívódik meg, milyen elemzési fa épül, és mi íródik a konzolra.



3.3. ábra. Hibakeresés közben egyszerre látjuk az inputot, az azt elemző szabályt és a készülő elemzési fát

3.4. Az ANTLR nyelvtanfájl felépítése

Egy ANTLR nyelvtan fájl egyszerűsített[3] szerkezete:

```
grammar NyelvtanNeve;

<Lexikális elemző szabályai>

<Szintaktikus elemző szabályai>
```

3.5. Lexikális elemző megvalósítása

A lexikális elemzőnk a lexikális szabályokat az alábbi formában várja:

```
TOKEN_NEVE : token_definíciója { akció; };
```

A token nevének és definíciójának megadása kötelező, az akció rész opcionális. Fontos, hogy a token nevének nagybetűvel kell kezdődnie. A definícióban azt kell megadni egy reguláris kifejezés formájában, hogy az adott token milyen karaktersorozatra illeszkedjen. Ezt láthatjuk a 3.4 ábrán.

Minta	Amire illeszkedik
'a'	az *a* karakterre
'abcde'	az abcde karaktersorozatra
'.'	egy tetszőleges karakterre
~A	olyan karakterre illeszkedik, amire A nem
A | B	arra, amire vagy A vagy B illeszkedik
'a'..'z'	egy karakterre, amely kódja a két karakter közé esik
A*	0 vagy több A-ra illeszkedik
A+	1 vagy több A-ra illeszkedik
A?	0 vagy 1 A-ra illeszkedik

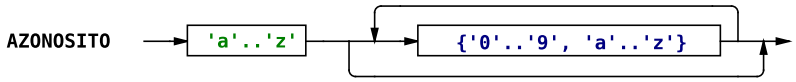
3.4. ábra. ANTLR-ben használható reguláris kifejezések

Például nézzük meg olyan AZONOSITO token megadását, amely kis betűvel kezdődik, és utána valamennyi kis betűvel vagy számjeggyel folytatódik. Ennek a szintaxis diagramja a 3.5 ábrán látható.

```
AZONOSITO : ('a'..'z')('a'..'z'|'0'..'9')*;
```

A token definíciója mögé opcionálisan írható akció egy kapcsos zárójelek közé írt programkód részlet. Leggyakrabban ezt a lexikális elemző vezérlésére használják, amelyek közül mi azt fogjuk most ismertetni, hogy hogyan vehető rá, hogy az adott tokent a lexikális elemző ne adja át a szintaktikus elemzőnek. A gyakorlatban ezt például a megjegyzések

[3] A jegyzet több ANTLR funkcióra szándékosan nem tér ki, hogy az kezdő olvasó számára is könnyebben érthető legyen. Az összes elérhető funkció leírását az olvasó megtalálja az ANTLR weboldalán.



3.5. ábra. Az AZONOSITO token szintaxis diagrammja

vagy whitespace-ek kezelésére használhatjuk. Az első megoldás erre a problémára a skip() függvény meghívása, amely az adott tokent megsemmisíti. A másik, ha a tokent átsoroljuk a rejtett csatornához a „$channel=HIDDEN;” kódrészlettel. Ekkor a tokent reprezentáló objektum megmarad, később akár elővehető, csak a szintaktikus elemző nem kapja meg. Ez utóbbi módszer a preferáltabb, mi is ezt fogjuk használni.

A következő példában C++ szerű kommenteket illetve újsor karaktereket szűrünk ki.

```
UJSOR    : '\n'          {$channel=HIDDEN;};
KOMMENT : '//' .* '\n' {$channel=HIDDEN;};
```

3.5.1. Feladatok

- Definiáljunk egy olyan lexikális elemzőt, amely egész számokat és azokat elválasztó szóközt vagy újsor karaktereket ismer föl, ez utóbbiakat viszont nem küldi tovább a szintaktikus elemzőnek, csak az egész számokat.
- Definiáljunk egy olyan szám tokent, amely ráillik az egész számokra is, és a tizedes törtekre is! (Pl. 12, 13.45, ...)
- Definiáljunk egy olyan tokent, ami a C szerű kommentet szűri ki az inputból!

3.6. Szintaktikus elemző megvalósítása

A szintaktikus elemző szabályainak megadása a következő formában történik:

```
nemterminális_neve : első alternatíva
                   | második alternatíva
                     ...
                   | utolsó alternatíva
                   ;
```

A nemterminális neve kis betűvel kell, hogy kezdődjön. Az egyes alternatívákban pedig a következő elemek szerepelhetnek egymástól szóközzel elválasztva:

- token nevek (pl. SZAM)
- token definíciók (pl. ’+’)
- nemterminális nevek (pl. tenyezo)

A rövidebb leírás mellett a fentiek kombinálhatók a lexikális elemzőnél már megismert reguláris kifejezések: +, *, ?, | jelek, a jelek csoportosítása pedig zárójelekkel irányítható.

Például tekintsük a korábbi 2.1.4-ban lévő nyelvtan ANTL-beli leírását:

```
s : '+' a a
  | '*' b b
  ;

a : '1'
  | '2'
  | '(' s ')'
  ;
```

A fenti példa egyszerű volt, mert láthatóan 1 karakter előrenézésével meg lehet állapítani, hogy melyik alszabályt válassza az elemző (pl. s-nél '+' esetén az elsőt, '*' esetén a másodikat), így az ANTLR is könnyen megbirkózik majd a feladattal.

Bonyolultabb a helyzet, ha nem LL(k) nyelvtannal van dolgunk, mint ahogyan az a 2.1 ábrán látható formális nyelvtan esetében is így van. Ha azt ANTLR formára hozzuk, a következőt kapjuk:

```
utasitas: azonosito ':=' kifejezes
        ;

kifejezes: kifejezes '+' tag
         | kifejezes '-' tag
         | tag
         ;

tag: tag '*' tenyezo
   | tag '/' tenyezo
   | tenyezo
   ;

tenyezo: '(' kifejezes ')'
       | SZAM
       | AZONOSITO
       ;
```

Látható, hogy a kifejezes és a tag nemterminális is balrekurzív, emiatt nem boldogul el vele egy LL(k) elemző. Ennek megszüntetése a nyelvtan kis átalakítással megoldható:

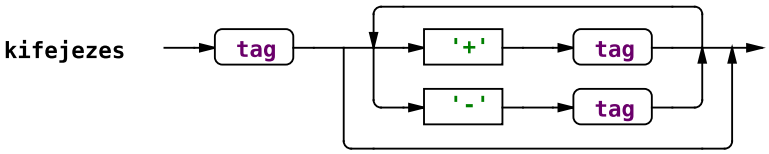
```
utasitas: azonosito ':=' kifejezes
        ;

kifejezes: tag kifejezes2
         ;

kifejezes2: '+' tag
          | '-' tag
          |
          ;

tag: tenyezo tag2
```

```
    ;

tag2: '*' tenyezo
    | '/' tenyezo
    |
    ;

tenyezo: '(' kifejezes ')'
       | SZAM
       | AZONOSITO
       ;
```

Az átalakítás után láthatóan LL(1) nyelvtant kaptunk, hiszen minden alternatívánál egy token előrenézéssel – még fejben számolva is – dönteni tudunk, hogy melyik alternatívát válasszuk.

Ha használjuk az ANTLR adta lehetőséget, hogy reguláris kifejezést is használhatunk a szabályainkban, akkor a fenti nyelvtant egy rövidebb formában is leírhatjuk:

```
kifejezes:
    tag
    (  '+' tag
     | '-' tag
    )*
 ;

tag:
    tenyezo
    (  '*' tenyezo
     | '/' tenyezo
    )*
 ;

tenyezo:
    SZAM
    | '(' kifejezes ')'
    | AZONOSITO
;
```



3.6. ábra. Az kifejezes szabály szintaxis diagramja

Az átalakított nyelvtan szintaxis diagramján (amelyet az ANTLWorks kirajzol nekünk, 3.6 ábra) látszik, hogy most már minden alszabályi döntést az elemző egy token előrenézésével el tud dönteni.

3.6.1. Feladatok

- Egészítsük ki a nyelvtant a tenyezo kibővítésével úgy, hogy a $\sin(x)$ is használható legyen, ahol x egy tetszőleges részkifejezés.
- Oldjuk meg, hogy a nyelvtan hatványozni is tudjon: *a^b* formában, ahol *a* és *b* tetszőleges részkifejezések. Figyelj arra, hogy a hatványozás a szorzás-osztásnál magasabb, de a zárójelnél alacsonyabb prioritású.

3.7. Akciók beépítése

Az ANTLR a szintaktikus elemzőből egy Java osztályt fog generálni, ahol minden nemterminális egy-egy metódus lesz. A fenti példában a Parser osztálynak lesz egy kifejezes, egy tag és egy tenyezo metódusa, melyek feladata, hogy az inputról beolvassák önmagukat, azaz leelemezzék az input rájuk eső részét. Ezekbe a metódusokba szúrhatunk be tetszőleges kódrészletet, ha azt { } jelek közé beírjuk. Lássunk arra példát, hogy írunk ki valamit a konzolra az után, amikor a tenyezo nemterminális elemzése közben a SZAM tokent már fölismertük.

```
tenyezo : SZAM {System.out.println("Szám!");}
        ;
```

A szemantikus akciókban lehetőség van a már elemzett tokenek vagy nemterminálisok adatainak az elérésére is. Az adott token vagy nemterminális objektumára úgy tudunk hivatkozni, hogy annak neve elé egy $ jelet teszünk. Ha ez a név nem egyértelmű, akkor címkét adhatunk a tokennek vagy nemterminálisnak úgy, hogy a címke nevét a token vagy nemterminális elé írjuk = jellel elválasztva attól. A token vagy nemterminális objektumának adattagjai közül számunkra az alábbiak az érdekesek:

text : a token vagy nemterminálisnak megfelelő input szövegrészlet

int : a fenti szövegrészlet egész számmá konvertálva - ha az lehetséges

line : azt mondja meg, hogy melyik sorban van az inputon belül az adott token vagy nemterminális

Lássunk egy példát a használatra, ahol megcímkézzük mindkét SZAM tokent, és mindkettőnek az *int* adattagjára hivatkozunk:

```
osszeg : n1=SZAM '+' n2=SZAM
              {int osszeg = $n1.int+$n2.int;
               System.out.println("Az␣összeg:␣"+osszeg);}
       ;
```

Van néhány különleges hely, ahova még be tudunk szúrni kódrészletet:

@members : ezt az akciót minden szabályon kívülre kell elhelyeznünk, és amit ide írunk, azt az ANTLR beszúrja az elemző osztály definíciójának törzsébe. Ezzel például új adattagokat vagy metódusokat is föl tudunk venni az elemző osztályunkba.

@init : ezt egy szabály neve után, még a „:” karakter elé lehet elhelyezni, és a bele írt kódrészlet a szabály metódusának legelejére fog bemásolódni.

@after : hasonló az előzőhöz, csak ez pedig a metódus végére másolódik.

3.8. Paraméterek kezelése

Mivel minden nemterminálisból metódus lesz, ezért lehetőség van arra is, hogy bemenő és kimenő paramétereket definiáljunk számukra a következő módon:

```
nemterminális [tipusb1 bemeno1, tipusb2 bemeno2,...]
              returns [tipusk1 kimeno1, tipusk2 kimeno2]: definició
                                                        ;
```

A bemenoX és a kimenoX a bemenő illetve a kimenő paraméterek neve. Hivatkozni rájuk szemantikus akcióból a címkéhez hasonlóan $ prefixszel lehet, így állíthatjuk be a szabály kimenő paraméterét, illetve használhatjuk fel a bemenő paraméter értékét.

Ha valamelyik szabály jobb oldalára bemenő paraméterrel rendelkező nemterminálist írunk, akkor ott (éppen mint függvényhívásnál) meg kell mondani milyen paraméterekkel hívjuk. Ezeket a paramétereket [érték1,érték2,...] szintaktikával kell megadnunk, például:

```
x [int be1, int be2] returns [int ki] :
    SZAM {$ki = $SZAM.int * $be1 + $be2;}
    ;

y :
    'f' x[2,3] {System.out.println("Visszadott␣érték="+$x.ki);}
    ;
```

3.9. Kifejezés értékének kiszámítása szemantikus akciókkal

A következőkben lássunk egy komplett példát, konkrétan a 2.1 ábrán lévő nyelvtan megvalósítását ANTLR nyelvtan formájában olyan módon, hogy annak értékét szemantikus függvények formájában ki is számolja.

A megvalósításban minden nemterminálishoz fölvettünk egy kimenő paramétert (v), amelybe adjuk vissza a kiszámolt részkifejezés értékét. A változók értékét pedig a Parser osztályunkban egy *ids* nevű Map típusú adattagban tároljuk.

SimpleExpr1.g

```
1  grammar SimpleExpr1;
2
3  @members {
4      private java.util.Map<String,Integer> ids = new
           java.util.TreeMap<String,Integer>();
5
6      public static void main(String[] args) throws Exception {
7          SimpleExpr1Lexer lex = new SimpleExpr1Lexer(new
               ANTLRFileStream(args[0]));
```

```
8           CommonTokenStream tokens = new CommonTokenStream(lex);
9           SimpleExpr1Parser parser = new SimpleExpr1Parser(tokens);
10          parser.program();
11      }
12  }
13
14  /*------------------------------------------------------------------
15   * Lexikalis elemzo
16   *------------------------------------------------------------------*/
17
18  AZONOSITO    : ('a'..'z')('a'..'z'|'0'..'9')*;
19  SZAM         : ('0'..'9')+;
20  SZOKOZ       : (' '|'\t'|'\r'|'\n') {$channel=HIDDEN;};
21  KOMMENT      : '#' .* '\n'          {$channel=HIDDEN;};
22
23  /*------------------------------------------------------------------
24   * Szintaktikus elemzo
25   *------------------------------------------------------------------*/
26
27  program : ( utasitas )+ EOF
28  ;
29
30  utasitas returns [int v] :
31      AZONOSITO ':=' kifejezes
32      { ids.put($AZONOSITO.text,$kifejezes.v);
33        System.out.println($AZONOSITO.text + " = " + $kifejezes.v);
34      }
35  ;
36
37  kifejezes returns [int v] :
38      t1=tag { $v = $t1.v; }
39      (  '+' t2=tag { $v += $t2.v; }
40       | '-' t3=tag { $v -= $t3.v; }
41      )*
42  ;
43
44  tag returns [int v] :
45      t1=tenyezo { $v = $t1.v; }
46      (  '*' t2=tenyezo { $v *= $t2.v; }
47       | '/' t3=tenyezo { $v /= $t3.v; }
48      )*
49  ;
50
51  tenyezo returns [int v] :
52        SZAM { $v = $SZAM.int; }
53      | '(' kifejezes ')' { $v = $kifejezes.v; }
54      | AZONOSITO { $v = ids.get($AZONOSITO.text); }
55  ;
```

3.9.1. Feladatok

- Valósítsuk meg a 3.6.1-ban leírt feladatokat!

3.9.2. ANTLR által generált kód

Ahhoz, hogy jobban megértsük az ANTLR működését, érdemes megnézni milyen jellegű kódot generál. Minden nemterminálisból a Parser osztályunk egy metódusa lesz, amit a fenti példa tenyezo nemterminálisa esetén a következő szerkezettel lehet leírni:

```
int tenyezo()
{
    int v;
    if (input.LA(1)=='(') {
        readToken('(');
        v=kifejezes();
        readToken(')');
    }
    elseif (input.LA(1)==SZAM) {
        Token s=readToken(SZAM);
        v=s.getInt();
    }
    elseif (input.LA(1)==AZONOSITO) {
        Token a=readToken(AZONOSITO);
        String varibleName=a.getText();
        v=getVarible(varibleName);
    }
    else {
        syntaxError();
    }
    return v;
}
```

A kódban jól látható, hogyan dönt az elemző az egyes alternatívák között: előre néz egy tokent (ez az input.LA(1)), és az alapján dönt, hogy az melyik jobb oldalnak felelhet meg, amit most „szemmel” is könnyen meg tudunk tenni – általános kiszámításához pedig segítség az előző fejezetben ismételt FI halmaz fogalma. Ha valamelyik alternatívának megfelel az adott input – LL(k) nyelvtan esetében ez csak az egyik alternatíva lehet – akkor annak az elemzésébe kezd, beolvasva az abban lévő tokeneket, illetve meghívja az ott lévő nemterminálisoknak megfelelő metódusokat, amelyek elemzik az adott nemterminálist. Ha pedig egyik alternatívának sem felel meg, akkor szintaktikai hibát kell jeleznünk.

Ugyanezt az ANTLR a következő formában generálja le:

SimpleExpr1 - tenyezo

```
// $ANTLR start "tenyezo"
// SimpleExpr1.g:51:1: tenyezo returns [int v] : ( SZAM | '(' kifejezes
    ')' | AZONOSITO );
public final int tenyezo() throws RecognitionException {
    int v = 0;

    Token SZAM3=null;
    Token AZONOSITO5=null;
    int kifejezes4 = 0;

    try {
        // SimpleExpr1.g:51:25: ( SZAM | '(' kifejezes ')' | AZONOSITO )
```

```
        int alt4=3;
        switch ( input.LA(1) ) {
        case SZAM:
            {
            alt4=1;
            }
            break;
        case 13:
            {
            alt4=2;
            }
            break;
        case AZONOSITO:
            {
            alt4=3;
            }
            break;
        default:
            NoViableAltException nvae =
            new NoViableAltException("", 4, 0, input);
            throw nvae;
        }
        switch (alt4) {
        case 1 :
            // SimpleExpr1.g:52:7: SZAM
            {
            SZAM3=(Token)match(input, SZAM, FOLLOW_SZAM_in_tenyezo291);
            v = (SZAM3!=null?Integer.valueOf(SZAM3.getText()):0);
            }
            break;
        case 2 :
            // SimpleExpr1.g:53:7: '(' kifejezes ')'
            {
            match(input,13,FOLLOW_13_in_tenyezo301);
            pushFollow(FOLLOW_kifejezes_in_tenyezo303);
            kifejezes4=kifejezes();
            state._fsp--;
            match(input,14,FOLLOW_14_in_tenyezo305);
            v = kifejezes4;
            }
            break;
        case 3 :
            // SimpleExpr1.g:54:7: AZONOSITO
            {
            AZONOSITO5=(Token)match(input, AZONOSITO,
                FOLLOW_AZONOSITO_in_tenyezo315);
            v = ids.get((AZONOSITO5!=null?AZONOSITO5.getText():null));
            }
            break;
        }
    }
    catch (RecognitionException re) {
        reportError(re);
```

```
        recover(input,re);
    }
    finally {
    }
    return v;
}
// $ANTLR end "tenyezo"
```

A megjegyzésekben látható, hogy melyik nyelvtan fájl sor melyik kódrészletet generálta, és hogy hova másolódtak be az általunk szemantikus akcióként megadott kódsorok. Látható az alternatívák közötti döntés módja is, továbbá a kimenő paraméter (v) visszatérési értékként való visszaadása.

3.10. AST építés

Ha a fordítóprogramunk feladata komplexebb annál, hogy a szintaktikai elemzés közben megoldható legyen, akkor a szintaktikai elemzővel csak egy köztes formátumot érdemes építenünk, egy AST-t, amit aztán a szemantikus elemzőnk fog feldolgozni.

Ennek megvalósításához első körben az AST-nket kell megterveznünk. A fenti példánk esetében szükségünk lesz egy Program csomópontra a fa gyökere számára, egy Assignment csomópontra az értékadások reprezentálására, egy Operator csomópontra az aritmetikai műveletek ábrázolásához, és a fa leveleiben lévő információkat pedig az Identifier (azonosító) és a Constant (szám) fogja tárolni.
A Program csomópontnak van valamennyi utasitas azaz Assignment gyermeke, amit a list nevű adattagjában tárol:

Program.java

```
package ast;

import java.util.List;
import java.util.ArrayList;

public class Program {

    private List<Assignment> list;

    public Program(){
        list = new ArrayList<Assignment>();
    }

    public void addAssignment(Assignment asgmt){
        list.add(asgmt);
    }

    public List<Assignment> getAssignments() {
        return list;
    }

    public void evaluate(java.util.Map<String,Integer> ids) {
```

```
        for (Assignment asgmt : list)
            asgmt.evaluate(ids);
    }

    public String toString() {
        StringBuffer buf = new StringBuffer();
        buf.append("(Program \n");
        for (Assignment asgmt : list)
            buf.append(String.format("  %s\n", asgmt));
        buf.append(")\n");
        return buf.toString();
    }

}
```

Egy Assignment, azaz értékadó utasításnak (utasitas) két gyermeke van: az AZONOSITO, amire az értékadás bal oldala (id adattag), és egy kifejezés (kifejezes nemterminális), amely az értékadás jobb oldal (expr adattag tárolja).

Assignment.java

```
package ast;

public class Assignment {

    private Identifier id;
    private Expression expr;

    public Assignment(Identifier id, Expression expr) {
        this.id = id;
        this.expr = expr;
    }

    public Identifier getIdentifier() {
        return id;
    }

    public Expression getExpression() {
        return expr;
    }

    public void evaluate(java.util.Map<String,Integer> ids) {
        int v = expr.evaluate(ids);
        ids.put(id.getName(), v);
        System.out.println(id.getName() + " = " + v);
    }

    public String toString() {
        return String.format("(Assignment %s %s)", id, expr);
    }

}
```

A kifejezések egységes tárolása kedvéért létrehozunk egy közös őst, az Expression osztályt, amely leszármazottja a Constant osztály egy SZAM tokent tárol a value adattagjában, az Identifier osztály pedig egy AZONOSITO-t tárol a name adattagjában, és egy Operator osztály, amely egy bináris kifejezést tárol: bal oldalát (left adattag), jobb oldalát (right adattag) és a közöttük lévő műveletet (op adattag).

Expression.java

```
package ast;

public abstract class Expression {

    public abstract int evaluate(java.util.Map<String,Integer> ids);

}
```

Identifier.java

```
package ast;

public class Identifier extends Expression {

    private String name;

    public Identifier(String name) {
        this.name = name;
    }

    public String getName(){
        return name;
    }

    public int evaluate(java.util.Map<String,Integer> ids) {
        return ids.get(name);
    }

    public String toString() {
        return String.format("(ID␣%s)", name);
    }

}
```

Constant.java

```
package ast;

public class Constant extends Expression {

    private int value;

    public Constant(int value) {
        this.value = value;
    }
```

```
    public int getValue() {
        return value;
    }

    public int evaluate(java.util.Map<String,Integer> ids) {
        return value;
    }

    public String toString() {
        return String.format("(Constant␣%d)", value);
    }

}
```

Operator.java

```
package ast;

public class Operator extends Expression {

    private static final String[] symbols = { "+", "-", "*", "/" };

    public static final int ADD = 0;
    public static final int SUB = 1;
    public static final int MUL = 2;
    public static final int DIV = 3;

    private int op;
    private Expression left;
    private Expression right;

    public Operator(int op, Expression left, Expression right) {
        this.op = op;
        this.left = left;
        this.right = right;
    }

    public int getOperator() {
        return op;
    }

    public Expression getLeftOperand() {
        return left;
    }

    public Expression getRightOperand() {
        return right;
    }

    public int evaluate(java.util.Map<String,Integer> ids) {
        int lhs = left.evaluate(ids);
        int rhs = right.evaluate(ids);
```

```
        switch (op) {
            case ADD:
                return lhs + rhs;
            case SUB:
                return lhs - rhs;
            case MUL:
                return lhs * rhs;
            case DIV:
                return lhs / rhs;
            default:
                throw new RuntimeException("Unexpected operator " + op);
        }
    }

    public String toString() {
        return String.format("(%s %s %s)", symbols[op], left, right);
    }

}
```

Ezekkel az osztályokkal definiáltuk az AST-nk építőköveit, amelyből a nyelvtanunk a korábban megszokott módon szemantikus akciókon belül építkezni tud. A korábbi példához képest annyi a különbség, hogy itt most az egyes nemterminálisok kimenő paramétere nem a részkifejezés értéke lesz, hanem a részkifejezést reprezentáló AST részfa.

SimpleExpr3.g

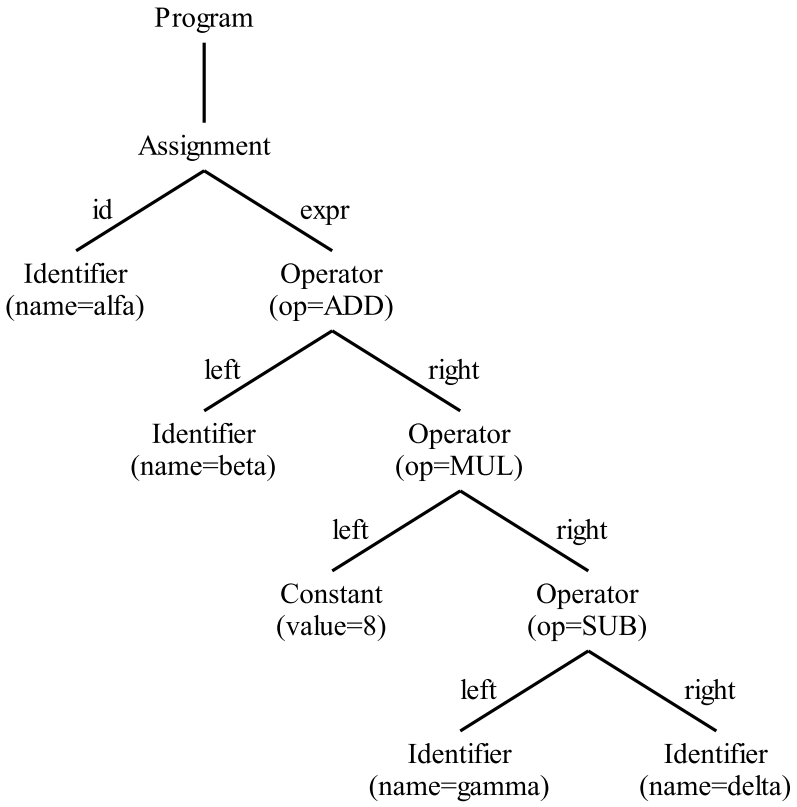
```
grammar SimpleExpr3;

@members {
    private java.util.Map<String,Integer> ids = new
        java.util.TreeMap<String,Integer>();

    public static void main(String[] args) throws Exception {
        SimpleExpr3Lexer lex = new SimpleExpr3Lexer(new
            ANTLRFileStream(args[0]));
        CommonTokenStream tokens = new CommonTokenStream(lex);
        SimpleExpr3Parser parser = new SimpleExpr3Parser(tokens);
        parser.program();
    }
}

/*------------------------------------------------------------------
 * Lexikalis elemzo
 *------------------------------------------------------------------*/

AZONOSITO   : ('a'..'z')('a'..'z'|'0'..'9')*;
SZAM        : ('0'..'9')+;
SZOKOZ      : (' '|'\t'|'\r'|'\n') {$channel=HIDDEN;};
KOMMENT     : '#' .* '\n'          {$channel=HIDDEN;};

/*------------------------------------------------------------------
 * Szintaktikus elemzo
 *------------------------------------------------------------------*/
```

```
program returns [ast.Program p]
    @init{ $p = new ast.Program(); }
    @after{ $p.evaluate(ids); }
    : ( utasitas { $p.addAssignment($utasitas.p); } )+ EOF
;

utasitas returns [ast.Assignment p]:
    AZONOSITO ':=' kifejezes  { $p = new ast.Assignment(new
        ast.Identifier($AZONOSITO.text), $kifejezes.p); }
;

kifejezes returns [ast.Expression p] :
    t1=tag { $p = $t1.p; }
    (  '+' t2=tag { $p = new ast.Operator(ast.Operator.ADD, $p, $t2.p); }
     | '-' t3=tag { $p = new ast.Operator(ast.Operator.SUB, $p, $t3.p); }
    )*
;

tag returns [ast.Expression p] :
    t1=tenyezo { $p = $t1.p; }
    (  '*' t2=tenyezo { $p = new ast.Operator(ast.Operator.MUL, $p,
        $t2.p); }
     | '/' t3=tenyezo { $p = new ast.Operator(ast.Operator.DIV, $p,
         $t3.p); }
    )*
;

tenyezo returns [ast.Expression p] :
      SZAM { $p = new ast.Constant($SZAM.int); }
    | '(' kifejezes ')' { $p = $kifejezes.p; }
    | AZONOSITO { $p = new ast.Identifier($AZONOSITO.text); }
;
```

A teljes AST felépítése után, melynek ábrája a 3.7 ábrán látható, a program nemterminális after részében a nyelvtan meghívja a gyökér evaluate metódusát, átadva neki az azonosítókat tárolni hivatott Map-et. Ez az evaluate függvény azután a fát bejárva kiszámítja minden Assignment node által reprezentált értékadást, kiírva annak értékét a képernyőre.

3.10.1. Feladatok

- Valósítsuk meg ebben a környezetben is a 3.6.1-ban leírt feladatokat!
- Írjunk a fenti példához olyan AST-t bejáró és transzformáló algoritmust, amely képes az olyan aritmetikai kifejezéseket egyszerűsíteni illetve összevonni, mint amilyen a $0x+3y$ vagy a $2x+4y+5x-2y$!



3.7. ábra. Az alfa := beta + 8 * (gamma - delta) *inputra generálódó AST*

4. fejezet

Attribútumos fordítás

Amint azt a korábbi fejezetekben láttuk a lexikális illetve szintaktikus elemzés segítségével lehetőség van az egyszerűbb programozási hibák kiszűrésére. Ilyenek például speciális karakterek helytelen használata, kulcsszavak elhagyása vagy utasítások szerkezetének eltévesztése. Azonban vannak olyan fordítási problémák, amelyek nem kezelhetők környezet-független nyelvtanokkal. Ilyen például a *típus kompatibilitás* illetve a *változók láthatóságának* kezelése. Ha például egy értékadó utasítás baloldalán egész változó szerepel a jobboldalán szereplő kifejezés pedig valós értéket ad és az adott programozási nyelv nem engedélyezi az ilyen értékekre a konverziót, akkor típus kompatibilitási probléma jelentkezik. Az ilyen probléma detektálása környezet-független nyelvtannal általános esetben nem lehetséges, hiszen a problémát sok esetben a kifejezésben szereplő változók típusa okozza, ami csak a szimbólumtáblán keresztül elérhető. A változók láthatóságának kérdése ott jelentkezik, amikor egy adott változó használatakor a fordítóprogramnak meg kell keresni a változó adott környezetben érvényes deklarációját. Szimbólumtábla használata nélkül ez sem lehetséges, így a környezetfüggetlen-nyelvtanokon alapuló egyszerű szintaktikus elemzés alkalmatlan erre a feladatra. A típus kompatibilitás eldöntését és a változók láthatóságának kezelését szokás fordítási szemantikus problémának vagy a nyelvek környezetfüggő tulajdonságainak is nevezni. Látjuk, hogy ezen problémák kezelésére a környezet-független nyelvtanoknál 'erősebb' eszközre van szükség. Ezt felismerve Knuth 1968-ban definiálta az Attribútum Nyelvtan alapú számítási modellt [7], amely alkalmas programozási nyelvek környezetfüggő tulajdonságainak kezelésére. A továbbiakban először formálisan definiáljuk a Attribútum nyelvtanok fogalmát, ismertetünk többfajta attribútum kiértékelő stratégiát majd pedig bemutatjuk, hogyan lehet attribútum nyelvtanok használatával kezelni a típus kompatibilitási és láthatósági problémákat.

4.1. Attribútum nyelvtanok definíciója

Egy attribútum nyelvtant öt komponenssel adhatunk meg:

$$AG = (G, A, Val, SF, SC)$$

Az *első komponens* $G = (N,T,S,P)$ egy *redukált*[1] *környezet-független* nyelvtan, ahol N a nemterminálisokat, T a terminális szimbólumokat ($V = N \cup T$ a nyelvtan összes szimbólumát), S pedig a kezdő szimbólumot jelöli. P-vel jelöljük a környezet-független szabályokat, amelyek formája:

$$p \in P : X_0 \to X_1 \dots X_{n_p} \quad (X_0 \in N, X_i \in V, 1 \leq i \leq n_p)$$

A *második komponens* az attribútumok halmaza, amit A-val jelölünk. Ezeket az attribútumokat a környezet-független nyelvtan szimbólumaihoz fogjuk hozzárendelni az alábbiak szerint. Jelölje AN az attribútum nevek halmazát, amit két diszjunkt részhalmazra osztunk. ANI jelöli az örökölt attribútum nevek ANS pedig a szintetizált attribútum nevek halmazát. (Az örökölt és szintetizált attribútumok szemléletes jelentését később fogjuk ismertetni). Teljesül, hogy

$$AN = ANI \cup ANS \quad \text{és} \quad ANI \cap ANS = \emptyset$$

Jelölje *X.a az X szimbólum a attribútumát*, ahol $X \in V$ és $a \in AN$. Egy szimbólumhoz több attribútum nevet is rendelhetünk és egy attribútum név több szimbólumhoz is rendelhető. Egy attribútum tehát egy pár, amit egy nyelvtani szimbólum és egy attribútum név határoz meg. Az A attribútumok halmaza az összes nyelvtani szimbólum attribútumainak uniójaként áll elő, vagyis $A = \bigcup_{X \in V} A_X$, ahol A_X az X szimbólum attribútum halmaza. Az A is felbontható két diszjunkt részhalmazra, az AI örökölt attribútumok és az AS szintetizált attribútumok halmazára.

Teljesül, hogy

$$A = AI \cup AS \quad \text{és} \quad AI \cap AS = \emptyset$$

Ezek után vezessük be az *attribútum előfordulások* halmazát. Legyen $p \in P : X_0 \to X_1, \dots, X_{n_p}$ a környezetfüggetlen nyelvtan egy szabálya. A szabály szimbólumaihoz rendelt összes attribútum uniója adja meg a *p szabály attribútum előfordulásainak halmazát*, amit A_p-vel jelölünk.

Vagyis,

$$A_p = \bigcup_{i=0..n_p} \bigcup_{X_i.a \in A_{X_i}} X_i.a$$

A p szabály attribútum előfordulásai közül *definiáló attribútum előfordulásoknak* nevezzük és AF_p-vel jelöljük a baloldali nemterminális szintetizált és a jobboldali szimbólumok örökölt attribútumait.

Vagyis,

$$AF_p = \{X_i.a \mid (i = 0 \text{ és } X_i.a \in AS_{X_i}) \text{ vagy } (i > 0 \text{ és } X_i.a \in AI_{X_i})\}$$

[1] Egy környezetfüggetlen nyelvtan redukált, ha minden nemterminális levezethető a kezdő szimbólumból, és minden nemterminálisból levezethető egy csak terminálisokat tartalmazó sztring.

A p szabály többi attribútum előfordulását, vagyis a baloldali szimbólum örökölt attribútumait és a jobboldali szimbólumok szintetizált attribútumait *alkalmazott attribútum előfordulásoknak* nevezzük, és AC_p-vel jelöljük.

Vagyis,

$$AC_p = \{X_i.a \mid (i = 0 \text{ és } X_i.a \in AI_{X_i}) \text{ vagy } (i > 0 \text{ és } X_i.a \in AS_{X_i})\}$$

Teljesül, hogy

$$A_p = AF_p \cup AC_p \quad \text{és} \quad AF_p \cap AC_p = \emptyset$$

Az attribútum nyelvtanok *harmadik komponensét* a szabályokhoz rendelt *szemantikus függvények* alkotják, amit SF-el jelölünk. Legyen $p \in P : X_0 \to X_1, \ldots, X_{n_p}$ egy környezetfüggetlen szabály és jelöljön az

$$X_i.a = g(X_j.b, \ldots, X_k.c, \ldots)$$

egy olyan értékadást, ami az $X_i.a$ definiáló attribútum értékét definiálja a g függvény segítségével. A g függvény argumentumai a p szabály attribútum előfordulásai lehetnek.

Vagyis,

$$X_i.a \in AF_p \quad \text{és} \quad X_j.b, X_k.c \in A_p$$

Az $X_i.a = g(X_j.b, \ldots, X_k.c, \ldots)$ értékadást a p szabály szemantikus függvényének nevezzük.

Jelölje SF_p a p szabály szemantikus függvényeinek halmazát.

Az attribútum nyelvtanok szigorú megkötése, hogy a p szabály minden $X_i.a$ definiáló attribútum előfordulására pontosan egy olyan SF_p-beli szemantikus függvényt kell megadni, ami $X_i.a$ értékét definiálja. A p szabály alkalmazott attribútumait nem definiálhatjuk SF_p-beli szemantikus függvényekkel. Vagyis az AF_p és az SF_p halmazok ugyanannyi elemet tartalmaznak. Az attribútum nyelvtan SF halmazát az SF_p halmazok uniójaként kapjuk meg.

Vagyis,

$$SF = \bigcup_{p \in P} SF_p$$

Egy attribútum nyelvtant *normál formában megadott attribútum nyelvtannak* nevezünk, ha a szemantikus függvények argumentumai csak alkalmazott attribútum előfordulások lehetnek. Megjegyezzük, hogy ez nem jelent lényeges megszorítást, mivel minden attribútum nyelvtanhoz megadható egy vele ekvivalens normál formájú attribútum nyelvtan. A jegyzetben szereplő attribútum nyelvtanokat normál formában adjuk meg.

Az attribútum nyelvtanok *negyedik komponense* az attribútumok lehetséges értékeinek halmaza, amit *Val*-lal jelölünk. Az attribútumok a programozási nyelvek változóihoz hasonlíthatók, ezért érték készletük általában a programozási nyelvekben szokásos típusoknak megfelelő érték készletekkel adható meg.

Az *ötödik komponens* a *szemantikus feltételek halmaza*, amit *SC*-vel jelölünk. A szemantikus függvényekhez hasonlóan a szemantikus feltételeket is a környezet-független nyelvtan szabályaihoz rendeljük hozzá, azzal a különbséggel, hogy a szemantikus feltételekkel nem definiáljuk attribútum előfordulások értékét. Szerepük elsősorban az, hogy az elemzés során bizonyos szemantikus akciókat (pl. hibaüzenetek írását, kódgenerálást) adhassunk meg. A szemantikus feltételek általában egy függvény hívást jelentenek, ahol a függvény argumentumai az adott szabály attribútum előfordulásai lehetnek.

Az attribútum nyelvtan definíciójának szemléltetésére tekintsük a 4.1. ábrán szereplő példát. Az ábrán szereplő attribútum nyelvtannak nincs konkrét jelentése csupán szemléltetési célokat szolgál.

Attribútum nyelvtan: kezdő példa;

Szintetizált attribútum nevek: a: int;

Örökölt attribútum nevek: b: int;

Nemterminálisok és attributumok:
S;
X: a,b;
Y: a,b;

Terminálisok: x,y,u,v,w ;

Szabályok és szemantikus függvények:

P1: S → u Y y ;
do
Y.a := 0;
end

P2: Y → v X x ;
do
X.a := Y.a + 1;
Y.b := X.b;
end

P3: X → w ;
do
X.b := X.a;
end

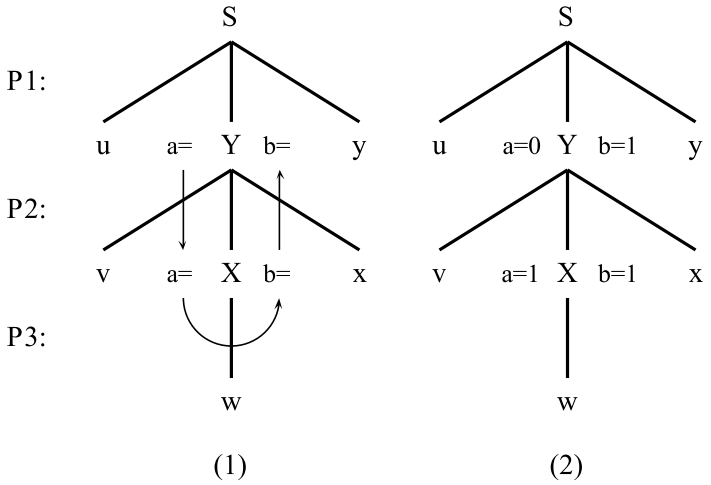
4.1. ábra. Példa attribútum nyelvtan

A 4.1. ábrán szereplő attribútum nyelvtannak két egész típusú attribútum neve van *a* és *b*, amik közül *a* örökölt és *b* szintetizált. A környezet-független nyelvtan nemterminálisait az $\{S,X,Y\}$ halmaz, terminálisait az $\{x,y,u,v,w\}$ halmaz adja meg. A nyelvtan *S* kezdő

szimbólumának nincsen attribútuma, az X és Y nemterminálisoknak van 'a' és 'b' attribútuma. A terminális szimbólumoknak nincs attribútuma. A nyelvtanban három szintaktikus szabály van. Az első szabályban csak az Y nemterminálisnak vannak attribútumai és ezek közül az $Y.a$ definiáló előfordulás. Ennek ad értéket az $Y.a := 0$ szemantikus függvény. A második szabályban az Y és X nemterminálisoknak vannak attribútumai, amelyek közül az $X.a$ és az $Y.b$ definiáló előfordulások. Ezeknek adnak értéket az $X.a := Y.a + 1$ és az $Y.b := X.b$ szemantikus függvények. A harmadik szabályban az X nemterminális $X.b$ definiáló attribútum előfordulását az $X.b := X.a$ szemantikus függvény definiálja.

4.2. Attribútum kiértékelés

Ezek után készítsünk egy elemzést a példa attribútum nyelvtannal az $'uvwxy'$ inputra. Megjegyezzük, hogy a nyelvtan erre az egy inputra ad érvényes elemzést, vagyis egy elemzési fája van. Az elemzési fát kiegészítve az attribútumok üres példányaival láthatjuk a 4.2. ábra baloldalán.



4.2. ábra. A kezdő példa kezdeti (1) és kiértékelt (2) attribútumos elemzési fája

Az *attribútum példányokkal* kiegészített elemzési fát *attribútumos elemzési fának* (röviden attribútumos fának nevezzük). Szemléletesen ez annyit jelent, hogy az elemzési fa csomópontjait kiegészítjük az adott csomópontot reprezentáló nyelvtani szimbólumhoz rendelt attribútumokkal. Az így keletkezett attribútum példányok kezdeti értéke nem definiált.

Az attribútumos számítási modell lényege, hogy az attribútumos elemzési fát bejárva megpróbáljuk az attribútum példányok értékeit meghatározni. Az attribútum értékek kiszámításához a szabályokhoz rendelt szemantikus függvényeket használjuk.

A 4.3. ábrán láthatjuk egy egyszerű attribútum kiértékelési stratégiát megvalósító program vázlatát.

```
Procedure X_0 (N:node)
  begin
  for i := 1 to n_p do
    begin
    kiertekel(AI_{X_i});
    callX_i
    end
  kiertekel(AS_{X_0});
  end
```

4.3. ábra. Attribútum kiértékelési stratégia

A 4.3. ábrán szereplő rekurzív program bemenetként megkapja egy attribútumos fa gyökér csomópontját. Feltesszük, hogy a gyökér pontban a $p \in P : X_0 \to X_1, \ldots, X_{n_p}$ szabály lett végrehajtva. Az algoritmus először kiértékeli a legbaloldalibb részfa örökölt attribútumait majd meghívja önmagát erre a részfára. A részfa bejárásából visszatérve a következő részfa gyökerének örökölt attribútumait értékeljük ki és újból meghívjuk az eljárást az új részfára. Miután az összes részfát kiértékeltük a program kiszámítja a gyökér csomópont szintetizált attribútumait.

Alkalmazzuk ezt a kiértékelési stratégiát a 4.2. ábra baloldalán szereplő attribútumos fára. Az S csomópontról indulva először az $Y.a$ örökölt attribútumot számítjuk ki. Ehhez a $P1$ szabály környezetében található $Y.a := 0$ szemantikus függvényt használjuk. Ezután meghívjuk az eljárást az Y részfára. Ezzel átlépünk a $P2$-es szabály környezetébe és az $X.a := Y.a + 1$ szemantikus függvény felhasználásával kiszámítjuk az $X.a$ örökölt attribútum értékét. A következő lépésben meghívjuk az eljárást az X részfára. Ennek a csomópontnak már nincs további részfája, ezért a $P3$ szabály környezetében kiszámítjuk az $X.b$ szintetizált attribútumot a $X.b := X.a$ függvény alkalmazásával. Elhagyjuk az X részfát és visszatérünk az Y csomóponthoz a $P2$-es szabály környezetébe. Kiszámítjuk az $Y.b$ szintetizált attribútum példányt az $Y.b := X.b$ függvény felhasználásával. Visszatérünk az S csomóponthoz és befejezzük a kiértékelést. Minden attribútum példány értékét sikerült kiszámítani. A kiértékelt attribútumos fát láthatjuk a 4.2. ábra jobb oldalán.

Az attribútum kiértékelési stratégiák fontos tulajdonsága, hogy minden attribútum példánynak pontosan egyszer kell értéket kapni. Ez lényeges különbség a programozási nyelvek változói és az attribútum példányok között. Egy változó értékét felül írhatjuk, attribútum példányét nem.

Vegyük észre, hogy az attribútum példányok kiértékelési sorrendje nagyban befolyásolja az attribútum kiértékelés sikerét. Például az $X.a$ kiszámításához már ismerni kell az $Y.a$ attribútum példány értékét. Vagyis az attribútumos fa attribútum példányai között függőségek

keletkeznek, amelyek a szemantikus függvények alapján határozhatók meg. Egy $X_i.a = g(X_j.b,..,X_k.c,..)$ alakú szemantikus függvény az $(X_j.b,X_i.a)$ és $(X_k.c,X_i.a)$ *függőségeket* indukálja. Ez a jelölés azt mutatja, hogy az $X_i.a$ attribútum előfordulás értéke függ az $X_j.b$ és $X_k.c$ előfordulások értékeitől. Szemléletesen ez annyit jelent, hogy valahányszor ezt a szemantikus függvényt alkalmazzuk egy attribútumos fa kiértékelése során, mindannyiszor az attribútum előfordulásoknak megfelelő attribútum példányok között is létrejönnek ezek a függőségek. A 4.2. ábra baloldalán szereplő attribútumos fában az attribútum példányok közötti függőségeket reprezentálják a köztük lévő irányított élek. Az attribútum példányokat a hozzájuk kapcsolódó függőségi élekkel *attribútumos függőségi gráfnak* (röviden függőségi gráfnak) nevezzük.

Nyilvánvalóan az attribútum kiértékelés szempontjából komoly problémát jelent, ha ez a függőségi gráf kört tartalmaz, hiszen, akkor a körben szereplő attribútum példányok értékeit nem tudjuk meghatározni. Egy attribútum nyelvtant *cirkulárisnak* nevezünk, ha létezik olyan elemzési fája, amihez tartozó függőségi gráf kört tartalmaz. Egy cirkuláris attribútum nyelvtan nem igazán alkalmas fordítási probléma megoldására, hiszen nincs garancia arra, hogy az attribútum kiértékelést minden elemzési fára meg lehet valósítani.

Egy tetszőleges attribútum nyelvtanról a cirkularitás eldöntése NP nehéz probléma, vagyis általános esetre nem létezik hatékony algoritmus. Ennek elsősorban azaz oka, hogy egy általános attribútum nyelvtan alapján végtelen sok különböző elemzési fa generálható, így végtelen sok függőségi gráf körmentességét kell ellenőrizni. Ez a probléma elviekben megkérdőjelezheti az attribútum nyelvtanok gyakorlati alkalmazását, azonban van megoldás a probléma kezelésére. Nevezetesen a nem-cirkuláris attribútum nyelvtanok olyan részosztályait definiáljuk, amelyekbe való tartozás tetszőleges attribútum nyelvtanról hatékony algoritmussal eldönthető. Ha egy attribútum nyelvtan eleme egy ilyen részosztálynak, akkor a függőségi gráfjai biztosan körmentesek, így az attribútum példányai minden esetben kiértékelhetők. Ha egy attribútum nyelvtan nem eleme egy ilyen nem-cirkuláris részosztálynak, az nem jelenti azt, hogy az illető attribútum nyelvtan biztosan cirkuláris. Nyilvánvalóan az cél, hogy a nem-cirkuláris attribútum nyelvtanok minél nagyobb részosztályára tudjunk hatékony ellenőrző és attribútum kiértékelő módszert konstruálni. Ezzel tudjuk biztosítani azt, hogy a részosztály attribútum nyelvtanaival bonyolult fordítási problémákat is lehet kezelni.

A későbbiekben megismerkedünk néhány nem-cirkuláris attribútum nyelvtan részosztállyal, de most először egy összetettebb példán mutatjuk be az attribútumos számítási modell működését.

4.3. Attribútum nyelvtan bináris törtszám értékének kiszámítására

A példa attribútum nyelvtan bemenete egy tetszőleges bináris tört szám és attribútumos számítással határozzuk meg a tört értékét. Ez a példa szerepelt Knuth 1968-as cikkében

is [7].

A 4.4. ábrán szereplő attribútum nyelvtan egy tetszőleges bináris tört szám decimális értékét számítja ki az $S.v$ attribútumban. Az attribútum nyelvtan normál formában adott, vagyis a szemantikus függvények baloldalán definiáló attribútum előfordulások a jobboldalakon pedig alkalmazott előfordulások szerepelnek. Ha egy szintaktikus szabályban több azonos nevű szimbólum fordul elő (1-es , 2-es szabályok), akkor a szimbólumokat alsó indexszel különböztetjük meg. Az azonos nevű, de különböző indexszel rendelkező szimbólumok azonos nevű attribútumai különböző attribútum előfordulásnak számítanak. Tehát $N_1.v$ és $N_2.v$ két különböző attribútum előfordulás az 1-es szabályban.

A 4.5. ábrán láthatjuk az adott attribútum nyelvtan alapján készült attribútumos fát az `10.1` inputra.

Láthatjuk, hogy egy nemterminális szimbólum többször is előfordulhat a fában, és minden előfordulásra a hozzárendelt attribútumokból létrejön egy-egy példány. A példányok érteke kezdetben nem definiált. Az értékek meghatározására használjuk a 4.3. ábrán ismertetett attribútumos kiértékelési stratégiát. A számítási módszert alkalmazva a 4.6. ábrán szereplő attribútum példány értékeket tudjuk meghatározni.

A 4.6. ábrán megadtuk a kiszámított értékek számítási sorrendjét is (lásd az attribútum példányok melletti körben lévő számok). A kiértékelési algoritmus egyszeri végrehajtását szokás *kiértékelési menetnek* is nevezni. Láthatjuk, hogy végrehajtva egy kiértékelési menetet nem tudtuk kiszámítani minden attribútum példány értékét. A baloldali részfa minden attribútumát sikerült meghatározni, azonban a jobboldali részfa N csomópontjánál az r értékének meghatározása nem sikerült. Ennek azaz oka, hogy az $N.r$ attribútum példány kiszámításához az 1-es számú szintaktikus szabály $N_2.r := -N_2.l$ szemantikus függvényét kell használni. Azonban, mivel $N_2.r$ az N szimbólum örökölt az $N_2.l$ pedig ugyan ennek a szimbólumnak szintetizált attribútum, ezért az adott kiértékelési stratégia szerint $N_2.r$-et az N_2 csomópontú részfa bejárása előtt, míg $N_2.l$-et a részfa bejárása után értékeljük ki. Vagyis $N_2.r$ számításakor még nem ismerjük $N_2.l$ értékét. Láthatjuk viszont, hogy a jobboldali részfában az $N.l$ attribútum példányokat ki tudtuk számítani, tehát egy további kiértékelési menettel van lehetőség a maradék attribútum példányok értékeinek meghatározására is. A második kiértékelési menetben tehát az első menetben részlegesen kiértékelt attribútumos fából indulunk el. A korábban kiszámított attribútum példány értékeket változatlanul hagyjuk az érték nélküli példányokat pedig meg próbáljuk kiszámolni. A 4.7. ábrán láthatjuk a teljes kiértékelt attribútumos fát, ahol jelöltük a második menetben számított attribútumok kiértékelési sorrendjét is.

Tehát a korábbiakban ismertetett attribútum kiértékelési stratégiával két menetben sikeresen ki tudtuk számítani a bináris törtszám decimális értékét, meg tudtuk határozni az attribútumos fa minden példányát. Azonban vegyük észre, hogy ennek a kiértékelési stratégiának van egy komoly problémája. Nevezetesen az, hogy a fabejárás során minden egyes attribútum példány értékelésekor meg kell vizsgálni azt, hogy az adott példány nem függ-e olyan attribútum értéktől, ami még ismeretlen. Ez a probléma kiértékelési

Attribútum nyelvtan: bináris;

Szintetizált attribútum nevek:
l: int; {egy nemterminálisból levezethető sztring hosszát adja meg}
v: real; {egy nemterminálisból levezethető sztring decimális értékét adja meg}

Örökölt attribútum nevek:
r: int; {egy nemterminálisból levezethető sztring legjobboldalibb helyiértékét adja meg}

Nemterminálisok és attributumok:
S: v;
N: l,v,r;
B: v,r;

Terminálisok: '.','0','1'

Szabályok és szemantikus függvények:

1: $S \rightarrow N_1$ '.' N_2 ;
do
$S.v := N_1.v + N_2.v$;
$N_1.r := 0$;
$N_2.r := -N_2.l$;
end

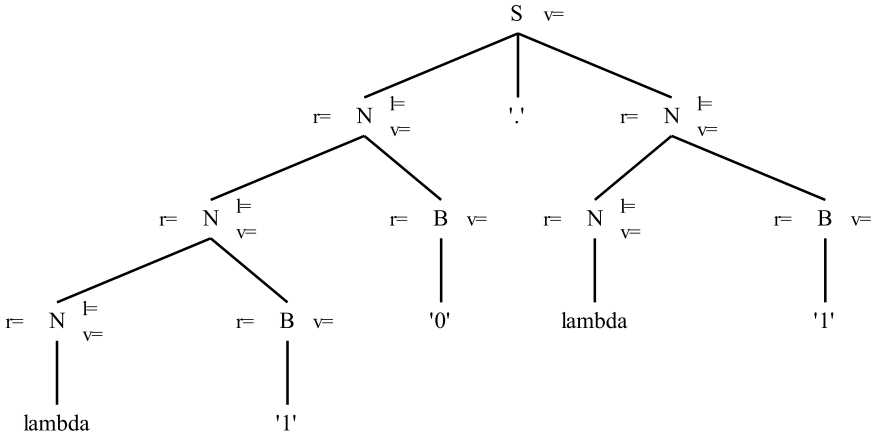
2: $N_0 \rightarrow N_1$ B;
do
$N_0.v := N_1.v + B.v$;
$N_0.l := N_1.l + 1$;
$N_1.r := N_0.r + 1$;
$B.r := N_0.r$;
end

3: $N \rightarrow \lambda$;
do
N.v:= 0;
N.l:= 0;
end

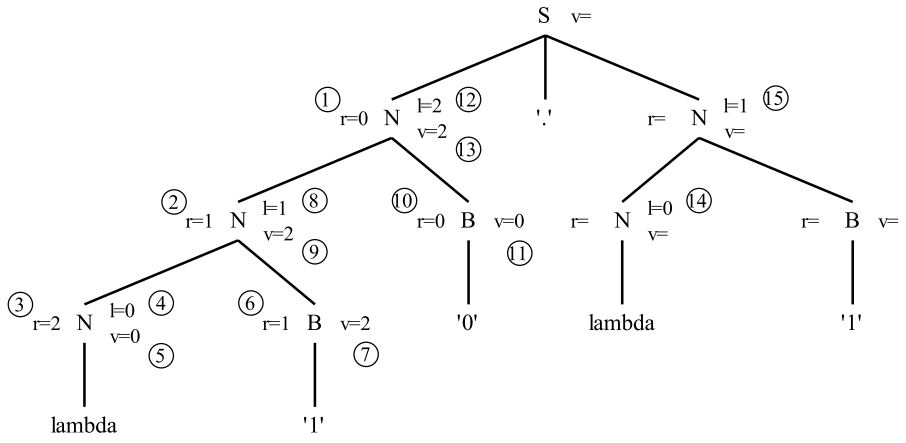
4: B $\rightarrow$ '1';
do
$B.v := 2^{B.r}$;
end

5: B $\rightarrow$ '0';
do
B.v := 0;
end

4.4. ábra. Attribútum nyelvtan bináris törtszám kiértékelésére



4.5. ábra. Attribútumos fa az `10.1` *inputra*



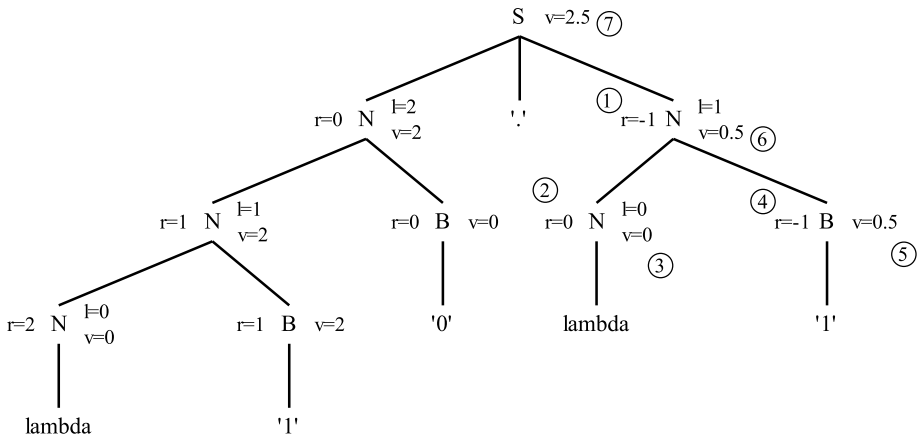
4.6. ábra. Az első menetben meghatározott attribútum példány értékek megadva a számítás sorrendjét

stratégia gyakorlati alkalmazhatóságát megkérdőjelezi, hiszen nagyobb méretű programok attribútumos fái sok millió attribútum példány számítását igénylik gyakran több menetben.

4.4. Többmenetes attribútum kiértékelő

A probléma kezelésére megoldást jelent, ha a korábbiakban ismertetett menet-vezérelt kiértékelési stratégiát ki tudjuk egészíteni olyan információval, hogy az egyes menetekben mely attribútum értékek biztosan kiértékelhetők. Ekkor a kiértékelő mindig csak az adott menetben kiértékelhető attribútumok értékeit határozza meg, így nincs szükség a költséges ellenőrzések végrehajtására.

Azt is vegyük észre, hogy az ismertetett kiértékelési stratégiában bizonyos korlátot jelent



4.7. ábra. A kiértékelt attribútumos fa, megadva a második menet számítási sorrendjét

az, hogy az attribútumos fa részfáit mindig balról jobbra haladva végezzük. Ezt a korlátozást könnyen megszüntethetjük azzal, hogy megengedjük a fordított irányú, vagyis a részfák jobbról balra történő bejárását. Ez alapján már megadhatunk egy általános több-menetes attribútum kiértékelő algoritmust, amit a 4.8. ábrán ismertetünk.

Az algoritmus bemenetként megkap egy attribútumos fát, továbbá egy menet-vezérlő sorozatot, ami előírja, hogy az egyes menetekben milyen sorrendben járjuk be a részfákat. Adott továbbá az attribútumok osztályozása, ami annyit jelent, hogy minden kiértékelési menetben csak az adott menethez tartozó osztály attribútumait fogjuk kiszámítani.

A $bejar(N : node; i : Integer)$ eljárás első részében az aktuális menet-vezérlő szimbólum alapján beállítjuk a részfa bejárási irány paramétereit. (Az N csomópontban a $p \in P : X_0 \rightarrow X_1, \ldots, X_{n_p}$ alakú szabályt alkalmaztuk). A bejárási irány szerint kiértékeljük az aktuális részfa olyan örökölt attribútum példányait, amelyekhez tartozó attribútum benne van az adott menethez tartozó attribútum osztályban, majd meghívjuk a $bejar()$ eljárást a részfára. Az összes részfa bejárása után kiértékeljük az X_0 szimbólumnak megfelelő csomópont olyan szintetizált attribútum példányait, amelyekhez tartozó attribútum benne van a menethez tartozó attribútum osztályban. Felhívjuk a figyelmet arra, hogy a menetekhez tartozó osztályokban attribútumok szerepelnek, míg a kiértékelés az attribútumos fákhoz tartozó attribútum példányokon történik. Ez annyit jelent, hogy egy $X.a$ attribútumnak megfelelő összes attribútum példányt ugyanabban a menetben kell kiértékelni.

Ha a $bejar()$ eljárást n-szer meghívjuk, akkor n menetben ki tudjuk értékelni az attribútumos fákat.

Természetesen egy tetszőleges attribútum nyelvtan attribútumos fái csak akkor értékelhetők ki az általános több-menetes kiértékelő algoritmussal, ha az attribútum halmazra meg tudunk adni egy menetek szerinti kiértékelő osztályozást. A továbbiakban ismertetünk egy olyan algoritmust, amely segítségével egy tetszőleges attribútum nyelvtanról eldönthető, hogy attribútumos fái kiértékelhetők-e az általános több-menetes kiértékelő algoritmussal.

Többmenetes kiértékelő

Input:
- Attribútumos elemzési fa
- $< d_1,..,d_n >$ menet vezérlő sorozat. A sorozat elemei R vagy L szimbólumok lehetnek. Ha $d_i = L$, akkor az i-edik menetben balról-jobbra, különben jobbról-balra történik a fabejárás.
- $< A_1,..,A_n >$ az A attribútum halmaz osztályozása. Az A_i halmaz jelöli az i-edik menetben kiértékelhető attribútumok halmazát.

Output: kiszámított attribútumos fa

Procedure $bejar(N : node; i : Integer)$;
var $u, v, z : Integer$;
begin
  if $d_i = L$ **then begin** $u = 1; v = 1; z = n_p$ **end**
  else begin $u = n_p; v = -1; z = 1$ **end**
  for $k = u$ **to** z **by** v **do**
  begin
    forall $X_k.a \in AI_{X_k}$ **do if** $X_k.a \in A_i$ **then** $kiertekel(N_k.a)$;
    $bejar(N_k, i)$;
  end
  forall $X_0.b \in AS_{X_0}$ **do if** $X_0.b \in A_i$ **then** $kiertekel(N_0.b)$;
end

for $i = 1$ **to** n **do** $bejar(gyoker, i)$;

4.8. ábra. Általános több-menetes attribútum kiértékelő

Pontosabban, olyan algoritmust definiálunk, amely azt feltételezi, hogy a több-menetes kiértékelés során a balról-jobbra és a jobbról-balra meneteket váltakozva hajtjuk végre. (erről kapta az algoritmus a nevét – ASE: Alternating Semantic Evaluator.) Megjegyezzük, hogy ez a kiértékelési mód nem jelent lényegi eltérést az általános módszerhez képest. Egy általános több-menetes kiértékelővel kiszámítható attribútum nyelvtan ASE módszerrel is kiértékelhető (legfeljebb üres menetek lesznek benne).

Az ASE algoritmus leírását a 4.9. ábra tartalmazza.

Az ASE algoritmus inputját egy tetszőleges attribútum nyelvtan alkotja. Az algoritmus erről az attribútum nyelvtanról eldönti, hogy attribútumai kiértékelhetők-e véges számú váltakozó irányú menetben. Amennyiben kiértékelhetők az algoritmus azt is megadja,hogy az egyes menetekben mely attribútumokat tudjuk kiszámítani. Az algoritmus úgy dolgozik, hogy az első lépésben feltételezzük, hogy az összes attribútum kiértékelhető az első menetben (ez balról-jobbra menet). Vagyis kezdetben az A_1 halmaz tartalmazza az összes attribútumot.

ASE algoritmus

Input: egy tetszőleges $AG = (G, A, Val, SF, SC)$ attribútum nyelvtan.
Output: annak eldöntése, hogy az AG eleme-e az ASE nem-cirkuláris attribútum nyelvtan osztálynak. Továbbá, ha $AG \in ASE$, akkor az algoritmus megadja, hogy az egyes attribútumok melyik menetben értékelhetők ki.

Procedure
begin
$m := 0$
$B := A$
repeat
$m := m + 1$
$A_m := B$
repeat
forall $X.a \in A_m(X_i = X)$ **do**
if $(\exists(X_j.b, X_i.a)$, ahol $X_j.b \notin A_k, k \leq m$ **or**
$X\,j.b \in A_m$ de $X_j.b$ az $X_i.a$ után fordul elő az adott kiértékelési menet szerint)
then $X.a \notin A_m$
until nem törölhető több attribútum
$B := B - A_m$
until $B = \emptyset$ **or** $(A_m = A_{m-1} = \emptyset)$
End

Az algoritmus befejeztével, ha:

1. $B = \emptyset$ akkor $AG \in ASE$
2. $A_m = A_{m-1} = \emptyset$ akkor $AG \notin ASE$

4.9. ábra. ASE (Alternating Semantic Evaluator) algoritmus

Ezután minden A_1 halmazbeli $X.a$ attribútumra megvizsgáljuk, hogy létezik-e olyan $X_i.a$ attribútum előfordulása (vagyis X és X_i ugyanaz a szimbólum), ami olyan attribútum előfordulástól függ, amit később tudunk kiszámítani mint az $X_i.a$ attribútumot. Ehhez az attribútum nyelvtan minden olyan szemantikus függvényét meg kell vizsgálnunk, aminek $X_i.a$ a baloldalán szerepel. Ha találunk ilyen attribútum előfordulást, akkor $X.a$-t töröljük az A_1 halmazból. Ha egy attribútumot törlünk az A_1 halmazból, akkor újra kell vizsgálnunk a már korábban megvizsgált attribútumokat is, mivel ha valamelyik függ a törölt attribútumtól, akkor azt is törölni kell. Mivel véges sok attribútumunk és szemantikus függvényünk van, ezért az A_1 halmaz meghatározható. Ezután azokat az attribútumokat, amelyeket töröltünk az

A_1 halmazból beletesszük az A_2 halmazba és megismételjük az attribútum kiértékelhetőségi vizsgálatot. Annyi eltérés van az előző menethez képest, hogy a második menetben jobbról-balra történő fabejárás szerint történik a kiértékelési ellenőrzés.

Az ASE algoritmus akkor ér véget, ha minden attribútumot be tudunk sorolni valamelyik menetbe, vagy pedig két egymást követő menetben nem tudunk új attribútumot kiszámítani. Az első esetben az adott attribútum nyelvtan m váltakozó menetben kiértékelhető vagyis AG $\in$ ASE. Ebben az esetben az A_i halmazok megadják azt is, hogy az egyes menetekben melyik attribútumok értékelhetők ki.

Ha két egymást követő menetben nem tudunk új attribútumot kiszámítani, akkor azt mondjuk, hogy az adott attribútum nyelvtan nem értékelhető ki az ASE algoritmus szerint, vagyis AG $\notin$ ASE . Ez nem jelenti azt, hogy az attribútum nyelvtan feltétlenül cirkuláris.

Az ASE algoritmus illusztrálására a 4.4. ábrán ismertetett bináris törtszám értékét meghatározó attribútum nyelvtanra eldöntjük, hogy eleme-e az ASE attribútum nyelvtan osztálynak.

Jelölje A a példa attribútumainak halmazát:

$$A = \{S.v, N.v, N.l, N.r, B.v, B.r\}$$

Az algoritmus szerint első lépésben az összes attribútumot beletesszük az A_1 halmazba:

$$A_1 = \{S.v, N.v, N.l, N.r, B.v, B.r\}$$

Megvizsgáljuk a halmaz elemeit, hogy van-e olyan függőség, ami meggátolja az első menetben való kiértékelést. A vizsgálat sorrendje tetszőleges. Ha balról-jobbra haladunk az A_1 halmaz attribútumain, akkor azt tapasztaljuk, hogy az $S.v, N.v, N.l$ attribútumokra nincs ilyen függőség viszont az $N.r$ attribútum az első szabályban függ az $N.l$ attribútum értékétől,amit csak később tudunk kiértékelni. Ezért $N.r$-et töröljük az A_1 halmazból. Mivel az $S.v, N.v, N.l$ attribútumok közvetlenül nem függnek $N.r$-től, ezért ezek az attribútumok továbbra is A_1-ben maradnak. A $B.v$ attribútum is az első ellenőrzés után még az A_1 halmazban marad, azonban a $B.r$ attribútumot törölni kell, mivel 2-es szabályban a $B.r := N_0.r$; szemantikus függvény miatt értéke függ az $N.r$ attribútum értékétől és $N.r$-et már korábban töröltük A_1-ből. $B.r$ törlése miatt újra ellenőrizni kell az A_1-ben maradt attribútumokat. Azt találjuk, hogy $B.v$ attribútumot is törölni kell a $B.v := 2^{B.r}$; függvény miatt. A $B.v$ attribútum törlése miatt törölnünk kell $N.v$-t ($N_0.v := N_1.v + B.v$; miatt) és ezután az $S.v$ attribútumot is ($S.v := N_1.v + N_2.v$; miatt). Azt kapjuk, hogy az A_1 halmazban csak az $N.l$ attribútum marad.

$$A_1 = \{N.l\}$$

Ezután az A_1 halmazból törölt attribútumokat hozzáadjuk az A_2 halmazhoz.

$$A_2 = \{S.v, N.v, N.r, B.v, B.r\}$$

Ismételten végrehajtjuk az attribútum kiértékelhetőségi ellenőrzést, és azt találjuk, hogy az A_2 halmazból nem kell törölni, vagyis ezek az attribútumok kiértékelhetők a második

menetben. Tehát a bináris törtszám értékét meghatározó attribútum nyelvtan eleme az ASE osztálynak, két menetben minden attribútumos fája kiértékelhető. Az első menetben csak az $N.l$ attribútum összes példányát értékeljük ki, majd a második menetben a többi attribútum példányait.

4.5. Típus kompatibilitás ellenőrzés

A menet-vezérelt attribútum kiértékelők megismerése után nézzük meg, hogyan alkalmazhatók az attribútum nyelvtanok fordítóprogramokban egyszerű típus kompatibilitási probléma kezelésére. A 4.10. és 4.11. ábrán ismertetünk egy értékadó utasítás egyszerű típus ellenőrzését megvalósító attribútum nyelvtant.

Attribútum nyelvtan: típus_kompatibilitás;

Típus:
$mod1 = (int, real)$;
$mod2 = (int, unspec)$;
Szintetizált attribútum nevek:
$aktualis\_tipus : mod1$;

Örökölt attribútum nevek:
$elvart\_tipus : mod2$;

Nemterminálisok és attribútumok:
$Utasitas$:;
$Kifejezes : aktualis\_tipus, elvart\_tipus$;
$Tag : aktualis\_tipus, elvart\_tipus$;
$Tenyezo : aktualis_t ipus, elvart_t ipus$;

Tokenek: $azonosito = betu(betu|szam|'\_')* : aktualis\_tipus$;

Terminálisok: $':=', '+', '-', '*', '/', '(', ')'$

4.10. ábra. Típus kompatibilitás ellenőrzés (definíciók)

Ez az attribútum nyelvtan azt ellenőrzi, hogy az értékadó utasítás baloldalán szereplő változó típusa kompatibilis-e a jobboldalon szereplő kifejezés típusával. A könnyebb érthetőség miatt a kifejezésben csak műveleti jeleket és egyszerű változó neveket engedünk meg. A változók és a kifejezés típusa is egész illetve valós lehet. Ha az értékadó utasítás baloldali változójának típusa egész és a kifejezés típusa valós, akkor hibát kell jelezni. Jelezni kell azt is, hogy mi okozza a hibát. Teljesülni kell, hogy hiba esetén tudjuk folytatni az elemzést, tehát további típus kompatibilitás hibákat is azonosítani kell. Az osztás művelet eredménye mindig valós típusú, a többi művelet eredményének típusa az operandusok típusától függ.

Szabályok és szemantikus függvények:

1: $Utasitas \rightarrow azonosito := Kifejezes$;
do
$Kifejezes.elvart\_tipus := (\ azonosito.aktualis\_tipus = int\ )$**?** int **:** $unspec$;
end

2: $Kifejezes_0 \rightarrow Kifejezes_1 + Tag$;
do
$Kifejezes_1.elvart\_tipus := Kifejezes_0.elvart\_tipus$;
$Tag.elvart\_tipus := Kifejezes_0.elvart\_tipus$;
$Kifejezes_0.aktualis\_tipus :=$
$(\ Kifejezes_1.aktualis\_tipus = real\ )$**?** $real$ **:** $Tag.aktualis\_tipus$;
end

3: $Kifejezes_0 \rightarrow Kifejezes_1 - Tag$;
do
$Kifejezes_1.elvart\_tipus := Kifejezes_0.elvart\_tipus$;
$Tag.elvart\_tipus := Kifejezes_0.elvart\_tipus$;
$Kifejezes_0.aktualis\_tipus :=$
$(\ Kifejezes_1.aktualis\_tipus = real\ )$**?** $real$ **:** $Tag.aktualis\_tipus$;
end

4: $Kifejezes \rightarrow Tag$;
do
$Tag.elvart\_tipus := Kifejezes.elvart\_tipus$;
$Kifejezes.aktualis\_tipus := Tag.aktualis\_tipus$;
end

5: $Tag_0 \rightarrow Tag_1 * Tenyezo$;
do
$Tag_1.elvart\_tipus := (\ Tag_0.elvart\_tipus = int\ )$**?** int **:** $unspec$;
$Tenyezo_1.elvart\_tipus := (\ Tag_0.elvart\_tipus = int\ )$**?** $real$ **:** $unspec$;
$Tag_0.aktualis\_tipus :=$
$(\ Tag_1.aktualis\_tipus = real\ )$**?** $real$ **:** $Tenyezo.aktualis\_tipus$;
end

6: $Tag_0 \rightarrow Tag_1 / Tenyezo$;
do
$Tag_1.elvart\_tipus := (\ Tag_0.elvart\_tipus = int\ )$**?** int **:** $unspec$;
$Tenyezo_1.elvart\_tipus := (\ Tag_0.elvart\_tipus = int\ )$**?** int **:** $unspec$;
$Tag_0.aktualis\_tipus := (\ Tag_0.elvart\_tipus = int\ )$**?** $error(), int$ **:** $real$;
end

7: $Tag \rightarrow Tenyezo$;
do
$Tenyezo.elvart\_tipus := Tag.elvart\_tipus$;
$Tag.aktualis\_tipus := Tenyezo.aktualis\_tipus$;
end

8: $Tenyezo \rightarrow (Kifejezes)$;
do
$Kifejezes.elvart\_tipus := Tenyezo.elvart\_tipus$;
$Tenyezo.aktualis\_tipus := Kifejezes.aktualis\_tipus$;
end

9: $Tenyezo \rightarrow azonosito$;
do
$Tenyezo.aktualis\_tipus := (\ Tenyezo.elvart\_tipus = int$ **&&**
$azonosito.aktualis\_tipus = real\ )$**?** $error(), int$ **:** $azonosito.aktualis\_tipus$;
end

4.11. ábra. Típus kompatibilitás ellenőrzés (szabályok)

Az attribútum nyelvtan egy-egy szintetizált és örökölt attribútum nevet tartalmaz (*aktualis_tipus*, *elvart_tipus*). Egy X szimbólum $X.aktualis\_tipus$ attribútuma az X szimbólummal címkézett elemzési részfa tényleges típusát adja meg. Az X szimbólum $X.elvart\_tipus$ attribútuma pedig azt mutatja meg, hogy a környezeti feltételek alapján az X szimbólummal címkézett elemzési részfa típusára milyen elvárás van. Például, ha az utasítás baloldali változójának típusa egész, akkor a jobboldali kifejezés elvárt típusa is egész. Az *elvart_tipus* attribútum értéke lehet nem specifikált is (*unspec*). Például, ha a baloldali változó típusa valós, akkor a jobboldali kifejezés típusa lehet valós és egész is. A nyelvtan kezdő szimbólumának nincs attribútuma (utasítás), a többi nemterminálisnak van mindkét attribútuma. Az *azonosito* a nyelvtan token szimbóluma, vagyis nem fordul elő szintaktikus szabály baloldalán. Az *azonosito* szimbólumot egy reguláris kifejezéssel definiáltuk. Vegyük észre, hogy az *azonosito*-nak is van szintetizált attribútuma (*aktualis_tipus*). Mivel az azonosító nem szerepel szabály baloldalán, ezért az *aktualis_tipus* attribútumának értékét a nyelvtan keretein belül nem tudjuk kiszámítani. Ezt az értéket az attribútum nyelvtan kívülről kapja, ezért az ilyen attribútumokat *kitüntetett szintetizált attribútumoknak* nevezzük. Konkrétan feltételezhetjük, hogy minden változóra a szimbólumtáblából megkaphatjuk a változó típusát. Általánosan mondhatjuk, hogy egy attribútum nyelvtan kezdő szimbólumának lehetnek *kitüntetett örökölt*, a tokeneknek pedig *kitüntetett szintetizált* attribútumai. A kitüntetett attribútumok értékeit az attribútum nyelvtan kívülről kapja. A 4.10. és 4.11. ábrákon ismertetett attribútum nyelvtant is normál formában adtuk meg, vagyis a szemantikus függvények baloldalán a definiáló, jobboldalán pedig alkalmazott előfordulások szerepelnek. Példaként értelmezzünk néhány szemantikus függvényt.

Az 1. szabályban található az alábbi függvény:

$$Kifejezes.elvart\_tipus := \textbf{(}\ azonosito.aktualis\_tipus = int\ \textbf{)?}\ int\ \textbf{:}\ unspec;$$

Ez azt jelenti, hogy amennyiben az értékadó utasítás baloldali változójának típusa egész, akkor a jobboldali kifejezés elvárt típusa egész, egyébként nem specifikált.

A 6. szabályban ($Tag_0 \rightarrow Tag_1/Tenyezo;$) található az alábbi szemantikus függvény:

$$Tag_0.aktualis\_tipus := \textbf{(}\ Tag_0.elvart\_tipus = int\ \textbf{)?}\ error(), int\ \textbf{:}\ real;$$

A függvény alapján, amennyiben a részfa gyökerének elvárt típusa egész ($Tag_0.elvart\_tipus$), akkor hibajelzés történik, mivel a részfában egy osztás műveletet hajtunk végre és ennek eredménye valós típusú lesz. A hibajelzést az $error()$ eljárás meghívásával valósítjuk meg. Vegyük észre, hogy a hibajelzés során pontosan meg tudjuk mondani mi okozta a típus kompatibilitási problémát. A hibajelzés után a részfa gyökerének *aktualis_tipus* attribútumát egészre állítjuk be, azért, hogy a kifejezésben esetlegesen előforduló további hibákat is azonosítani tudjuk.

A 9. szabályban (*Tenyezo* → *azonosito*;) található az alábbi szemantikus függvény:

$$\begin{aligned} Tenyezo.aktualis\_tipus := &(Tenyezo.elvart\_tipus = int \ \&\& \\ &azonosito.aktualis\_tipus = real \ )? \ error(), int \\ &: azonosito.aktualis\_tipus; \end{aligned}$$

Ez a függvény hibajelzést ad, ha a tényező részfa elvárt típusa egész ugyanakkor az *azonosito* kitüntetett szintetizált attribútumának (*aktualis_tipus*) értéke valós. A hibajelzésben itt is pontosan megtudjuk mutatni, hogy melyik azonosító okozta a problémát. Ezután a tényező részfa aktuális típusát egészre állítjuk és folytatjuk az elemzést.

4.6. Rendezett attribútum nyelvtanok

A korábbiakban ismertettünk egy általános több-menetes attribútum kiértékelési stratégiát, illetve az ASE módszert annak eldöntésére, hogy egy tetszőleges attribútum nyelvtan kiértékelhető-e véges számú menetben. A menet-vezérelt attribútum kiértékelés során a fabejárás rögzített, minden elemzési fára ugyanazt a bejárást alkalmazzuk. Ez azt is jelenti, hogy az elemzési fa minden csomópontját ugyanannyiszor látogatjuk meg a bejárás során. Ez nyilvánvalóan nem optimális, hiszen lehetnek olyan részfák, amelyekhez tartozó attribútumok egy menetben is kiértékelhetők lennének, más részfák attribútumainak kiszámításához pedig esetleg 3-4 menetre is szükség van. A továbbiakban megismerkedünk egy olyan attribútum kiértékelési stratégiával, ami megoldja ezt a problémát. Ezt a stratégiát OAG néven ismeri szakirodalom (OAG – Ordered Attribute Grammars [6]). Az attribútum kiértékelési stratégia lényege, hogy nincs előre meghatározott fabejárási sorrend, hanem minden attribútum nyelvtan esetén a nemterminálisok attribútumaira egy rendezést készítünk. A rendezés alapján a nyelvtan szabályaihoz lokális vizit-sorozatokat rendelünk,amelyek vezérlik a fabejárást és attribútum kiértékelést. Azok az attribútum nyelvtanok, amelyek kiértékelhetők az OAG stratégia alapján alkotják az OAG attribútum nyelvtan osztályt. A továbbiakban először ismertetünk egy módszert, amely segítségével tetszőleges attribútum nyelvtanról eldönthető, hogy eleme-e az OAG osztálynak.

4.6.1. OAG teszt

Az OAG teszt első lépésében meghatározzuk egy tetszőleges attribútum nyelvtanra – $AG = (G, A, Val, SF, SC)$ – a direkt függőségek halmazát.

Legyen $p \in P : X_0 \to X_1, \ldots, X_{n_p}$ a nyelvtan szabálya. A szabályhoz tartozó *direkt függőségek halmazát* az alábbi formula definiálja:

$$DP_p = (X_i.a, X_j.b) : \exists f \in SF_p, \text{ hogy } X_j.b = f(\ldots X_i.a \ldots)$$

Az attribútum nyelvtan direkt függőségi halmaza:

$$DP = \cup_{p \in P} DPp$$

Ezután minden $p \in P : X_0 \to X_1, \ldots, X_{n_p}$ szabályra meghatározzuk az *indukált függőségek halmazát*:

$$IDP_p = DP_p \cup \{(X_i.a, X_i.b) : \quad X_i \; p-\text{beli szimbólum}, Y_j \; q-\text{beli szimbólum}, \\ X_i = Y_j \text{ és } (Y_j.a, Y_j.b) \in (IDP_q)^+\}$$

Az indukált függőségek halmazát úgy kapjuk meg, hogy a direkt függőségek halmazát kiegészítjük a szabályban előforduló szimbólumok saját attribútumaik közötti függőségekkel. Ez annyit jelent, hogy ha egy szimbólum több szabályban is előfordul, akkor attribútumai közötti függőségeket minden előforduláshoz uniózzuk. A szimbólum attribútumai közötti függőségeket a direkt függőségi gráf tranzitív lezártja alapján számítjuk ki.

A teljes indukált függőségi halmazt az alábbi formula adja meg:

$$IDP = \cup_{p \in P} IDP_p$$

Ezután meghatározzuk a szimbólumok indukált függőségi halmazát:

$$IDS_x = \{(X.a, X.b) : \; X_i = X_j \text{ és } (X_i.a, X_i.b) \in IDP_p\}$$

Ez a halmaz minden szimbólumra megadja, hogy az attribútumai között milyen függőségek vannak. Ezek uniójával áll elő az IDS halmaz:

$$IDS = \cup_{x \in V} IDS_x$$

A szimbólumokra bevezetett indukált függőségi halmaz alapján a szimbólumok attribútumait osztályokba soroljuk. Ezt az osztályozást akkor tudjuk megtenni, ha az IDS gráf nem tartalmaz kört. Ellenkező esetben az $AG \notin OAG$. A szimbólumok attribútumainak osztályozását az alábbi formulák definiálják:

$$\begin{aligned} A_{X,1} = \quad & \{X.a \in AS_X : \; \neg(\exists X.b), \text{ hogy } (X.a, X.b) \in IDS_X\} \\ A_{X,2n} = \quad & \{X.a \in AI_X : \; \text{ha } \exists (X.a, X.b) \in IDS_X \text{ akkor} \\ & X.b \in A_{X,k}, k \leq 2n\} \; - \; \cup_{k=1}^{2n-1} A_{X,k} \\ A_{X,2n+1} = \quad & \{X.a \in AS_X : \; \text{ha } \exists (X.a, X.b) \in IDS_X \text{ akkor} \\ & X.b \in A_{X,k}, k \leq 2n+1\} \; - \; \cup_{k=1}^{2n} A_{X,k} \end{aligned}$$

Az osztályozás szerint páratlan indexű osztályba kerülnek a szimbólum szintetizált, párosba pedig az örökölt attribútumai. Az első osztályba kerülnek azok a szintetizált attributumok, amiktől az adott szimbólum egyik attribútuma sem függ. A $2n$-edik osztályba kerülnek azok az örökölt attribútumok, amelyektől csak olyan attribútumai függhetnek az adott szimbólumnak,amik már korábbi osztályokban vannak. A feltételt teljesítő örökölt attribútumok közül csak azokat hagyjuk a $2n$-edik osztályba , amelyeket nem soroltunk be korábbi osztályokba. Vagyis a korábbi osztályokba besorolt attribútumokat elvesszük a $2n$-dik osztályból. Ezt jelöli a – kivonás.

Az osztályozás alapján láthatjuk, hogy a magasabb indexű osztályba kerülő attribútum értékeket korábban számíthatjuk mint az alacsonyabb indexűeket. Ezért az osztályozás szerint bevezetünk új függőségéket és így kapjuk meg a *szimbólumok függőségi* halmazát:

$$\begin{aligned} DS_X &= IDS_X \cup \{(X.a, X.b) : X.a \in A_{X,k}; \text{és } X.b \in A_{X,k-1}\} \\ DS &= \cup_{x \in V} DS_x \end{aligned}$$

A szimbólumokra bevezetett új függőségeket visszavezetjük a direkt függőségi gráfba és így megkapjuk a *kiterjesztett függőségi* gráfot:

$$\begin{aligned} EDP_p &= DP_p \cup \{(X_i.a, X_i.b) : \quad X_i \; p - \text{beli szimbólum} \\ &\qquad\qquad\qquad\qquad\quad X = X_i \text{ és } (X.a, X.b) \in DS_X\} \\ EDP &= \cup_{p \in P} EDP_p \end{aligned}$$

Ha az EDP^+ (EDP tranzitív lezártja) nem tartalmaz kört, akkor $AG \in OAG$.

4.6.2. OAG vizit-sorozat számítása

Az OAG teszt végrehajtásával még csak azt tudjuk eldönteni egy tetszőleges AG attribútum nyelvtanról, hogy benne van-e az OAG nyelvosztályban. Amennyiben $AG \in OAG$, akkor az AG minden szabályához vizit-sorozatot számítunk. A $p \in P : X_0 \to X_1, \ldots, X_{n_p}$ szabályhoz rendelt vizit-sorozatot az EDP_p gráf éleiből kapjuk. A gráfban előforduló definiáló attribútum előfordulásokat változatlanul hagyjuk, az alkalmazott attribútum előfordulások helyébe viszont vizit utasításokat írunk. Szemléletesen ez annyit jelent, hogy a p szabály definiáló attribútum előfordulásait ott számítjuk ki, ahol a p szabályt alkalmaztuk az elemzési fában az alkalmazott attribútum előfordulásainak kiszámításához viszont el kell mozdulnunk az attribútumos fában. Gyökér szimbólum alkalmazott attribútumánál felfelé mozdulunk, jobboldali szimbólumok esetében részfát látogatunk meg. Tehát a vizit-sorozatok elemei definiáló attribútum előfordulásokból és vizit utasításokból álló párok. Formálisan:

$$\begin{aligned} VS_p &= AV_p \times AV_p \text{ ahol } AV_p = AF_p \cup V_p \\ V_p &= \{V_{k,i} \mid 1 \le k \le n_{V_X} X_i = X\} \\ V_{k,0} &: k\text{-adik vizit a gyökérre} \\ V_{k,i} &: k\text{-adik vizit} X_i\text{-re} \end{aligned}$$

A $V_{k,i}$ szimbólum reprezentálja a vizit utasításokat. Amint azt a korábbiakban említettük az OAG kiértékelési stratégiában nincs előre rögzítve, hogy egy elemzési fa csomópontot hányszor látogatunk meg. A vizitek számát az határozza meg, hogy a csomópontnak megfelelő nyelvtani szimbólum attribútumait hány osztályba soroltuk az OAG teszt során. Konkrétan, ha m_X jelöli az X szimbólumhoz tartozó attribútumok (A_X halmaz) osztályainak számát, akkor jelölje $n_{V_X} = f_x div 2$ az X szimbólum attribútumainak kiértékeléséhez szükséges menetek számát, ahol $f_x \ge m_X$ legkisebb páros számot jelenti. Ez annyit jelent, hogy az OAG algoritmus minden olyan elemzési csomópontot, amit az X szimbólummal címkézünk n_{V_X} menetben tudja kiértékelni. Egy menetben a csomópontot kétszer látogatjuk meg, egyszer, amikor belépünk a csomópont részfájába, másodszor, amikor elhagyjuk azt.

Az attribútum előfordulások leképezését az alábbi formula definiálja:

$$MAPVS(X_i.a) = \begin{cases} X_i.a \text{ ha } X_i.a \in AF_p \\ V_{k,i} \text{ ha } X_i = X, X_i.a \in (A_{X,m} \cap AC_p) \\ \qquad k = (f_X - m_X + 1)\ div 2,\ k > 0 \\ \text{nem definiált, ha } X_i = X\ X_i.a \in (A_{X,m} \cap ACp) \text{ és } k = 0 \end{cases}$$

A $k = 0$ esetben azért nincs szükség vizit utasításra, mivel feltételezhetjük, hogy az adott alkalmazott attribútum előfordulás értéke az elemzési fa építéssel együtt meghatározható. Miután a $MAPVS(X_i.a)$ leképezést végrehajtjuk a p szabály attribútum előfordulásain az EDP_p függőségi gráf alapján a definiáló attribútum előfordulásokra és az alkalmazott előfordulásokból kapott vizit utasításokra egy részben rendezett gráfot kapunk. Ezt a részben rendezett gráfot kiegészítjük olyan élekkel, hogy VS_p lineárisan rendezett legyen. Vagyis,

$$\begin{aligned} VSp = \quad & \{(MAPVS(X_i.a), MAPVS(X_j.b)) \mid (X_i.a, X_j.b) \in EDP_p\} \cup \\ & \{\text{tetszőleges olyan él, hogy } VS_p \text{ lineárisan rendezett legyen, és} \\ & V_{k,0}(k = n_{V_X}) \text{ a legnagyobb elem legyen}\} \end{aligned}$$

A 4.4. ábrán szereplő attribútum nyelvtan vizit sorozatait a 4.12. ábra szemlélteti.

$VS_1 = v_{1,1}\ N_1.r\ v_{2,1}\ v_{1,2}\ N_2.r\ v_{2,2}\ v\ v_{1,0}$

$VS_2 = v_{1,1}\ l\ v_{1,0}\ N_1.r\ v_{2,1}\ B.r\ v_{1,2}\ v\ v_{2,0}$

$VS_3 = l\ v_{1,0}\ v\ v_{2,0}$

$VS_4 = v\ v_{1,0}$

$VS_5 = v\ v_{1,0}$

4.12. ábra. A bináris példa vizit-sorozatai

A továbbiakban megmutatjuk, hogy a vizit-sorozatok alapján, hogyan lehet az OAG attribútum kiértékelő stratégiát implementálni.

Feltételezzük, hogy az attribútumos fa egy csomópontja tartalmazza:

- Az attribútum példányokat
- Referenciát a leszármazottakra
- A csomópontnál alkalmazott szabály sorszámát

A vizit-sorozatokat annyi részre osztjuk, ahány gyökér vizit van bennük. Tehát a bináris példánál a VS_2 és VS_3 vizit-sorozatokat két részre a többieket egy részre osztjuk. Vagyis,

$$VS_p = VS_{p,1}, VS_{p,2}, \ldots, VS_{p,m}$$

Minden részre egy eljárást írunk, az alábbiak szerint:

```
Procedure pi(ref node X_0)
begin
...
X_i.a = f(...)
...
q_k(X_j)   ( A v_{k,j} vizit utasításra).
...
end
```

Vagyis a vizit-sorozatokban szereplő definiáló attribútum előfordulások értékeinek kiszámítására meghívjuk a megfelelő szemantikus függvényeket, a $v_{k,j}$ vizit utasításra pedig egy $q_k(X_j)$ eljárás hívást hajtunk végre. Ha az X_j szimbólumnak megfelelő nemterminális csak a q-adik szabály baloldalán fordul elő, akkor a q_k eljárás egyértelműen meghatározható.

Amennyiben az X_j több szabály baloldalán is előfordul, akkor a megfelelő eljárás kiválasztása az alábbi case utasítással valósítható meg:

```
Case X_j.szabály_sorszám of
q_1 :  q_{1_k}(X_j)
...
q_n  :  q_{n_k}(X_j)
```

4.7. Attribútum nyelvtanok osztályozása

A korábbiakban ismertettük a nem-cirkuláris attribútum nyelvtanok néhány részosztályát. Nevezetesen a többmenetes váltakozva balról-jobbra és jobbról-balra fa bejáráson alapuló *ASE* illetve az *OAG* osztályokat. Az olyan attribútum nyelvtanokat, amelyek csak szintetizált attribútumokat tartalmaznak *S-attribútum nyelvtanoknak* (röviden *S-attr*) nevezzük. Ha feltételezzük, hogy normál formában adjuk meg a szemantikus egyenleteket, akkor az S-attr nyelvtanok nyilvánvalóan nem-cirkulárisak, hiszen minden szemantikus egyenlet a baloldali nemterminális szintetizált attribútumát számítja ki a jobboldali szimbólumok szintetizált attribútumai felhasználásával és ez a 4.3. ábrán ismertetett attribútum kiértékelési stratégiával mindig megvalósítható. További nevezetes nem-cirkuláris attribútum nyelvtan osztály az egy-menetben kiértékelhető attribútum nyelvtanok osztálya. Az ASE nyelvosztály ezen részosztályát szokás *L-attr osztálynak* nevezni. Az L-attr osztályba való tartozás az alábbi egyszerű feltétellel eldönthető:

Egy attribútum nyelvtan eleme az L-attr osztálynak, ha minden $p \in P : X_0 \to X_1, \ldots, X_{n_p}$ szabály minden $(X_i.a, X_j.b)$ függőségére teljesül:

$$(1)\ \text{Ha } X_j.b \in AS_{X_0},\ \text{akkor } X_i.a \in AI_{X_0} \text{ vagy } X_i.a \in AS_{X_i} (1 \le i \le n_p)$$
$$(2)\ \text{Ha } X_j.b \in AI_{X_i}, (1 \le i \le n_p),\ \text{akkor } X_i.a \in AI_{X_0} \text{ vagy } X_i.a \in AS_{X_i} (1 \le i < j)$$

Nyilvánvalóan teljesül, hogy minden S-attr nyelvtan egyben L-attr nyelvtan is és minden L-attr nyelvtan egyben ASE nyelvtan is. Az is triviális, hogy a tartalmazás valódi, tehát van olyan attribútum nyelvtan, ami ASE de nem L-attr-os és van olyan L-attr-os nyelvtan, ami nem S-attr-os. Az ASE és az OAG nyelvosztály közötti tartalmazás kérdésének eldöntése már nem ennyire egyszerű.

A 4.13. ábrán láthatunk egy olyan példa attribútum nyelvtant, ami OAG de nem ASE.

1: $S \to Y$;
do
$Y.c := Y.b$;
end

2: $Y_0 \to tY_1$;
do
$Y_0.b = Y_0.a$;
$Y_0.d = Y_1.d$;
$Y_1.c = Y_1.b$;
$Y_1.a = Y_0.c$;
end

3: $Y_0 \to t$;
do
$Y_0.b = Y_0.a$;
$Y_0.d = Y_0.c$;
end

4.13. ábra. Példa olyan attribútum nyelvtanra, ami OAG de nem ASE

A példában csak a szintaktikus szabályokat és hozzájuk tartozó szemantikus egyenleteket adtuk meg. Mivel az attribútum nyelvtan normál formában van megadva ezért könnyen meghatározható, hogy melyik attribútum a szintetizált és örökölt. Ez alapján pedig as ASE illetve OAG teszt végrehajtható. (A tesztek végrehajtását az olvasóra bízzuk.)

Úgy gondolhatnánk, hogy mivel az OAG kiértékelő jóval rugalmasabb fabejárási stratégiával dolgozik mint az ASE, ezért minden OAG nyelvtan egyben ASE nyelvtan is. Azonban ez nincs így, a 4.14. ábrán láthatunk egy olyan attribútum nyelvtant, ami ASE de nem OAG.

Ha elvégezzük az OAG tesztet erre az attribútum nyelvtanra, akkor azt találjuk, hogy az EDP gráf cirkularitását a nemterminálisok osztályozásából keletkező plusz élek okozzák. Ez elkerülhető azzal, ha az osztályozás előtt bevezetünk olyan éleket, amelyek az ASE osztályok alapján keletkeznek. Vagyis az IDS halmazt bővítjük az alábbi élekkel:

$$IDS_{ASE} = \{(X_i.a, X_i.b) \mid k_a < k_b \text{ vagy } (k_a = k_b \text{ és } X_i.a \in AI, X_i.b \in AS)\}$$

Ezt a transzformációt alkalmazva az ilyen attribútum nyelvtanok is teljesítik az OAG feltételt.

1: $Z \to XY$;
do
$X.c := Y.h$;
$Y.e := X.d$;
end

2: $X \to t$;
do
$X.b = X.a$;
$X.d = X.c$;
end

3: $Y \to s$;
do
$Y.e = Y.e$;
$Y.h = Y.g$;
end

4.14. ábra. Példa olyan attribútum nyelvtanra, ami ASE de nem OAG

4.8. Közbülső kód generálása

A továbbiakban megmutatjuk, hogyan lehet attribútum nyelvtanok segítségével közbülső kódot előállítani. A korábbiakban már említettük, hogy a fordítóprogramok általában nem közvetlenül gépi kódot generálnak, hanem egy általánosabb géptől független reprezentációs formára transzformálják az analízis fázisban előállított elemzési fát. A három címes kód a leginkább elterjedt közbülső reprezentációs forma. Nevét onnét kapta, hogy legfeljebb három operandus fordulhat elő az utasításokban. A 4.15. ábrán láthatunk példát a tipikus három címes utasításokra.

A 4.16. ábrán bemutatunk egy egyszerű S-attribútumos nyelvtant, amely segítségével három címes kód generálható a korábbiakban bevezetett értékadó utasításra.

A 4.16. ábrán szereplő attribútum nyelvtanban az *out* és *end* kulcsszavak közötti print eljárás hívásokkal valósítjuk meg a három címes kód tényleges generálását. Ezek a print utasítások tekinthetők szemantikus feltételeknek (lásd 4.1 fejezet), hiszen nem definiálnak attribútum előfordulás értéket. A három címes kódban temporális változókban tároljuk a részkifejezések értékeit. A nemterminálisok *temp_hely* attribútumai ezeket a temporális változókat azonosítják. A newtemp eljárással egy új temporális változót hozunk létre. Az *azonosito.azon_hely* kitüntetett szintetizált attribútum az adott változó memória helyét azonosítja (feltesszük, hogy ez rendelkezésre áll).

```
a) x := y op z
b) x := op y
c) x := y
d) if x relop y goto L
e) param x1
   ...
   param xn
   call p,n
f) x := y[i], x[i] := y
g) x := &y (y értéke y címe)
   x := *y (y értéke az y pointer által mutatott
            cím tartalmával lesz egyenlő)
  *x := y  (az y pointer által mutatott objektum
            tartalma lesz egyenlő y tartalmával)
```

4.15. ábra. Három címes utasítások

4.9. Szimbólumtábla kezelése

A szimbólumtábla a fordítóprogramok nagyon fontos része, a fordítás minden fázisában használjuk. A szimbólumtáblában az adott nyelv azonosítóiról (pl. típusok nevei,változók, függvény nevek) tárolunk információkat. Ilyen információ lehet például egy változó esetén a típusa illetve a relatív memória címe. A 4.17. ábrán ismertetjük egy olyan attribútum nyelvtan vázlatát, amely segítségével lehetőség van szimbólumtábla építésére. Feltételezzük, hogy a példa szerinti nyelvben lehetőség van egymásba skatulyázott eljárások használatára. Vagyis a nyelv egymástól $'$;$'$-vel elválasztott deklarációkat tartalmaz, amik eljárás illetve változó deklarációk lehetnek. Az eljárás deklarációból (3. szabály) láthatjuk, hogy az eljáráson belül további deklarációkat adhatunk meg. A szabályban szereplő *S* szimbólum az eljárás utasítás blokkját jelöli, amit ebben a példában nem részletezünk. A változó deklarációban (4-7. szabály) adjuk meg egy változó típusát, relatív memória címét és azt, hogy mennyi helyet foglal el a memóriában. Minden eljárásra egy külön szimbólumtáblát építünk, amibe tároljuk az eljárásban deklarált változók adatait és referenciát az adott eljárásban deklarált eljárások szimbólumtábláira. Minden szimbólumtábla tartalmaz egy referenciát a tartalmazási struktúra szerinti ősére.

Az attribútum nyelvtanban két szintetizált-örökölt attribútum párt használunk. Az *s_offset*, *i_offset* attribútumokban tároljuk a főprogramban illetve az egyes eljárásokban deklarált változók tárolási címének meghatározásához szükséges szabadhely értéket. Az *s_symt*, *i_symt* szintetizált-örökölt attribútum pár az adott környezetben aktuálisan érvényes szimbólumtábla állapotot mutatja meg. A programba való belépéskor (1.szabály) a *mktabla()* eljárással létrehozunk egy üres szimbólumtáblát és az aktuális szabadhely értéket 0-ra állítjuk.

Attribútum nyelvtan: harom_cimes_kod;

Szintetizált attribútum nevek:
temp_hely :int;
azon_hely :int;

Nemterminálisok és attributumok:
Utasitas:;
Kifejezes: temp_hely;
Tag: temp_hely;
Tenyezo: temp_hely;

Tokenek: azonosito = betu (betu | szam | '_')* : azon_hely;

Terminálisok: ':=', '+' , '-', '*' , '/', '(' , ')'

Szabályok és szemantikus függvények:

1: Utasitas → azonosito := Kifejezes;
do
out
print (azonosito.azon_hely :=Kifejezes.temp_hely);
end

2: Kifejezes_0 → Kifejezes_1+ Tag;
do
Kifejezes_0. temp_hely := newtemp;
out
print (Kifejezes_0.temp_hely ':=' Kifejezes_1.temp_hely '+' Tag.temp_hely); **end**

3: Kifejezes_0 → Kifejezes_1 - Tag;
do
Kifejezes_0.temp_hely := newtemp;
out
print (Kifejezes_0.temp_hely ':=' Kifejezes_1.temp_hely '-' Tag.temp_hely); **end**

4: Kifejezes → Tag;
do
Kifejezes.temp_hely := Tag.temp_hely;
end

5: Tag_0 → Tag_1 * Tenyezo;
do
Tag_0.temp_hely := newtemp;
out
print (Tag_0.temp_hely ':=' Tag_1.temp_hely '*' Tenyezo.temp_hely);
end

6: Tag_0 → Tag_1 / Tenyezo;
do
Tag_0.temp_hely := newtemp;
out
print (Tag_0.temp_hely ':=' Tag_1.temp_hely '/' Tenyezo.temp_hely);
end

7: Tag → Tenyezo;
do
Tag.temp_hely := Tenyezo.temp_hely;
end

8: Tenyezo → '(' Kifejezes ')';
do
Tenyezo.temp_hely := Kifejezes.temp_hely;
end

9: Tenyezo → azonosito;
do
Tenyezo.temp_hely := azonosito.azon_hely;
end

4.16. ábra. Három címes kód generálása attribútum nyelvtannal

A 2.szabály a deklarációs lista megadására szolgál. Itt láthatjuk, hogy a szintetizált-örökölt attribútum párokkal, hogyan valósítható meg az aktuális szabadhely érték illetve szimbólumtábla állapot megadása. A 3.szabály új eljárás deklarációjára szolgál. Ekkor a *mktable()* eljárással egy új szimbólumtáblát hozunk létre, és az aktuális szabadhely értéket nullázzuk. Az *enterproc()* eljárással az ős szimbólumtáblába teszünk egy bejegyzést az eljárás deklarációról, amiben megadjuk az adott eljárás szimbólumtáblájának az eljárás feldolgozása utáni állapotát ($Dekl_1.s\_symt$). A 4. szabályban egy változót deklarálunk. A szabadhely értéket a változó típusának megfelelő mérettel megnöveljük. Az *enter()* eljárással a változó adatait (név,típus,memóriacím) tároljuk az aktuális szimbólumtáblába. Az 5. és 6. szabály az aktuális típus és méret megadására szolgál.

4.10. Feladatok

1. Adjunk meg egy-menetes attribútum nyelvtant 4.4. ábrán szereplő bináris attribútum nyelvtanra. Az `10.1` inputra adjuk meg a kiértékelést.

2. A 4.10. és 4.11. ábrán szereplő típus kompatibilitást ellenőrző attribútum nyelvtannal építsünk attribútumos elemzési fát az $a := b + c/d$ inputra. Az a,b,c és d egész változók. Értékeljük ki az attribútumos fát.

3. Hajtsuk végre az OAG tesztet a 4.4. ábrán szereplő bináris attribútum nyelvtanra. Számítsunk vizit sorozatot és adjuk meg a OAG kiértékelő programjának vázát.

4. Hajtsuk végre az OAG és ASE tesztet a 4.13. ábrán szereplő attribútum nyelvtanra.

5. A 4.16. ábrán szereplő attribútum nyelvtannal építsünk attribútumos fát az $a := b+c/d$ inputra. Az attribútumos fa kiértékelésével adjuk meg a generált háromcímes kódot

1: Program → Dekl ;
do
Dekl.i_offset := 0;
Dekl.i_symt := mktable(nil);
end

2: Dekl_0 → Dekl_1 ';' Dekl_2;
do
Dekl_0.s_offset := Dekl_2.s_offset;
Dekl_1.i_offset := Dekl_0.i_offset;
Dekl_2.i_offset := Dekl_1.s_offset;
Dekl_0.s_symt := Dekl_2.s_symt;
Dekl_1.i_symt := Dekl_0.i_symt;
Dekl_2.i_symt := Dekl_1.s_symt;
end

3: Dekl_0 → **proc** azon ':' Dekl_1 ';' S;
do
Dekl_0.s_offset := Dekl_0.i_offset;
Dekl_0.s_symt := Dekl_0.i_symt;
Dekl_1.i_offset := 0;
Dekl_1.i_symt := mktable(Dekl_0.i_symt);
S.i_symt := Dekl_1.s_symt;
out
enterproc (Dekl_0.i_symt, azon.név, Dekl_1.s_symt);
end

4: Dekl → azon ':' Típus;
do
Dekl.s_offset := Dekl.i_offset + Típus.méret;
Dekl.s_symt := Dekl.i_symt;
out
enter(Dekl.i_symt, azon.név, Típus.tip, Dekl.i_offset);
end

5: Típus → real;
do
Típus.tip := 'real';
Típus.méret := 8;
end

6: Típus → int;
do
Típus.tip := 'int';
Típus.méret := 2;
end

4.17. ábra. Szimbólumtábla kezelés attribútum nyelvtannal

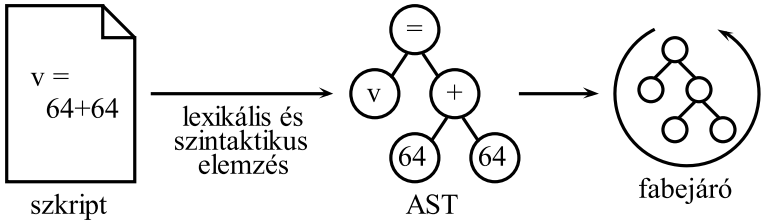
5. fejezet

Interpretált kód és röpfordítás

A korábbi fejezetekben bemutatott folyamat során a program forráskódja több lépésben egy kiválasztott architektúra gépi kódjára fordul, amit egy megfelelő processzor közvetlenül végrehajt. A programok végrehajtására azonban van egy másik lehetőség is, amikor a programot közvetetten hajtjuk végre egy értelmező, azaz interpreter vagy virtuális gép segítségével. Azokat a programozási nyelveket, amelyek programjait gép kódra fordítjuk, gyakran fordított programozási nyelveknek nevezzük, míg amelyek programjait értelmezővel hajtjuk végre, azokat interpretált vagy szkript nyelveknek hívjuk. A két kategória között a határ azonban nem átjárhatatlan, hiszen vannak hagyományosan fordítottnak tekintett nyelvek, amiknek interpretált megvalósítása is létezik (ilyen például a C nyelv is), de vannak technikák szkript programok gépi kódra fordítására is (pl.: futás közbeni, avagy röpfordítás – angolul just-in-time compilation, JIT).

Szkript nyelveket manapság igen sok területen használnak. A két legismertebb terület az operációs rendszerek parancssori értelmezői (pl.: a Unix rendszereken gyakran használt bash – ami egyszerre jelöli az értelmezőt és annak szkript nyelvét –, vagy a Microsoft rendszerek batch fájljai) és a web programozási nyelvei (ezek közül a legismertebb a JavaScript, de ide tartozik a Flash platform ActionScript programozási nyelve és a szerveroldali PHP is). Emellett sokszor ágyaznak még be interpretereket gépi kódra fordított alkalmazásokba, hogy megkönnyítsék a programhoz bővítmények készítését.

Az interpretált nyelvek széles körű elterjedésének oka, hogy rendelkeznek néhány olyan pozitív tulajdonsággal, amellyel a fordított nyelvek nem. Ezek közül a legfontosabb a platformfüggetlenség, azaz hogy egy interpretált nyelv programja módosítás és újrafordítás nélkül futtatható tetszőleges platformon (legalábbis minden olyan platformon, amire az értelmezőt elkészítették). Jellemző még az interpretált nyelvekre, hogy a programok önmagukat – a deklarált adatszerkezeteket vagy modulokat, metódusokat – futás közben vizsgálhatják (ez a reflexió), valamint az interpreternek bemenetként kapott programkódot adhatnak át végrehajtásra (egy – általában – eval nevű beépített függvény segítségével). Az interpretált nyelvek gyakori előnyei továbbá (a teljesség igénye nélkül): a változók dinamikus típusossága és sokszor dinamikus érvényességi köre, valamint a fordított nyelvekhez képest



5.1. ábra. Fa alapú interpreter működése

gyorsabb fejlesztési ciklus. Természetesen nem csak előnyei vannak az interpretált végrehajtásnak. Egy program interpretált nyelven megírt változata sokszor lényegesen lassabb egy közvetlenül gépi kódra fordított változatánál.

A fejezet további részeiben példák segítségével áttekintjük, hogy milyen megoldások léteznek interpreterek megvalósítására.

5.1. Fa alapú interpreter

Egy programozási nyelv interpretálására a legegyszerűbb megközelítés az elemző által felépített fa (AST) bejárása. Ennél a megközelítésnél a fa bejárása közben történik meg a nyelv kifejezéseit és utasításait reprezentáló részfák kiértékelése, végrehajtása. Az 5.1. ábra sematikusan bemutatja egy fa alapú interpreter működését.

> **Példa:** A lexikális és szintaktikus elemzőkről szóló 3. fejezetben már láthattunk egy példát az egyszerű kifejezésekből és értékadó utasításokból álló nyelv AST-jén végzett kiértékelésre.

5.2. Bájtkód alapú interpreter

Interpretált nyelvek esetén a szintaktikus elemzés során épített AST-ből, ha az nem közvetlenül kerül kiértékelésre, egy virtuális gép utasításkészletére szokás a programot fordítani, majd az így kapott virtuális utasításokat végrehajtani. A virtuális utasításkészletet és az arra fordított programot általában bájtkódnak hívjuk (utalva arra, hogy a virtuális utasítások műveleti kódja rendszerint egy bájton ábrázolható).

A fa alapú értelmezővel ellentétben a bájtkód alapú interpreter esetén az elemzés és a végrehajtás szétválhat. Megtehető, hogy a forrásprogram elemzése és a bájtkód előállítása, azaz a fordítás, a program futtatásától teljesen elkülönülten, azt lényegesen megelőzve, akár teljesen más számítógépen történjen. Ehhez természetesen platformfüggetlen bájtkódformátumra van szükség. Ilyenkor az interpreter valójában csak a bájtkódot végrehajtó virtuális gép, az elemző-fordító modult nem tekintjük a részének. (Példa az elemzést és a végrehajtást szétválasztó megközelítésre a Java nyelv fordítóprogramja, az általa előállított

bájtkód és az azt végrehajtó Java virtuális gép. Egyes JavaScript megvalósítások pedig a másik, az elemzést és bájtkód-generálást együtt, futásidőben végző megközelítést követik.)

Egy bájtkód alapú értelmező központi része a végrehajtó modul, más néven motor. A motor működése nagyban hasonlít egy valós processzorra: az utasításbetöltés, dekódolás, végrehajtás ciklusát futtatja. A végrehajtó motor egy változóban, mint egy virtuális regiszterben, tárolja a végrehajtandó utasítás pozícióját. (Ennek gyakori nevei virtuális utasításszámláló – angolul virtual program counter, vPC –, vagy virtuális utasításmutató – angolul virtual instruction pointer, vIP.) Az első lépés az utasításmutató által megadott utasítás betöltése, majd következik az utasítás dekódolása (azaz annak a programrésznek a meghatározása, amely a virtuális utasítást megvalósítja), végül a végrehajtás. Az utolsó lépésnek része az utasítás esetleges operandusainak a betöltése, a számítás eredményének tárolása, és a következőként végrehajtandó utasítás meghatározása is.

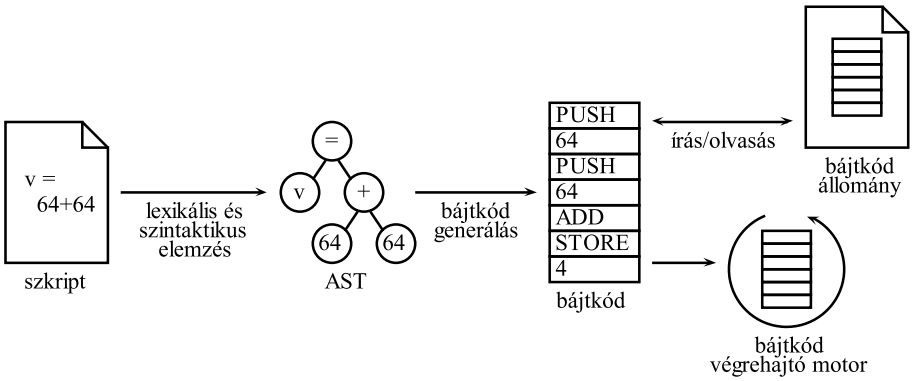
A bájtkódok operandusainak típusától függően kétfajta virtuális gép architektúrát különböztetünk meg: a verem alapút és a regiszter alapút. Az első esetben az operandusok az operandusveremben helyezkednek el, végrehajtáskor annak tetejéről kerülnek levételre, majd az eredmény a verem tetejére kerül ráhelyezésre. A második esetben a végrehajtó motor egy (nem feltétlenül véges elemszámú) virtuális regiszterkészlettel rendelkezik, és az utasítások meghatározzák a végrehajtandó művelet forrás- és célregiszterét (vagy regisztereit). Mindkét architektúra esetén a fő operandustípus mellett szokás közvetlen (azaz a bájtkódban elhelyezett) konstans operandust is megengedni (legalább némely utasítás esetén), valamint hozzáférést biztosítani valamilyen egyéb adatszerkezetekhez, objektumokhoz (sokszor így nyújtva lehetőséget a kommunikációra az interpreter és a beágyazó környezete között).

Példa: Definiáljuk egy verem alapú architektúra utasításkészletének 3 utasítását:

- a PUSH művelet egy közvetlen szám operandussal rendelkezik, amit az operandusverem tetejére helyez,
- az ADD műveletnek nincsen közvetlen operandusa, viszont az operandusverem két legfelső elemét összeadja, és az eredményt a verem tetejére írja,
- a STORE művelet pedig a közvetlen operandusként kapott számmal azonosít egy külső memóriaterületet, ahová a verem tetejéről leemelt értéket letárolja.

Legyen a PUSH művelet kódja 0, az ADD műveleté 1, a STORE műveleté 2, ekkor a {0, 64, 0, 64, 1, 2, 4} bájtkódot PUSH 64, PUSH 64, ADD, STORE 4 utasítássorozatként értelmezhetjük, aminek eredményeképpen a 4 számmal azonosított memóriaterületre az összeadás eredményét, 128-at írunk.

Ez a bájtkód még nem alkalmas arra, hogy a kifejezésekből és értékadásokból álló példanyelvünket lefordítsuk rá, de könnyen belátható, hogy a nyelv egy jól meghatározott részhalmazának – ahol minden kifejezés csupa konstansból álló összeadás – már megfelel. Ez a szűkített kódgenerálás meglehetősen egyszerű,



5.2. ábra. Bájtkód alapú interpreter működése

így itt most nem mutatunk rá példát. A fejezet későbbi részében található olyan program, amely kibővíti ezt a bájtkódot minden szükséges művelettel és arra is példát ad, hogy hogyan lehet a kibővített bájtkódra az AST-ből kódot generálni. Egy ilyen komplett rendszer működési modelljét mutatja be az 5.2. ábra.

A bájkód alapú interpreterek megvalósítása egyetlen nagy, ciklusba ágyazott többszörös elágazáson (**switch** utasításon) alapul, ahol az elágazás minden egyes ága egy virtuális utasításnak felel meg.

Példa: Az 5.3. ábrán a fenti bájtkódot végrehajtó motor programjának egy kisebb részlete látható. A motor bemenetként kapja a végrehajtható bájtkódot (code) és a beágyazó környezettel adatcserét lehetővét tevő adatszerkezetet (ctx). A stack változó az operandusvermet tartalmazza, míg a vPC a virtuális utasításmutató, amely az interpreter ciklus minden iterációjának kezdetén a végrehajtandó utasítás bájtkódbeli indexét tartalmazza.

5.3. Szálvezérelt interpreterek

A szálvezérelt interpreterek[1] a bájtkód alapú interpretereket fejlesztik tovább úgy, hogy kiiktatják a végrehajtó motorból a ciklusba ágyazott elágazást és magát a ciklust is, így gyorsítva a futást.

A következőkben áttekintjük a leggyakrabban alkalmazott szálvezérlési modelleket.

[1] A „threaded interpreter” kifejezésnek még nincs a magyar nyelvű szakirodalomban elfogadott fordítása. A jegyzetben a threaded interpreter, threaded code kifejezéseket szálvezérelt interpreternek, szálvezérelt kódnak, a threading modelt szálvezérlési modellnek fordítjuk, míg a token-threaded, direct-threaded, stb. megnevezéseket tokenvezérelt és direkt vezérelt kifejezésekre magyarítjuk.

```
public void execute(byte[] code, StackContext ctx) {
    Stack<Integer> stack = new Stack<Integer>();
    int vPC = 0;
    while (true) {
        switch (code[vPC++]) {
            case PUSH:
                stack.push((int)code[vPC++]);
                break;
            case ADD:
                stack.push(stack.pop() + stack.pop());
                break;
            case STORE:
                ctx.setVariable(code[vPC++], stack.pop());
                break;
        }
    }
}
```

5.3. ábra. Verem alapú bájtkód végrehajtó motorjának részlete

5.3.1. Tokenvezérelt interpreter

Egy tokenvezérelt értelmezőben – a bájtkód alapú interpreternél használt többszörös elágazás helyett – minden virtuális utasítás végrehajtása után azonnal a következőként végrehajtandó utasítást megvalósító kódrészletre kerül a vezérlés. A végrehajtó motorban egy táblázat tárolja a bájtkód utasításait megvalósító programrészletek címét, így a következő virtuális utasítás műveleti kódjával ez a tömb megindexelhető és a kapott címre közvetlenül átadható a vezérlés. (Egy bájtkód alapú interpretert tokenvezérelt interpreterré igen egyszerű továbbfejleszteni, de ehhez az interpretert olyan programozási nyelven szükséges megírni, amely lehetővé teszi a kódcímkék értékként való kezelését.)

> **Példa:** Az 5.4. ábrán a verem alapú bájtkód értelmezőjének tokenvezérelt interpreterré továbbfejlesztett változata látható. (A Java nyelv a kódcímkéket nem képes értékként kezeli, ezért a továbbiakban már GNU C nyelven írt példákkal mutatjuk be az interpreterek megvalósítását. A GNU C ugyanis kibővíti a szabványos ISO C nyelvet kódcímkékre mutató pointerekkel – pl.: a &&label_PUSH kifejezés a label_PUSH kódcímke memóriabeli címét adja vissza –, amik **goto** utasításban felhasználhatók.) A Java és a C közötti nyelvi különbségektől (**int** vPC helyett **char** *vPC, StackContext ctx helyett **int** *vars, Stack<Integer> stack helyett **int** stack[STACK_SIZE] és **int** *vSP) eltekintve a verem alapú bájtkód értelmező és a tokenvezérelt interpreterben a virtuális utasítások megvalósítása nagyrészt azonos. A lényegi különbség a **while** (**true**) **switch** (code[vPC++]) helyett a minden virtuális utasítás implementációja után használt, közvetlen vezérlésátadást végző **goto** *labels[*vPC++].

```
void execute(char *code, int *vars) {
    static void *labels[] = { &&label_PUSH, &&label_ADD, &&label_STORE };

    int stack[STACK_SIZE];
    char *vPC = code;
    int *vSP = stack;
    int lhs, rhs;

    goto *labels[*vPC++];
    label_PUSH:
        *vSP++ = (int)*vPC++;
        goto *labels[*vPC++];
    label_ADD:
        lhs = *(--vSP);
        rhs = *(--vSP);
        *vSP++ = lhs + rhs;
        goto *labels[*vPC++];
    label_STORE:
        vars[*vPC++] = *(--vSP);
        goto *labels[*vPC++];
}
```

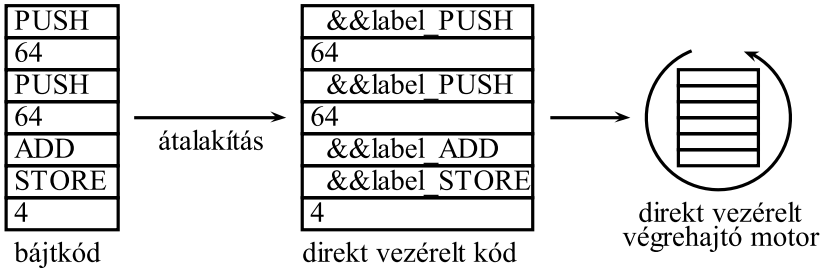
5.4. ábra. Tokenvezérelt interpreter részlete

Úgy tűnhet, hogy a bájtkódról tokenvezérelt végrehajtásra való áttérés az interpreter teljesítményének igen kis mértékű optimalizációja csupán. A következő utasítás címének meghatározása és a vezérlés átadása azonban olyan lépések, amelyek minden egyes virtuális utasítás lefuttatása után végrehajtódnak. Amennyiben a virtuális utasítások nem túl bonyolult szemantikájúak és kevés gépi kódú utasítással végrehajthatók, akkor a virtuális vezérlésátadás gépi kódú utasításigénye már összehasonlítható velük, és a teljes futásidő jelentős részét kiteheti.

5.3.2. Direkt vezérelt interpreter

A direkt vezérelt interpreterek továbbfejlesztik a tokenvezérelt interpreterek vezérlésátadási megoldását olyan módon, hogy kiiktatják a következő utasítás műveleti kódjával való tömbindexelést. A bájtkódot egy előfeldolgozási lépésben direkt vezérelt kóddá alakítják, ami során a bájtkódban szereplő műveleti kódokat lecserélik a bájtkódokat megvalósító kódrészletek címére. Így a direkt vezérelt kódban a műveleti kódok valójában azonnal végrehajtható programterületre mutatnak. Egy direkt vezérelt interpreter működésének sémája az 5.5. ábrán látható (a lexikális és szintaktikus elemzés, valamint a kódgenerálás lépéseinek újbóli bemutatását az egyszerűség kedvéért elhagytuk).

> **Példa:** Az 5.6. ábrán látható programrészlet bemutatja a példa bájtkód direkt vezérelt kóddá történő átalakítását, valamint az ezt futtató direkt vezérelt interpretert. A programrészlet a tokenvezérelt interpreter példáját viszi tovább,



5.5. ábra. Direkt vezérelt interpreter működése

a **goto** $*$labels[$*$vPC] vezérlésátadásokat **goto** $**$vPC++ szerkezetre egyszerűsítve. (Az átalakítást végző ciklus a korábban példaként hozott {0, 64, 0, 64, 1, 2, 4} bájtkódot {&&label_PUSH, 64, &&label_PUSH, 64, &&label_ADD, &&label_STORE, 4} direkt vezérelt kóddá alakítaná.)

A direkt vezérelt kód futtatása jelentősen gyorsabb lehet, mint a tokenvezérelt vagy bájtkód alapú interpreterek működése, de a megoldásnak költsége is van. Az előfeldolgozó, átalakító lépésnek a lefuttatása plusz időbe kerül, valamint a direkt vezérelt kód tárolása a memóriafogyasztást is megnöveli (hiszen míg a bájtkód reprezentációban egy műveleti kód egy bájton elfér, addig a direkt vezérelt kódban ez egy kódmutatóra cserélődik le, aminek a mérete 32 bites rendszeren 4 bájt, 64 bites rendszeren már 8 bájt).

5.3.3. Környezetvezérelt interpreter

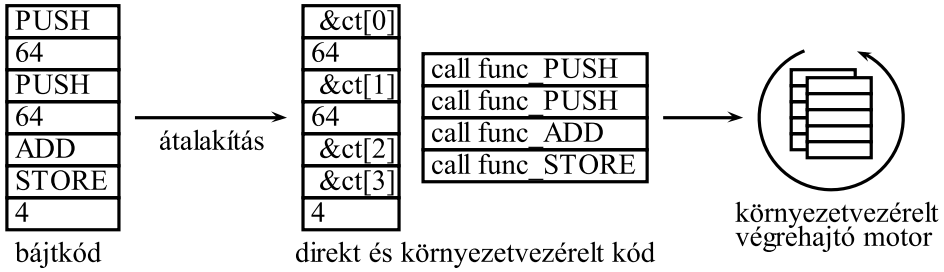
Az interpreterekben a virtuális vezérlésátadás mindig együtt jár az interpreter kódjában történő vezérlésátadással. (Bájtkód alapú interpreternél többszörös elágazással, token- és direkt vezérelt interpreterek esetén pedig címkére történő ugrással.) A modern hardverek már rendelkeznek ugráspredikciós modulokkal, azonban az eddig tárgyalt értelmezők ezt nem tudják kihasználni: a következő virtuális utasítást megvalósító kódrészletre ugró utasítások (látszólag) az interpreter tetszőleges pontjára átadhatják a vezérlést, így a predikció nem tudja meghatározni a legvalószínűbb célpontját az ugrásnak. Egy nem- vagy félreprediktált ugrásnak pedig a modern szuperskalár architektúrákon komoly időköltsége van.

Az ugrásoknál tehát plusz információ, környezet hiányzik, ami alapján a predikció jól működhet. A környezetvezérelt interpreterek azt használják ki, hogy a modern predikciós modulok nem csak az egyszerű ugrások, de a gépi szintű eljáráshívások és szubrutinból való visszatérések célpontjait is igen pontosan képesek (általában a verem alapján) előrejelezni.

A környezetvezérelt interpreterek a direkt vezérelt interpreterekhez képest még egy átalakítást végeznek a kódon: a direkt vezérelt kód mellett egy ú.n. környezetvezérelt kódot (vagy táblát) is létrehoznak, amibe a futtató platform gépi kódjának megfelelő utasításokat generálnak. Minden műveleti kód megvalósítása külön szubrutinban történik, és egy bájtkód minden virtuális utasításából egy gépi függvényhívás készül a műveleti kódjának megfelelő

```
void execute(char *code, int codesize, int *vars) {
    static void *labels[] = { &&label_PUSH, &&label_ADD, &&label_STORE };
    static int operands[] = { 1, 0, 1 };

    int stack[STACK_SIZE];
    int *direct_thread = (int*)malloc(sizeof(int)*codesize);
    char *cp = code;
    int *dtp = direct_thread;
    int *vPC = direct_thread;
    int *vSP = stack;
    int lhs, rhs;

    while (cp != code + codesize) {
        char op = *cp++;
        int i;
        *dtp++ = (int)labels[op];
        for (i = 0; i < operands[op]; i++)
            *dtp++ = *cp++;
    }

    goto **vPC++;
    label_PUSH:
        *vSP++ = (int)*vPC++;
        goto **vPC++;
    label_ADD:
        lhs = *(--vSP);
        rhs = *(--vSP);
        *vSP++ = lhs + rhs;
        goto **vPC++;
    label_STORE:
        vars[*vPC++] = *(--vSP);
        goto **vPC++;
}
```

5.6. ábra. Direkt vezérelt interpreter részlete



5.7. ábra. Környezetvezérelt interpreter működése

szubrutinra. A környezetvezérelt kód a bájtkódban tárolt operandusokat nem tartalmazza, ezért a környezetvezérelt interpreterek megtartják a direkt vezérelt kódot is, ahol a bájtkód műveleti kódjait a környezetvezérelt kódba mutató címekkel cserélik le, valamint minden virtuális utasítást megvalósító szubrutin karbantartja a (direkt vezérelt kódba hivatkozó) virtuális utasításmutatót is. Ennek a szálvezérlési modellnek az elve az 5.7. ábrán látható.

> **Példa:** A {0, 64, 0, 64, 1, 2, 4} bájtkódból egy környezetvezérelt interpreter Intel x86 architektúrán ins1: call func_PUSH; ins2: call func_PUSH; ins3: call func_ADD; ins4: call func_STORE környezetvezérelt kódot és {&&ins1, 64, &&ins2, 64, &&ins3, &&ins4, 4} direkt vezérelt kódot gyártana.

Az interpreter futásakor a környezetvezérelt kódra kerül a vezérlés, ahol a generált gépi kód hívja az utasításokat megvalósító szubrutinokat, majd a vezérlés mindig oda is tér (jól prediktált módon) vissza.

> **Példa:** Az 5.8. ábra a korábbi direkt vezérelt interpretert fejleszti tovább környezetvezérelt interpreterré. A bájtkódot környezetvezérelt kóddá átalakító kódrészlet Intel x86 architektúrának megfelelő gépi kódot gyárt.

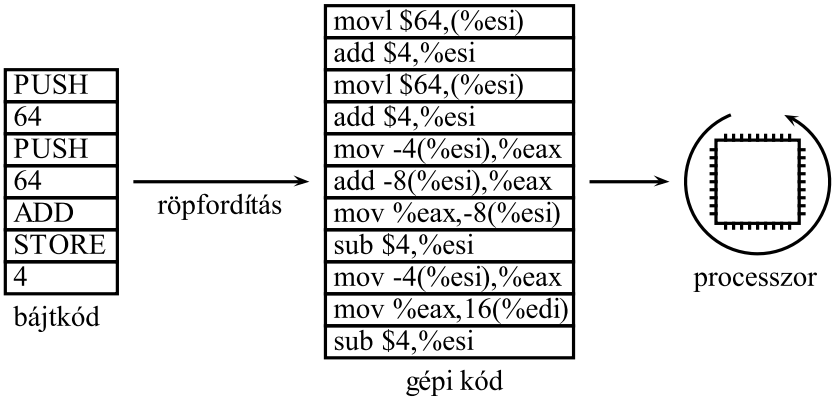
A példából jól látható az ár, amit a környezetvezérelt interpreter hatékonyságáért fizetni kell: a további memóriafogyasztás mellett (a direkt vezérelt kód mellett helyet foglal a környezetvezérelt kód is) az interpreter elveszíti platformfüggetlenségét. Minden platformra, amire az értelmezőt portolni kívánják, külön meg kell valósítani a függvényhívási konvenciónak megfelelő kódgeneráló rutint.

5.4. Röpfordítás

A szálvezérelt interpreterek közül a környezetvezérelt interpreterek által használt megközelítés már közel áll a röpfordításhoz (angolul just-in-time compilation, JIT). Röpfordítás során a program bájtkód reprezentációjából (vagy akár közvetlenül az AST-ből) futtatható gépi kódot generálunk a memóriába, majd az végrehajtjuk. Ennél a megközelítésnél a generált kód már

```
static int *vPC;
static int *vSP;
static int *vars;

static void func_PUSH() {
    vPC++;
    *vSP++ = (int)*vPC++;
}

static void func_ADD() {
    int lhs, rhs;
    vPC++;
    lhs = *(--vSP);
    rhs = *(--vSP);
    *vSP++ = lhs + rhs;
}

static void func_STORE() {
    vPC++;
    vars[*vPC++] = *(--vSP);
}

void execute(char *code, int codesize, int *v) {
    static void *funcs[] = { func_PUSH, func_ADD, func_STORE };
    static int operands[] = { 1, 0, 1 };

    int stack[STACK_SIZE];
    int *direct_thread = (int*)malloc(sizeof(int)*codesize);
    char *context_thread = (char*)malloc_exec(codesize*5);
    char *cp = code;
    int *dtp = direct_thread;
    char *ctp = context_thread;

    while (cp != code + codesize) {
        char op = *cp++;
        int i;

        *dtp++ = (int)ctp;
        for (i = 0; i < operands[op]; i++)
            *dtp++ = *cp++;

        *ctp++ = 0xE8 /*CALL*/;
        *(int *)ctp = (int)funcs[op] - (int)(ctp + 4);
        ctp += sizeof(int);
    }

    vPC = direct_thread;
    vSP = stack;
    vars = v;
    ((void(*)())(*direct_thread))();

    free(direct_thread);
    free(context_thread);
}
```

5.8. ábra. Intel x86 architektúrára tervezett környezetvezérelt interpreter részlete



5.9. ábra. Röpfordító működése

minden szükséges információt tartalmaz, ami a futáshoz kell, nincs szükség direkt vezérelt kódra. (Lásd: 5.9. ábra.)

> **Példa:** Az 5.10. ábra bemutatja, hogy hogyan lehet gépi utasítások bináris kódjai által alkotott mintákból összerakni egy röpfordított függvény törzsét.

A röpfordítás sokban hasonlít a hagyományos fordításra, ám néhány lényeges pontban el is tér tőle. Röpfordítás során a kód belső reprezentációjából (a bájtkódból) nem készül szöveges assembly nyelvű kimenet és azt nem egy külön assembler eszköz fordítja futtatható bináris kóddá, hanem mindez az interpreteren belül történik meg. Emellett a röpfordítók általában nem végeznek, nem végezhetnek túl bonyolult optimalizálásokat a belső reprezentáción kódgenerálás előtt vagy közben, mivel ezek a lépések a futásidőt növelnék. A szkriptnyelvek népszerűsége miatt természetesen az interpreterek és a röpfordítók optimalizálása napjainkban aktív kutatási terület. Ebben a jegyzetben nem lehet célunk a legfrissebb kutatási eredmények ismertetése, de a következő néhány bekezdésben vázlatosan említést teszünk néhány optimalizálási módszerről.

A leggyakrabban alkalmazott módszer az ú.n. szuperutasítások bevezetése. Amennyiben a virtuális utasítások műveleti kódjának ábrázolására használt kódteret nem használjuk ki teljesen (pl.: egy bájtot tartunk fent a műveleti kódoknak, de csak 64 különböző utasításunk van), akkor lehetőségünk van a leggyakrabban használt utasítás és operanduspárosítások, vagy utasításkombinációk számára egy saját műveleti kódot lefoglalni. A szuperutasítások meghatározása és az azokat megvalósító kódrészletek elkészítése általában statikusan történik, az interpreter írásakor. Ebben az esetben a bájtkódot generáló program az AST-ben egyszerű minták alapján felismeri az optimalizálásra alkalmas kódrészletet, és az általános bájtkódsorozat helyett a szuperutasítást generálja. A szuperutasítások gyártása történhet futás közben, dinamikusan gyűjtött statisztikákra épülve is, de ez a megközelítés természetesen bonyolultabb algoritmusokat igényel.

```
void execute(char *code, int codesize, int *vars) {
    int stack[STACK_SIZE];
    char *jit_code = (char*)malloc_exec(codesize*13);
    char *cp = code;
    char *jp = jit_code;

    while (cp != code + codesize) {
        switch (*cp++) {
            case OP_PUSH:
                *jp++ = 0xc7; // movl $imm,(%esi); *vSP=imm
                *jp++ = 0x06;
                *(int *)jp = (int)*cp++;
                jp += 4;
                *jp++ = 0x83; // add $4,%esi; ++vSP
                *jp++ = 0xc6;
                *jp++ = 0x04;
                break;
            case OP_ADD:
                *jp++ = 0x8b; // mov -4(%esi),%eax; temp=*(vSP-1)
                *jp++ = 0x46;
                *jp++ = 0xfc;
                *jp++ = 0x03; // add -8(%esi),%eax; temp+=*(vSP-2)
                *jp++ = 0x46;
                *jp++ = 0xf8;
                *jp++ = 0x89; // mov %eax,-8(%esi); *(vSP-2)=temp
                *jp++ = 0x46;
                *jp++ = 0xf8;
                *jp++ = 0x83; // sub $4,%esi; --vSP
                *jp++ = 0xee;
                *jp++ = 0x04;
                break;
            case OP_STORE:
                *jp++ = 0x8b; // mov -4(%esi),%eax; temp=*(vSP-1)
                *jp++ = 0x46;
                *jp++ = 0xfc;
                *jp++ = 0x89; // mov %eax,imm(%edi); vars[imm]=temp
                *jp++ = 0x87;
                *(int *)jp = ((int)*cp++)*4;
                jp += 4;
                *jp++ = 0x83; // sub $4,%esi; --vSP
                *jp++ = 0xee;
                *jp++ = 0x04;
                break;
        }
    }

    ((void(*)(int*,int*))(jit_code))(stack, vars);

    free(jit_code);
}
```

5.10. ábra. Intel x86 architektúrára tervezett röpfordító részlete

Példa: Tapasztalatok alapján sejtjük, hogy a 0 és 1 konstansok gyakran szerepelnek a programokban, valamint az eggyel való növelés és csökkentés is gyakori művelet. A verem alapú bájtkódunkba ezért bevezethetjük a PUSH_0 utasítást a PUSH 0 utasítás helyett (2 bájt helyett 1 bájtnyi kód), vagy az ADD_1 utasítást a PUSH 1, ADD utasítássorozat kiváltására (2 utasítás és 3 bájt helyet egyetlen bájt és utasítás).

A másik gyakran alkalmazott technika a röpfordítók optimalizálására, hogy egy program futtatása előtt nem fordítódik le a teljes bájtkód gépi kódra. Függvényekkel, szubrutinokkal rendelkező nyelvek esetében alkalmazható az a módszer, hogy az egyes függvényeket csak igény szerint, első meghívásukkor fordítjuk gépi kódra, addig bájtkód formában tároljuk. Így a soha meg nem hívott függvények lefordítására nem pazarlunk erőforrást (sem memóriát, sem futásidőt). Ezt a gondolatot viszi tovább az a megközelítés, amiben az interpreter tud bájtkódvégrehajtó és röpfordító módban is működni, és a röpfordító csak akkor lép működésbe, ha egy függvényt már kellően sokszor (egy előre beállított határértéket meghaladóan) meghívtak. Így tényleg csak arra a kódra (a program ú.n. forró pontjára) fordítunk erőforrásokat, amit valószínűleg megéri kioptimalizálni. Ennél a megoldásnál azonban biztosítani kell, hogy a bájtkódként és a már lefordított, gépi kódként tárolt kódrészletek együtt tudjanak működni. A harmadik technika pedig függvényeken belül és függvényhívásokon keresztül is képes optimalizálni. Az ú.n. nyomkövető röpfordító (angolul tracing JIT) a bájtkódok végrehajtása során naplózza a végrehajtási útvonalakat (az elágazásoknál a választott végrehajtási ágakat), és csak a leggyakrabban futtatott útvonalakat fordítja gépi kódra. A hatékonyságával szemben hártánya ennek a módszernek, hogy a naplózás miatt a memóriaigénye sokszorosa lehet a függvény alapú röpfordítóknak.

5.5. Példamegvalósítás

A következő pár oldalon a fejezetben korábban bemutatott interpreter megvalósítási módszerek példáit fejlesztjük tovább egy teljes, működőképes rendszerré. Előbb definiálunk egy verem alapú bájtkódot (stackvm.StackCode), majd megmutatjuk, hogy hogyan lehet a 3. fejezetben megadott példa nyelvtan AST-jéből ilyen bájtkódot generálni (stackvm.StackGenerator, stackvm.StackSymbolTable). A végrehajtó motort több megközelítéssel is megvalósítottuk: a bájtkód alapú végrehajtásra Java (stackvm.StackMachine, stackvm.StackContext) és C (stackcode.h, stackvm−switch.c) nyelvű példát is láthatunk, míg a token- és direkt vezérelt interpreterek (a Java nyelv korlátai miatt) csak C nyelven készültek el (stackvm−token.c és stackvm−direct.c). A környezetvezérelt interpreter (stackvm−context.c) és a röpfordító (stackvm−jit.c) még speciálisabbak: csak C nyelven készültek el, és a bemutatott formájukban csak Intel Linux platformon futatthatók.

5.5.1. Bájtkódgenerálás és -interpretálás megvalósítása Java nyelven

StackCode.java

```
package stackvm;

public class StackCode {

    private static final String[] mnemonic = { "RET", "PUSH", "ADD",
       "SUB", "MUL", "DIV", "LOAD", "STORE" };
    private static final int[] operands = { 0, 1, 0, 0, 0, 0, 1, 1 };

    public static final byte RET = 0;
    public static final byte PUSH = 1;
    public static final byte ADD = 2;
    public static final byte SUB = 3;
    public static final byte MUL = 4;
    public static final byte DIV = 5;
    public static final byte LOAD = 6;
    public static final byte STORE = 7;

    private StackCode() {}

    public static String toString(byte[] code) {
        StringBuffer buf = new StringBuffer();
        for (int i = 0; i < code.length; i++) {
            buf.append(mnemonic[code[i]]);
            for (int j = 0; j < operands[code[i]]; j++) {
                buf.append("␣");
                buf.append(code[i + 1 + j]);
            }
            i += operands[code[i]];
            buf.append("\n");
        }
        return buf.toString();
    }

}
```

StackGenerator.java

```
package stackvm;

import ast.*;

public class StackGenerator {

    public StackGenerator() {}

    public byte[] generateProgram(Program prog, StackSymbolTable syms) {
        byte[] retCode = { StackCode.RET };

        byte[] code = new byte[0];
        for (Assignment asgmt : prog.getAssignments())
            code = concatenate(code, generateAssignment(asgmt, syms));
```

```
        return concatenate(code, retCode);
    }

    public byte[] generateAssignment(Assignment asgmt, StackSymbolTable
        syms) {
        byte[] exprCode = generateExpression(asgmt.getExpression(),
            syms);
        byte[] assignCode = { StackCode.STORE,
            syms.getVariableIndex(asgmt.getIdentifier().getName()) };
        return concatenate(exprCode, assignCode);
    }

    public byte[] generateExpression(Expression expr, StackSymbolTable
        syms) {
        if (expr instanceof Operator)
            return generateOperator((Operator)expr, syms);
        if (expr instanceof Identifier)
            return generateIdentifier((Identifier)expr, syms);
        if (expr instanceof Constant)
            return generateConstant((Constant)expr, syms);
        throw new RuntimeException("Unexpected␣expression␣" + expr);
    }

    public byte[] generateOperator(Operator op, StackSymbolTable syms) {
        byte[] operandCode =
            concatenate(generateExpression(op.getRightOperand(), syms),
            generateExpression(op.getLeftOperand(), syms));

        byte[] operatorCode = new byte[1];
        switch (op.getOperator()) {
            case Operator.ADD:
                operatorCode[0] = StackCode.ADD;
                break;
            case Operator.SUB:
                operatorCode[0] = StackCode.SUB;
                break;
            case Operator.MUL:
                operatorCode[0] = StackCode.MUL;
                break;
            case Operator.DIV:
                operatorCode[0] = StackCode.DIV;
                break;
            default:
                throw new RuntimeException("Unexpected␣operator␣" + op);
        }

        return concatenate(operandCode, operatorCode);
    }

    public byte[] generateIdentifier(Identifier id, StackSymbolTable
        syms) {
        return new byte[] { StackCode.LOAD,
            syms.getVariableIndex(id.getName()) };
```

```
    }

    public byte[] generateConstant(Constant c, StackSymbolTable syms) {
        return new byte[] { StackCode.PUSH, (byte)c.getValue() };
    }

    private static byte[] concatenate(byte[] code1, byte[] code2) {
        byte[] code = new byte[code1.length + code2.length];
        System.arraycopy(code1, 0, code, 0, code1.length);
        System.arraycopy(code2, 0, code, code1.length, code2.length);
        return code;
    }

}
```

StackSymbolTable.java

```
package stackvm;

import java.util.Map;
import java.util.TreeMap;

public class StackSymbolTable {

    private byte nextVarIndex;
    private Map<String,Byte> indices;

    public StackSymbolTable() {
        nextVarIndex = 0;
        indices = new TreeMap<String,Byte>();
    }

    public byte getVariableIndex(String name) {
        if (!indices.containsKey(name))
            indices.put(name, nextVarIndex++);
        return indices.get(name);
    }

    public String lookupVariableName(byte idx) {
        for (String name : indices.keySet())
            if (indices.get(name) == idx)
                return name;
        return null;
    }

}
```

StackMachine.java

```
package stackvm;

import java.util.Stack;
```

```
public class StackMachine {

    public StackMachine() {}

    public void execute(byte[] code, StackContext ctx) {
        Stack<Integer> stack = new Stack<Integer>();
        int vPC = 0;
        while (true) {
            switch (code[vPC++]) {
                case StackCode.RET:
                    return;
                case StackCode.PUSH:
                    stack.push((int)code[vPC++]);
                    break;
                case StackCode.ADD:
                    stack.push(stack.pop() + stack.pop());
                    break;
                case StackCode.SUB:
                    stack.push(stack.pop() - stack.pop());
                    break;
                case StackCode.MUL:
                    stack.push(stack.pop() * stack.pop());
                    break;
                case StackCode.DIV:
                    stack.push(stack.pop() / stack.pop());
                    break;
                case StackCode.LOAD:
                    stack.push(ctx.getVariable(code[vPC++]));
                    break;
                case StackCode.STORE:
                    ctx.setVariable(code[vPC++], stack.pop());
                    break;
                default:
                    throw new RuntimeException("Unexpected␣StackCode␣" +
                        code[vPC-1] + "␣at␣" + (vPC-1));
            }
        }
    }

}
```

StackContext.java

```
package stackvm;

import java.util.Map;
import java.util.TreeMap;

public class StackContext {

    private Map<Byte,Integer> vars;

    public StackContext() {
```

```
        vars = new TreeMap<Byte,Integer>();
    }

    public int getVariable(byte idx) {
        Integer value = vars.get(idx);
        if (value == null) {
            System.err.println("Reading␣undefined␣variable␣" + idx);
            return 0;
        }
        else {
            return value;
        }
    }

    public void setVariable(byte idx, int value) {
        vars.put(idx, value);
    }

    public String toString(StackSymbolTable syms) {
        StringBuffer buf = new StringBuffer();
        for (byte idx : vars.keySet()) {
            buf.append(String.format("var␣%d", idx));
            if (syms != null) {
                String name = syms.lookupVariableName(idx);
                if (name != null)
                    buf.append(String.format("␣(%s)", name));
            }
            buf.append(String.format("␣=␣%d\n", getVariable(idx)));
        }
        return buf.toString();
    }

    public String toString() {
        return toString(null);
    }

}
```

5.5.2. Bájtkód alapú és szálvezérelt interpreterek megvalósítása C nyelven

stackcode.h

```
#ifndef STACKCODE_H
#define STACKCODE_H


#define OP_RET    0
#define OP_PUSH   1
#define OP_ADD    2
#define OP_SUB    3
```

```
#define OP_MUL    4
#define OP_DIV    5
#define OP_LOAD   6
#define OP_STORE  7

#endif
```

stackvm-switch.c

```
#include <stdlib.h>
#include <stdio.h>
#include "stackcode.h"

#define STACK_SIZE 256

void stackvm_execute_switch(char *code, int *vars) {
    int stack[STACK_SIZE];
    char *vPC = code;
    int *vSP = stack;
    int lhs, rhs;

    while (1) {
        switch (*vPC++) {
            case OP_RET:
                return;
            case OP_PUSH:
                *vSP++ = (int)*vPC++;
                break;
            case OP_ADD:
                lhs = *(--vSP);
                rhs = *(--vSP);
                *vSP++ = lhs + rhs;
                break;
            case OP_SUB:
                lhs = *(--vSP);
                rhs = *(--vSP);
                *vSP++ = lhs - rhs;
                break;
            case OP_MUL:
                lhs = *(--vSP);
                rhs = *(--vSP);
                *vSP++ = lhs * rhs;
                break;
            case OP_DIV:
                lhs = *(--vSP);
                rhs = *(--vSP);
                *vSP++ = lhs / rhs;
                break;
            case OP_LOAD:
                *vSP++ = vars[*vPC++];
                break;
            case OP_STORE:
                vars[*vPC++] = *(--vSP);
```

```
                break;
            default:
                fprintf(stderr, "Unexpected␣StackCode␣%d␣at␣0x%08X\n",
                    *(vPC-1), (int)(vPC-1));
        }
    }
}
```

stackvm-token.c

```
#include <stdlib.h>

#define STACK_SIZE 256

void stackvm_execute_token(char *code, int *vars) {
    static void *labels[] = { &&label_RET, &&label_PUSH, &&label_ADD,
       &&label_SUB, &&label_MUL, &&label_DIV, &&label_LOAD,
       &&label_STORE };

    int stack[STACK_SIZE];
    char *vPC = code;
    int *vSP = stack;
    int lhs, rhs;

    goto *labels[*vPC++];
    label_RET:
        return;
    label_PUSH:
        *vSP++ = (int)*vPC++;
        goto *labels[*vPC++];
    label_ADD:
        lhs = *(--vSP);
        rhs = *(--vSP);
        *vSP++ = lhs + rhs;
        goto *labels[*vPC++];
    label_SUB:
        lhs = *(--vSP);
        rhs = *(--vSP);
        *vSP++ = lhs - rhs;
        goto *labels[*vPC++];
    label_MUL:
        lhs = *(--vSP);
        rhs = *(--vSP);
        *vSP++ = lhs * rhs;
        goto *labels[*vPC++];
    label_DIV:
        lhs = *(--vSP);
        rhs = *(--vSP);
        *vSP++ = lhs / rhs;
        goto *labels[*vPC++];
    label_LOAD:
        *vSP++ = vars[*vPC++];
        goto *labels[*vPC++];
```

```
    label_STORE:
        vars[*vPC++] = *(--vSP);
        goto *labels[*vPC++];
}
```

stackvm-direct.c

```
#include <stdlib.h>

#define STACK_SIZE 256

void stackvm_execute_direct(char *code, int codesize, int *vars) {
    static void *labels[] = { &&label_RET, &&label_PUSH, &&label_ADD,
       &&label_SUB, &&label_MUL, &&label_DIV, &&label_LOAD,
       &&label_STORE };
    static int operands[] = { 0, 1, 0, 0, 0, 0, 1, 1 };

    int stack[STACK_SIZE];
    int *direct_thread = (int*)malloc(sizeof(int)*codesize);
    char *cp = code;
    int *dtp = direct_thread;
    int *vPC = direct_thread;
    int *vSP = stack;
    int lhs, rhs;

    while (cp != code + codesize) {
        char op = *cp++;
        int i;
        *dtp++ = (int)labels[op];
        for (i = 0; i < operands[op]; i++)
            *dtp++ = *cp++;
    }

    goto **vPC++;
    label_RET:
        free(direct_thread);
        return;
    label_PUSH:
        *vSP++ = (int)*vPC++;
        goto **vPC++;
    label_ADD:
        lhs = *(--vSP);
        rhs = *(--vSP);
        *vSP++ = lhs + rhs;
        goto **vPC++;
    label_SUB:
        lhs = *(--vSP);
        rhs = *(--vSP);
        *vSP++ = lhs - rhs;
        goto **vPC++;
    label_MUL:
        lhs = *(--vSP);
        rhs = *(--vSP);
```

```
        *vSP++ = lhs * rhs;
        goto **vPC++;
    label_DIV:
        lhs = *(--vSP);
        rhs = *(--vSP);
        *vSP++ = lhs / rhs;
        goto **vPC++;
    label_LOAD:
        *vSP++ = vars[*vPC++];
        goto **vPC++;
    label_STORE:
        vars[*vPC++] = *(--vSP);
        goto **vPC++;
}
```

stackvm-context.c

```
#include <stdlib.h>
#include <sys/mman.h>
#include "stackcode.h"

#define STACK_SIZE 256

static int *vPC;
static int *vSP;
static int *vars;

static void func_RET() {
    return;
}

static void func_PUSH() {
    vPC++;
    *vSP++ = (int)*vPC++;
}

static void func_ADD() {
    int lhs, rhs;
    vPC++;
    lhs = *(--vSP);
    rhs = *(--vSP);
    *vSP++ = lhs + rhs;
}

static void func_SUB() {
    int lhs, rhs;
    vPC++;
    lhs = *(--vSP);
    rhs = *(--vSP);
    *vSP++ = lhs - rhs;
}

static void func_MUL() {
```

```
    int lhs, rhs;
    vPC++;
    lhs = *(--vSP);
    rhs = *(--vSP);
    *vSP++ = lhs * rhs;
}

static void func_DIV() {
    int lhs, rhs;
    vPC++;
    lhs = *(--vSP);
    rhs = *(--vSP);
    *vSP++ = lhs / rhs;
}

static void func_LOAD() {
    vPC++;
    *vSP++ = vars[*vPC++];
}

static void func_STORE() {
    vPC++;
    vars[*vPC++] = *(--vSP);
}

static void *malloc_exec(size_t size)
{
    void *ptr = malloc(size);
    if (ptr)
        mprotect(ptr, size, PROT_READ | PROT_WRITE | PROT_EXEC);
    return ptr;
}

void stackvm_execute_context(char *code, int codesize, int *v) {
    static void *funcs[] = { func_RET, func_PUSH, func_ADD, func_SUB,
       func_MUL, func_DIV, func_LOAD, func_STORE };
    static int operands[] = { 0, 1, 0, 0, 0, 0, 1, 1 };

    int stack[STACK_SIZE];
    int *direct_thread = (int*)malloc(sizeof(int)*codesize);
    char *context_thread = (char*)malloc_exec(codesize*5);
    char *cp = code;
    int *dtp = direct_thread;
    char *ctp = context_thread;

    while (cp != code + codesize) {
        char op = *cp++;
        int i;

        *dtp++ = (int)ctp;
        for (i = 0; i < operands[op]; i++)
            *dtp++ = *cp++;
```

```
        *ctp++ = (op == OP_RET) ? 0xE9 /*JMP*/ : 0xE8 /*CALL*/;
        *(int *)ctp = (int)funcs[op] - (int)(ctp + 4);
        ctp += sizeof(int);
    }

    vPC = direct_thread;
    vSP = stack;
    vars = v;
    ((void(*)())(*direct_thread))();

    free(direct_thread);
    free(context_thread);
}
```

5.5.3. Röpfordító megvalósítása C nyelven Intel Linux platformra

stackvm-jit.c

```
#include <stdlib.h>
#include <sys/mman.h>
#include "stackcode.h"

#define STACK_SIZE 256

static void *malloc_exec(size_t size)
{
    void *ptr = malloc(size);
    if (ptr)
        mprotect(ptr, size, PROT_READ | PROT_WRITE | PROT_EXEC);
    return ptr;
}

void stackvm_execute_jit(char *code, int codesize, int *vars) {
    int stack[STACK_SIZE];
    char *jit_code = (char*)malloc_exec(codesize*13);
    char *cp = code;
    char *jp = jit_code;

    *jp++ = 0x55; // push %ebp
    *jp++ = 0x89; // mov %esp,%ebp
    *jp++ = 0xe5;
    *jp++ = 0x56; // push %esi
    *jp++ = 0x57; // push %edi
    *jp++ = 0x52; // push %edx
    *jp++ = 0x8b; // mov 8(%ebp),%esi; %esi=vSP
    *jp++ = 0x75;
    *jp++ = 0x08;
    *jp++ = 0x8b; // mov 12(%ebp),%edi; %edi=vars
    *jp++ = 0x7d;
    *jp++ = 0x0c;

    while (cp != code + codesize) {
```

```
        switch (*cp++) {
            case OP_RET:
                *jp++ = 0x5a; // pop %edx
                *jp++ = 0x5f; // pop %edi
                *jp++ = 0x5e; // pop %esi
                *jp++ = 0x5d; // pop %ebp
                *jp++ = 0xc3; // ret
                break;
            case OP_PUSH:
                *jp++ = 0xc7; // movl $imm,(%esi); *vSP=imm
                *jp++ = 0x06;
                *(int *)jp = (int)*cp++;
                jp += 4;
                *jp++ = 0x83; // add $4, %esi; ++vSP
                *jp++ = 0xc6;
                *jp++ = 0x04;
                break;
            case OP_ADD:
                *jp++ = 0x8b; // mov -4(%esi),%eax; temp=*(vSP-1)
                *jp++ = 0x46;
                *jp++ = 0xfc;
                *jp++ = 0x03; // add -8(%esi),%eax; temp+=*(vSP-2)
                *jp++ = 0x46;
                *jp++ = 0xf8;
                *jp++ = 0x89; // mov %eax,-8(%esi); *(vSP-2)=temp
                *jp++ = 0x46;
                *jp++ = 0xf8;
                *jp++ = 0x83; // sub $4,%esi; --vSP
                *jp++ = 0xee;
                *jp++ = 0x04;
                break;
            case OP_SUB:
                *jp++ = 0x8b; // mov -4(%esi),%eax; temp=*(vSP-1)
                *jp++ = 0x46;
                *jp++ = 0xfc;
                *jp++ = 0x2b; // sub -8(%esi),%eax; temp-=*(vSP-2)
                *jp++ = 0x46;
                *jp++ = 0xf8;
                *jp++ = 0x89; // mov %eax,-8(%esi); *(vSP-2)=temp
                *jp++ = 0x46;
                *jp++ = 0xf8;
                *jp++ = 0x83; // sub $4,%esi; --vSP
                *jp++ = 0xee;
                *jp++ = 0x04;
                break;
            case OP_MUL:
                *jp++ = 0x8b; // mov -4(%esi),%eax; temp=*(vSP-1)
                *jp++ = 0x46;
                *jp++ = 0xfc;
                *jp++ = 0x0f; // imul -8(%esi),%eax; temp*=*(vSP-2)
                *jp++ = 0xaf;
                *jp++ = 0x46;
                *jp++ = 0xf8;
```

```
                *jp++ = 0x89; // mov %eax,-8(%esi); *(vSP-2)=temp
                *jp++ = 0x46;
                *jp++ = 0xf8;
                *jp++ = 0x83; // sub $4,%esi; --vSP
                *jp++ = 0xee;
                *jp++ = 0x04;
                break;
            case OP_DIV:
                *jp++ = 0x8b; // mov -4(%esi),%eax; temp=*(vSP-1)
                *jp++ = 0x46;
                *jp++ = 0xfc;
                *jp++ = 0x99; // cltd; temp/=*(vSP-2)
                *jp++ = 0xf7; // idivl -8(%esi)
                *jp++ = 0x7e;
                *jp++ = 0xf8;
                *jp++ = 0x89; // mov %eax,-8(%esi); *(vSP-2)=temp
                *jp++ = 0x46;
                *jp++ = 0xf8;
                *jp++ = 0x83; // sub $4,%esi ; --vSP
                *jp++ = 0xee;
                *jp++ = 0x04;
                break;
            case OP_LOAD:
                *jp++ = 0x8b; // mov imm(%edi),%eax; temp=vars[imm]
                *jp++ = 0x87;
                *(int *)jp = ((int)*cp++)*4;
                jp += 4;
                *jp++ = 0x89; // mov %eax,(%esi); *vSP=temp
                *jp++ = 0x06;
                *jp++ = 0x83; // add $4,%esi; ++vSP
                *jp++ = 0xc6;
                *jp++ = 0x04;
                break;
            case OP_STORE:
                *jp++ = 0x8b; // mov -4(%esi),%eax; temp=*(vSP-1)
                *jp++ = 0x46;
                *jp++ = 0xfc;
                *jp++ = 0x89; // mov %eax,imm(%edi); vars[imm]=temp
                *jp++ = 0x87;
                *(int *)jp = ((int)*cp++)*4;
                jp += 4;
                *jp++ = 0x83; // sub $4,%esi; --vSP
                *jp++ = 0xee;
                *jp++ = 0x04;
                break;
        }
    }

    ((void(*)(int*,int*))(jit_code))(stack, vars);

    free(jit_code);
}
```

5.6. Összegzés

Ebben a fejezetben áttekintettük az interpretált nyelvek értelmezőiben leggyakrabban használt technikákat: a fabejáráson alapuló kiértékelést, a bájtkód interpretert, a token-, a direkt és a környezetvezérelt interpretereket, valamint a röpfordítást. A módszerek tárgyalása során egyszerre haladtunk a kevésbé hatékonytól a hatékony felé, és a könnyen megvalósíthatótól a bonyolultig. A fejezetet egy Java és C nyelveken írt rendszer teljes forráskódjával zártuk, amely példát mutat az összes érintett megközelítés megvalósítására.

5.7. Feladatok

1. Fejlesszük tovább a fejezetben használt bájtkódot úgy, hogy ne csak 8, hanem akár 32 bites egész konstansok is ábrázolhatók legyenek utasítások közvetlen operandusaként (pl.: PUSH 1024). Fejlesszük tovább ennek megfelelően a virtuális gépeket és a bájtkódokat generáló metódusokat.

2. Bővítsük ki a verem alapú bájtkód utasításkészletét olyan, valószínűleg gyakran előforduló műveletekkel, amelyeknél a műveleti kód egyben az operandust is tárolja (pl.: PUSH_0, PUSH_1, PUSH_2, ADD_1, SUB_1, stb.). Bővítsük ennek megfelelően a virtuális gépe(ke)t.

3. Definiáljunk egy regiszter alapú virtuális utasításkészletet, generáljunk az AST-ből ennek megfelelő bájtkódot, majd írjuk meg hozzá interpreter(eke)t.

4. Vezessünk be új típusokat a nyelvbe (pl.: lebegőpontos szám vagy sztring típus), majd vigyük végig a fejlesztés hatását az absztrakt szintaxisfán, a fa alapú kiértékelőn, a bájtkód reprezentáció(ko)n és a vitruális gépeken.

5. Bővítsük a nyelvet logikai kifejezésekkel (pl.: $=$, $\neq$, $>$, $<$) és vezérlési szerkezetekkel (pl.: elágazás, ciklus), majd vigyük végig a fejlesztés hatását az absztrakt szintaxisfán, a faalapú kiértékelőn, a bájtkód reprezentáció(ko)n és a vitruális gépeken.

Irodalomjegyzék

[1] Antlr weboldal. http://www.antlr.org/.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., 1986.

[3] L. Aszalós and T. Herendi. *Fordítóprogramok feladatgyűjtemény*. Debreceni Egyetem, 2010.

[4] Z. Csörnyei. *Fordítóprogramok*. Typotex, 2006.

[5] Z. Fülöp. *Formális nyelvek és szintaktikus elemzésük*. Polygon, 2004.

[6] U. Kastens. Ordered attribute grammars. *Acta Informatica*, 1980.

[7] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[8] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufman, 1997.