Fordítóprogramok
Jegyzet

Tihanyi Norbert
tihanyi.norbert@stud.u-szeged.hu

2008. május 22.

Tételjegyzék

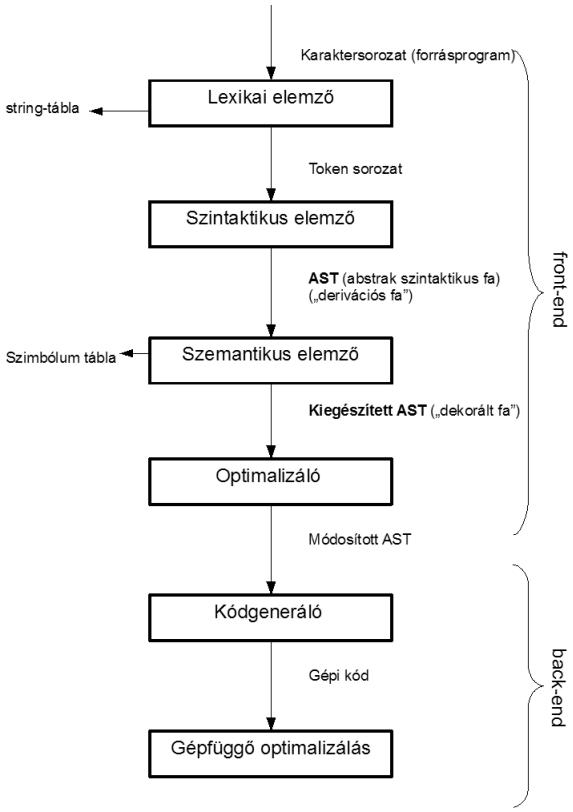
1. Fordítóprogramok szerkezete
2. Attributum nyelvtanok
3. ASE
4. OAG
5. OAG visit sorozat és implementációs séma
6. Kódgenerálás

Forrás

- Előadáson készített jegyzet
- Előadáson kiadott jegyzet
- Csörnyei Zoltán: Fordítóprogramok (TYPOTEX 2006)

Vizsga menete

- Szóbeli vizsga
- egy írásbeli rész és annak megvédése



1. ábra. Fordítóprogramok részei

1. Fordítóprogramok szerkezete

1.1. Fordítóprogramok részei

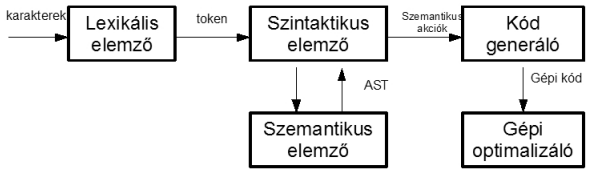
A fordítóprogramok alapvetően két részre bonthatóak.
Az első fázist *front-end* nek nevezzük, amely a géptől független folyamatokat végzi. Itt történik a forráskód feldolgozása, elemzése és optimalizálása.
A második fázis a *back-end* fázis, amely a gépfüggő folyamatok helye, a kód generálása és a gépfüggő optimalizálás.

Megjegyzendő, hogy egyes terminusok szerint a bonyolultabb compilerek (pl.: GNU GCC) három részre osztandóak. A front-end és a back-end közé beillesztenek egy *midle-end* részt, amely új faábrázolásokkal (ASG, GENERIC, GIMPLE, stb.) sokkal összetettebb optimalizálásokat végez.

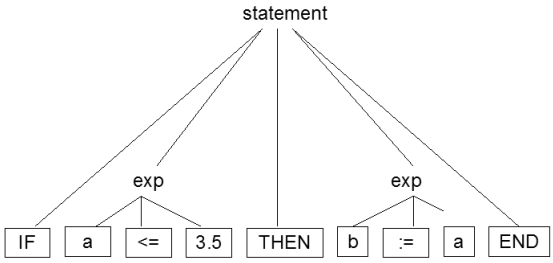
1.2. Fordítás folyamata

1.2.1. Lexikális elemző

Alapja a reguláris nyelvtan.
A lexikális elemző a forráskódot karakterről karakterre feldolgozza és elkülöníti az összetartozó karaktersorozatokat, mint pl. a fenntartott szavak, számok, változók, stb. Tehát a kapott karaktersorozatokat úgynevezett *token*ekké alakítja. Később ezen végzi a fordító a szintakikus elemzést. Emellett *hibajavítás*t is végez. Figyelmeztetést küld a hibás tokenekről. Általában a fordítókat úgy írják meg, hogy hiba esetén ne álljanak le, foglalkozzanak a program többi részével is.



2. ábra. A fordítás folyamatai



3. ábra. Példa lexikális elemzőre

Manapság már a fordítók *a lexikális elemzést a szintaktikus elmzéssel együtt* végzik el. Tulajdonképpen a lexikális elemző a szintaktikus elemző egy eljárása, amit akkor hív meg, amikor kéri a következő tokent.
A lexikális elemző feladata továbbá a *felesleges tokenek kiszűrése*. Ez általában programspecifikus, hiszen pl. a szóköz nem minden nyelv esetén felesleges karakter, van ahol jelentősége van a több szóköz használatának. Ebbe a kategoriába sorolható a *komment* kiszűrése is.

A fordítás legidőigényesebb része a lexikális elemzés.
A folyamat során létrejön a *string tábla*, amely a tokenek neveinek tárolója (pointer minden tokenről). Ehhez jó kereső algoritmus kell.
A string táblára leginkább a szemantikus elemzőnek van szüksége.

1.2.2. Szintaktikus elemző

A lexikális elemző által készített szimbólumsorozatot dolgozza fel. Feladata, hogy *ellenőrizze* a programot *szintaktikai szempontból*, vagyis, hogy az egyes szimbólumok a megfelelő helyen vannak-e, nem hiányzik-e valahonnan egy közülük, stb.
Outputja a program *absztrakt szintaxis fája* (AST) (Megj.: egymenetes fordítók esetén vagy rekurzív eljárásnál nem épül fel)

1.2.3. Szemantikus elemző

A szemantikus elemző a kód *szemantikai helyességé*t hivatott vizsgálni. Ilyen feladat pl. az *azonosító konstans* párosnál az azonosító deklaráltságának ellenőrzése. Továbbá hasonló példa lehet a típusellenőrzés is pl. egy összeadás esetében kapott végeredmény és annak tárolásásra kijelölt változó között.
Emellett ez a rész hivatott a scope-olás ("láthatóság") ellenőrzésére is. Ennek a folyamatnak a

végén egy *dekorált AST* jön létre, s közben felépül a szimbólumtábla is, amely az azonosítók szemantikus információit tartalmazza.

1.2.4. Optimailzáló

Belső ábrázolást használó rész, amely még *gépfüggetlen optimalizálás*okat használ. Mint korábban említettem a nagyobb fordítóprogramok esetében ez a rész is igen robosztus, s vannak terminológiák, amelyek middle-end névvel különítik el a front-endtől és a back-endtől.
Feladatai közé tartozik pl. az elérhetetlen kód vagy a death kód megkeresése.

Elérhetetlen kód. Olyan kódrészlet, ahova a program a futása során sosem ér el.

Death kód (halott kód). Olyan kódrészlet, amely lefutásának eredményét a programban sehol sem használjuk fel. Általában a copy-paste programozás eredménye.

1.2.5. Kódgeneráló

A *tárgykód létrehozás*áért felelős a fordítás folyamán.
Ez a tárgykód lehet számítógéptől és *operációsrendszertől független*, de általában inkább *assembly* vagy *gépi kodú* forma a kimenet.
Az egymenetes fordítók esetében ez a lépés is együtt zajlik a korábbiakkal (*on the fly*).

1.2.6. Gépfüggő optimailzálás

Célja, hogy jobb és hatékonyabb programot állítsunk elő, mint amit egy gyakorlott programozó készíteni tud. Ennek érdekében a fordítóprogram ezen része optimális regiszterhasználatot, konstanspropagációt, a hurkok ciklusváltozóitól független részének megkeresését és a hurkon kívül történő elhelyezését, stb. biztosítja. A fordítás legnehezebb (NP-nehéz), de ugyanakkor leghasznosabb része.
Megj.: GNU GCC esetén magasabb szintű optimalizálással is találkozhatunk.

Konstants propagáció. Azokat a kifejezéseket, amelyek konstansból kiszámíthatóak, ár a fordítás során kiszámolja, így csak egyszer kell kiszámolni, akárhányszor is fogjuk használni. Ez futási időt takarít meg.

1.3. Egymenetes fordítók

Korábbiakból látható, hogy általában a compilerek öt menetben végzik a fordítást. Azonban ez nem minden esetben van így. Léteznek egymenetes fordítók, amelyek csak egyszer mennek végig a kódon és készítenek belőle futtatható programot. Ilyenek pl. az általában oktatási célra készített compilerek. Ezekből pl. a kódoptimalizálás nem is szerepel.
A menetek számát a következő tényezők határozzák meg:

- rendelkezésre álló memória
- compiler mérete és sebessége
- tágyprogram mérete és sebessége
- hibajavítási lehetőségek
- hibafelismerési és hibajavítási stratégiák
- a compiler megírására rendelkezésre álló idő és szellemi kapacitás

Egymenetes fordítót igen nehéz írni, de a többmenetes verzióban a menetek száma csökkenthető ha egyes folyamatokat egyszerre végzünk el, mint pl. a szintaktikus elemzés közben ilexikális elemzés példája a korábbiakban ismertetve.

1.4. Lexikális elemzés

Alapja a *reugláris nyelvtan* vagy másként fogalmazva a *Chomsky 3-as típusú nyelvtan*ok. Ezeket *véges automatá*kkal tudjuk felismerni.

1.4.1. Reguláris nyelvtanok és felismerésük

Az $A \rightarrow aY$ és az $A \rightarrow b$ alakú szabályokból felépülő nyelvtanokat reguláris nyelvtanoknak nevezzük. Korábban tanulmányainkból tudjuk, hogy *minden reguláris kifejezéshez tudunk találni reguláris nyelvtant, ami felismeri*, s ennek megfordítása is igaz.
A reguláris nyelvtanok felépítéséhez *reguláris kifejezések* használandóak:

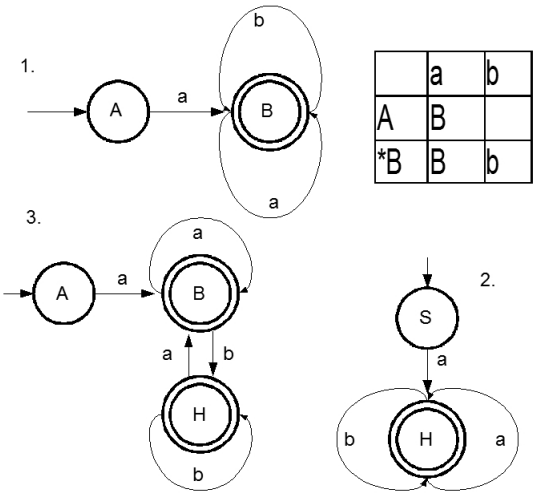
1. $RegExp \rightarrow RegExp\ '|'\ Term$ (—: vagy)
2. $RegExp \rightarrow Term$
3. $Term \rightarrow Term\ Primary$ (konkatenáció)
4. $Term \rightarrow Primary$
5. $Primary \rightarrow Factor\ '*'$ (*: iteráció)
6. $Primary \rightarrow Factor$
7. $Factor \rightarrow '\ (\ '\ RegExp\ '\ )'$
8. $Factor \rightarrow \delta$ (δ: bármilyen karakter)
9. $Primary \rightarrow Factor\ '+'$ (+: legalább egyszer iterál)
10. $Primary \rightarrow Factor\ '?'$ (?: 1 vagy 0)

Determenisztikus véges állapotú automaták. Az $M = (\Sigma, Q, \Delta, q_0, F)$ ötöst *determinisztikus automatá*nak nevezzük, ahol:
Σ: az input ábécé
Q: az állapotok halmaza
$q_0 \in Q$: a kezdőállapot
$F \subseteq Q$: a végállapotok halmaza
Δ: átmeneti reláció

továbbá:
$\Delta : Q \times \Sigma \rightarrow Q$
$\delta\,(A, a) = B$
(q, w): konfiguráció $q \in Q; w \in \Sigma^*$
$q' \in F$ esetén az elfogadási állapot vagy másnéven végállapot: (q', λ)

A formális nyelvek témaköréből ismeretes, hogy két véges automata *ekvivalens*, ha ugyanazt a nyelvet ismerik fel, és *izomorf*, ha ekvivalens, és az állapotok átnevezéssel egymásba átvihetőek. Beszélhetünk még *redukált* véges autómatáról is, ha nincs olyan vele ekvivalens másik véges automata, amely nála kevesebb állapotot tartalmaz. (4. ábra)

Nemdeterminisztikus véges automata. A determinisztikus véges állapotú automatával szemben itt az állapotból egy bemente hatására *több* állapotba juthatunk, továbbá az *üres bemenet* is alkalmazható.



4. ábra. Példák determinisztikus véges automatákra. 1 és 2 izomorfak; 3 az 1-gyel ekvivalens de nem izomorf

1.4.2. Lexikális elemző készítése

A lexikális elemző célja, hogy egy szimbólumot (*token*) a lehető leghosszabb karaktersorozatból építsen fel. Ebből a tokenből a korábban tanultak felhasználásával *reguláris kifejezés* készíthető, amely *véges, nemdeterminisztikus automatával* felismerhető. Ebből a nemdeterminisztikus automatából szintén korábbi ismereteinkre alapozva tudunk *determinisztikus automatát* készíteni, amelyet *minimalizálva* megkapjuk a *lexikális elemző model*jét.
A *klasszikus model*ben a forrásból assembly kód készül, amely már végrehajtható. Erre legjobb példa a C, C++ nyelv.
Léteznek *interpreteres* nyelvek (van valamilyen futtató rendszere), amelyeknél az *eredeti forrás* utasításai kerülnek végrehajtásra (pl.: Prolog) vagy *bytecode* jön létre, és ezt értelmezi (pl.: Java). Ez utóbbi esetében érdmes említést tenni a a *JIT* fordítóról, amely a hagyományos compiler és az interpreter kombinációja. A forrásnak vannak részei amelyekből *native* kódot készít, s az interpretálás során ezt is használja. Segít kihasználni a hardver előnyeit ám ezzel elveszik a gépfüggetlensége pl. a java nyelven írt programoknak.

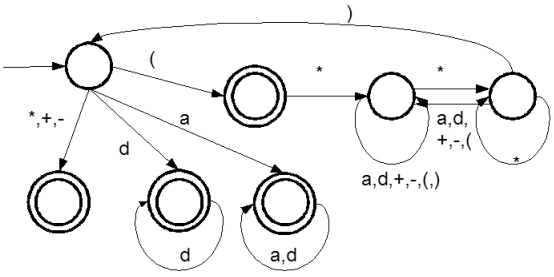
1.4.3. Implementációs problémák

A compiler a lehető legkevesebb esetben fordít tökéletes programot, tehát a műveletek közben képesnek kell lekezelnie a hibákat. A továbbiakban a lexikális elemzés közben felmerülő hibák és megoldásaik kerülnek tárgyalásra.

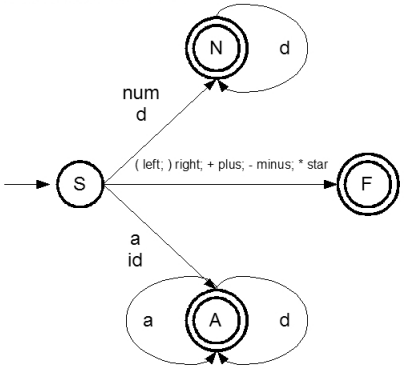
Nagy számű karakterek esetén célszerű úgynevezett *karakterhalmazok* bevezetése. Ilyen halmazok lehetnek pl.: a letter (betű), digit (számjegy), stb. Ha már csak ezeket kell kezelni, lényegesen könnyebbé válik az implementáció.

A *megállási probléma* szorosan összefügg a *kulcsszavak és standard szavak problémájával.*
Kulcsszavak (fenntartott szavak): speciális célra fenntartott szavak, előre definiált jelentéssel.
Standard szavak: előre definiált azonosítók, amelyek jelentése később a programban megváltoztatható.
A megállási probléma feloldására két eszköz használatos:

- *Pufferelési technika:*
  Lényegében egy pufferben helyezzük el a karaktereket, s ha az automata végállapotba kerül,



(,*,a,d,+,-,) szimbólumokat ismer fel. A (* *) közötti rész kommentezésnek számít.



Szemantikus akciókkal kiegészített ábra

5. ábra. Összehasonlítás szemantikus akciókra

akkor innen vesszük a karaktereket. Amennyiben folytatható az automata működése, akkor folytatjuk.

- *Kivesszük a végállapotokat:*
  Ez az általánosan elterjedt módszer. Így mindig a leghosszabb felismerhető tokent ismeri fel.

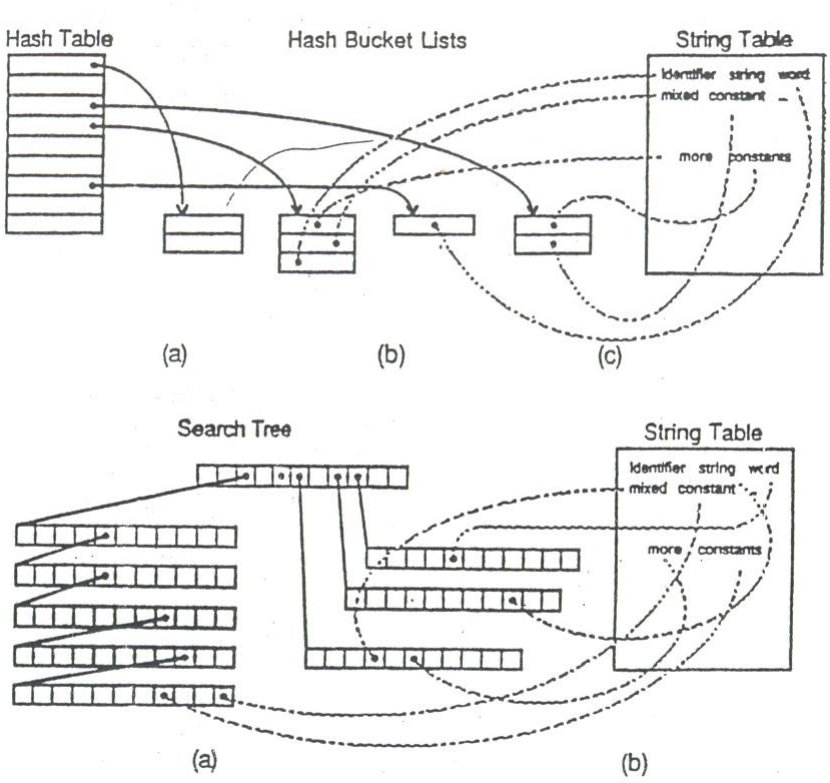
És itt kapcsolódik a kulcsszavak, standard szavak problémája, hiszen ezek felismerése is kritikus a program helyes fordítása szempontjából, ugyanakkor előfordulhat, hogy a programozó által adott változók is úgy kezdődnek, mint egy fenntartott szó a programozási nyelvben.
Megoldást nyújthat a végállapotok kivétele, hiszen így csak akkor kerül pl.: egy IF felismerésre, ha pusztán ebből a két karakterből áll.

A *szóközök és kommentek* kezelése is fontos kérdés a fordítás során. Általában ezek a forráskód azon részei, amelyekre az elemzőnek nincs szüksége, ezért elhagyhatóak, de születhet olyan programozási nyelv amelyben a szóközök száma jelentéssel bír. Érdemes ezek eldobása előtt megfontolni szükségességüket.

Amennyien az outputra adott tokeneket kiegészítjük szemantikus akciókkal, könnyebbé tesszük a szemantikai elemzést.

1.4.4. Keresés stringtáblában

Korábbiakban már láthattuk, hogy a lexikális elemző működése során string táblát hoz létre. Ebben a táblában kerülnek tárolásra az azonosítók és a hozzájuk tartozó pointerek. Belátható, hogy már néhány száz soros program esetében is igen nagy méretű tábla jöhet létre. Ez felvet egy



6. ábra. String tábla tárolási módjai

alapvető problémát, miszerint *a lineáris keresés* ideje nagyon nagy lehet.
Megoldást a *hash kódolás* jelenthet. Ennek alapelve, hogy megadunk egy hash függvényt (pl.: *(stringekben levő karakterek értékei) mod n.*), amely minden stringhez hozzárendel egy-egy számot. Ezzel nagyobb csoportokra oszlik a táblázat amely csoportokat egy-egy hash függvény azonosít. Így keresni már csak egy ilyen csoportban kell. Másik megoldást a *fa struktúra alkalmazása* lehet. Vagyis nem teszünk mást, mint az egyes azonosítókat fastruktúrában tároljuk.

1.5. Szintaktikus elemzés

Egy programnyelv teljes definiálása magában foglalja a nyelv szintaxisát és szemantikáját. Azonban a gyakorlat azt mutatja, hogy a programnyelvek szintaxisát nem lehet leírni csak környezetfüggetlen nyelvekkel. Szükségesek környezetfüggő, kétszintű vagy *attribútum* nyelvtanok is.
Bevett szokás, hogy a szintaxis megadása két részre oszltatik. Nagyobbik fele környezetfüggetlen grammatikával kerül megadásra, kisebbik része pedig környezetfüggő vagy attribútum nyelvtannal. Az első részhez tartozik a program struktúrájának, utasításkészletének felépítése, a második részhez pedig olyan megkötések, mint a típusazonosság vagy a szimbólumok érvényességi tartománya.
A környezetfüggetlen grammatikákkal leírható tulajdonságok vizsgálatát hívjuk *szintaktikus elemzésnek.*

1.5.1. Környezetfüggetlen nyelvtanok és felismerésük

A csak $A \rightarrow \alpha$ alakú szabályokat tartalmazó nyelveket *környezetfüggetlen* vagy másnéven *Chomsky 2-es típusú* nyelvtanoknak nevezzük.
Szintaktikai elemzés keretében azonban a CF nyelvtanokra néhány megszorítást kell tennünk:

1. ciklusmentes kell, hogy legyen, vagyis nem tartalmazhat $A \Rightarrow^* A$ levezetést (*balrekurzió)*
2. *redukált* kell, hogy legyen, azaz nem tartalmazhat felesleges nemterminális szimbólumokat

A környezetfüggetlen nyelvtanok felismerésének eszköze a *veremautomata*, amely annyiban különbözik a korábban megismert automatától, hogy egy verem segítségével végzi a számításait.

1.5.2. Szintaktikus elemzés folyamata

Egy programot akkor nevezünk helyesnek, ha a szintaxisfában találunk egy a gyökértől levélig vezető utat. Ehhez csak balról jobbra haladó elemzéseket használunk.
A szintaxisfa felépítésére több módszer is létezik.
Az egyik, amikor az S szimbólumból kiindulva építjük fel a szintaxisfát. Ezt *felülről-lefelé (top-down)* elemzésnek nevezzük.
Ha a szintaxisfa építését a levelektől indulva az S szimbólum felé haladva végezzük, *alulról-felfelé (bottom-up)* elemzést alkalmazunk.
Az 1980-as évektől vált nyílvánvalóvá, hogy az LR és az LL elemzési módszerek igen hatékonyak. Azóta ezek használatosak a fordítóprogramokbna.
Ez az elemzési módszer azonban nagyon munkaigényes, hiszen $FIRST$ illetve $FOLLOW$ halmazokat kell számolnunk. Ennek támogatására készültek el az elemző generátorok, amelyek a halmazok számolásának terhét leveszik a vállunkról.

LL(1) nyelvtanok. Alapötlet: a levezetés következő lépéséhez használjuk fel a szöveg következő terminálisát.

FIRST$_1(\alpha)$. az α mondatformából levezethető mondatok első terminálisainak halmaza.

FOLLOW$_1(\alpha)$. azok a terminálisok, amelyekhez van a nyelvnek olyan mondata, hogy a szintaxisfában közvetlenül α után szerepelnek.

LL(k) nyelvtanok. Az *LL(k)* nyelvtanok alaptulajdonsága, hogy ha az $S \Rightarrow^* wx$ legjobboldalibb levezetés építésekor eljutunk az $S \Rightarrow^* wA\beta$ mondatformáig, és az $A\beta \Rightarrow^* x$-t szeretnénk elérni, akkor az A-ra alkalmazható $A \to \alpha$ egyértelműen meghatározott, ha ismert x első k db szimbóluma.

$$FIRST_k(\alpha) = \{x | \alpha \Rightarrow^* x\beta \; es \; |x| = k\} \cup \{x | \alpha \Rightarrow^* x \; es \; |x| < k\}$$

$$FOLLOW_k\{\alpha\} = \{x | S \Rightarrow^* \alpha\beta\gamma \; es \; x \in FIRST_k(\gamma)\}$$

2. Szemantikus elemzés - Attributum nyelvtanok

Mint korábban említettem a szemantikus elemző feladata az input helyes szemantikájának ellenőrzése. Ezek a tulajdonságok csak jelentős sebességcsökkenés árán írhatóak le környezetfüggetlen nyelvtanokkal, ezért ezt az irányt nem preferálják manapság. Ehelyett az egyes szemantikus tulajdonságok vizsgálatára *önálló programoka* írnak.
A szemantikus elemző általában a következő tulajdonságokat vizsgálja:

- *változók deklarációja* és a *változók hatásköre, láthatósága*
- *változók többszörös deklarációja*, a *deklaráció hiánya*
- operátorok és operandusaik közötti *típuskompatibilitás*
- eljárások, tömbök formális és aktuális paraméterei közötti *kompatibilitás*
- *túlterhelések egyértelműsége*

Ha egy környezetfüggetlen nyelvtant kiegészítjük *jobb oldali akciószimbólumok*kal, akkor *fordítási grammatikát* kapunk.

2.1. Attribútum nyelvtanok (AG)

Az attribútum nyelvtanok is ilyen fordítási grammatikák.

Kifejlesztése *Donald* ***Knut*** nevéhez köthető. 1968-ban látta meg a napvilágot a cikke, amelyben a tudományos társadalom elé tárta. *Dekleratív* jellegű, akárcsak a Prolog programozási nyelv.

Dekleratív nyelv. A programozóra bízzuk a végrehajtást. Létezik egy végrehajtási model, amelyen dolgozhat. Ilyen lehet pl. klózok halmaza.

Imperatív nyelv. A ilyen fajta nyelven írt programoknál a számítási modelt is meg kell adni. Pl. C, C++, JAVA, PASCAL

Előnye, hogy hatékony elemző generálható a specifikációjából:
$AG = (G, A, Val, SF, SC)$
ahol $G = (N, T, S, P)$ egy redukált környezetfüggetlen nyelvtan
$V = N \cup T$
$p \in P : X_0 \to X_1, \ldots, X_{n_p}$

Redukált. a nyelvtan minden nemterminálisa elérhető legyen S-ből és minden nemterminálisból levezethető legyen egy csupa terminálisból álló sorozat.

2.1.1. További jelölések

- AN: attribútum nevek halmaza
  - ANI: örökölt attribútum nevek halmaza
  - ANS: szintetizált attribútum nevek halmaza
  - $AN = ANI \cup ANS$ és $ANI \cap ANS = \oslash$
- A: attribútumok halmaza
  - AI: örökölt attribútumok
  - AS: szintetizált attribútumok
  - $A = AI \cup AS$ és $AI \cap AS = \oslash$

- A_x: az X szimbólumhoz rendelt attribútumok halmaza
- $A = \cup_{x \in V} A_X$ attribútumok halmaza
- $X.a$ az X szimbólum a attribútuma

Az $X.a$ attribútumát *szintetizáltnak* nevezzük, ha értékét egy szemantikus függvény abban az esetben határozza meg, amikor az X szimbólum egy helyettesítési szabály *bal* oldalán található.
Az $X.a$ attribútum *örökölt*, ha értékét egy szemantikus függvény akkor határozza meg, amikor az X szimbólum egy helyettesítési szabály *jobb* oldalán áll.
Tehát az információt egy szintaxisfában a *szintetizált attribútum alulról-felfelé*, egy *örökölt attribútum felülről-lefelé* és *egy helyettesítési szabály jobb oldalát alkotó pontokon továbbítja.*
Az S kezdőszimbólumhoz nem tartozik örökölt attribútum, és a *terminális szimbólumokhoz* nem tartoznak szintetizált attribútumok, ugyanakkor lehetnek *kitüntetett szintetizált attribútumaik*, mint pl. egy konstans terminális szimbólum esetén a konstans értéke. Ezen attribútumok értékét *konstans értékű szemantikus függvények* határozzák meg, amely értéket egy külső eljárás, mondjuk a lexikális elemző adja át.

2.1.2. Attribútum előfordulások

Jelölje A_p a p $(p : X_0 \to X_1, \ldots, X_{n_p})$ attribútum előfordulásainak halmazát, vagyis

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

Ekkor *definiáló attribútum előfordulásokon* a következőt értjük:

$$AF_p = \{X_i.a | \, (i = 0 \; es \; X_i.a \in AS_{X_i}) \; \; vagy \; \; (i > 0 \; es \; X_i.a \in AI_{X_i})\}$$

Vagyis minden olyan attribútum, amely *baloldali szimbólum szintetizált* attribútuma, vagy *jobboldali szimbólum örökölt* attribútuma.

Az alkalmazott attribútumok pedig a

$$AC_p = A_p - AF_p$$

halmazba tartozó attribútumok, vagyis a *baloldali szimbólum örökölt* attribútumai, vagy a *jobboldali szimbólum szintetizált* attribútumai.

2.1.3. Szemantikus függvények

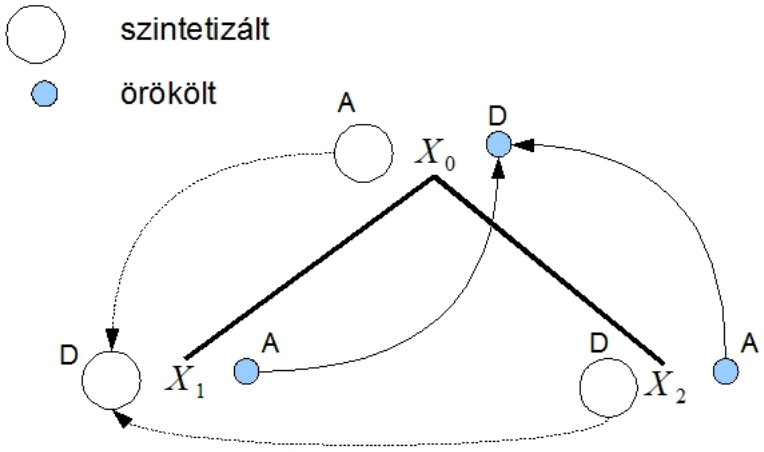
A szemantikus függvény segítségével értékeket rendelünk a csomópontokhoz a derivációs fában. Legyen

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

Ezt az $X_i.a$-t definiáló attribútum előfordulásokhoz rendeljük. Fontos azonban, hogy *minden* definiáló attribútum előfordulásra *pontosan egy* szemantikus függvényt kell írni. A függvény argomentumai csak olyan attribútumelőfordulások lehetnek, amelyek már korábban *kaptak* értéket.

Az attribútum nyelvtanoknál nagyon fontos kérdés, hogy hogyan biztosítható minden attribútum kiszámolása. A megoldást a *normál formában* lévő AG-k adják, vagyis a szemantikus függvények argomentumai csak *alkalmazott* attribtútum előfordulások lehetnek. Tehát jobb oldalon csak *örökölt*, bal oldalon csak *szintetizált* attribútumok szerepelnek.

Említést kell még tennünk a *Val* halmazról, amelybe az attribútumok lehetséges értékei kerülnek, továbbá a *szemantikus feltételekről (SC)*, amelyek logikai függvények. Argomentumai attribútum előfordulási szabályokhoz rendeltek. Felismerésüket megkönnyíti, hogy nincs bal oldaluk. Tökéletesen alkalmasak a hibaüzenetek kiírására.



7. ábra. Derivációs fa kiegészítve függőséggel

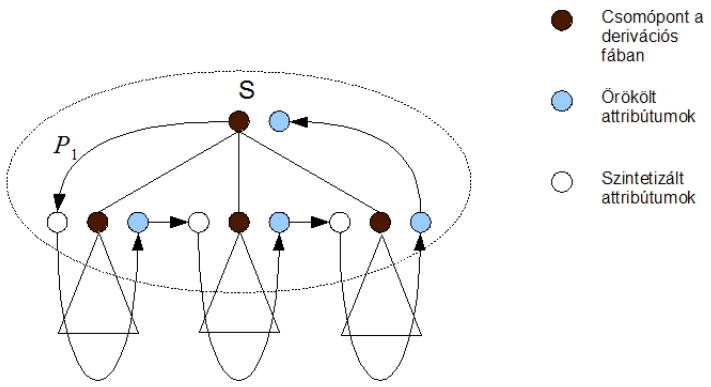
Attribútum példány. Az attribútum előfordulásnál létrejön egy példány a derivációs fában.

2.1.4. Attribútum nyelvtanok kiértékelése

Látható, hogy a jól definiált attribútum nyelvtanokban az attribútum értékei meghatározhatóak. E folyamat során *attribútum példányok* jönnek létre a derivációs fában. A problémát az okozza, hogy az *attribútum előfordulások* függőségi gráfot hoznak létre. E gráf körmentességének eldöntéséhez azonban *exponenciális* idejű algoritmus kell.

Cirkuláris AG. Van olyan függőségi gráfja, ami kört tartalmaz.

Tehát a kiértékeléshez szükséges további megszorításokat tenni. Egyik módszer, hogy *nem-cirkuláris* részosztályokat definiálunk.
Ekkor az alábbi nemdeterminisztikus algoritmussal meghatározható egy tetszőleges mondat szintaxisfájához tartozó attribútumok értékei:
Minden egyes attribútumérték kiszámításához rendeljünk hozzá egy *folyamatot*. Egy folyamat működésbe lép, ha a hozzátartozó attribútum kiszámításához szükséges attribútumérték már ismert. A folyamat befejeztével kiszámított attribútumértékek újabb folyamatokat indítanak el. Ez a folyamat a *kitüntetett szintetizált attribútumok* felhasználásával kezdődik, hiszen azok értékei már ismertek. A jóldefiniáltság biztosítja azt, hogy minden algoritmus értéke meghatározaható, és az algoritmus terminál.
Fontos kérdés azonban, hogy az egyes lépésekben melyik örökölt és melyik szintetizált attribútum értéke határozható meg. Erre számos stratégia létezik.



8. ábra. L-R kiértékelő stratégia

2.1.5. Fabejárási kiértékelési stratégia

1. `Proc` X_0`(N: node);`
2. `for i:= 1 to` n_p `do`
3. `begin eval (`AI_{X_i}`);`
4. X_i `end;`
5. `eval(`AS_{X_0}`);`
6. `end`

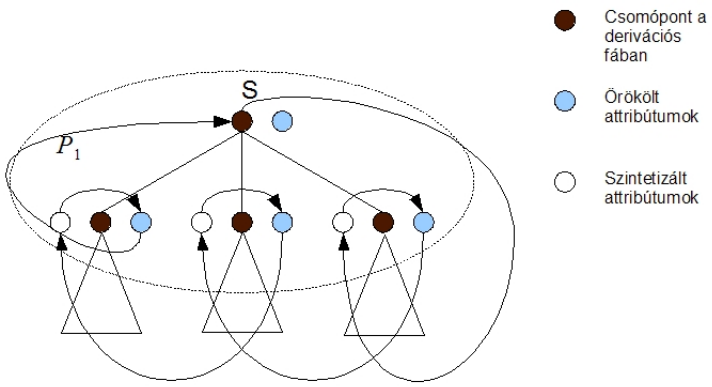
Említést kell tennünk a menetvezérelt kiértékelés fajtáiról, hiszen a legtöbb stratégia alapja.
Az attribútumok értékeinek meghatározása során a derivációs fát bejárhatjuk *postorder* bejárással. Ekkor *L-R kiértékelőről* beszélünk. Ez a stratégia feltételezi azt, hogy a szemantikus információ *balról jobbra* terjed.
Ha a postorder bejárást megváltoztatjuk, hogy a leveleket *jobbról balra* haladva járjuk be, akkor *R-L kiértékelő*t kapunk.
Azokat a többmenetes attribútum kiértékelőket, amelyek az attribútumokat páratlan menetekben *L-R*, páros menetekben *R-L* stratégiával értékeli ki, *alternáló kiértékelőknek* vagy röviden *ASE kiértékelőknek* nevezzük.

2.1.6. Egyszerű többmenetes kiértékelő

Input: attribútumos derivációs fa

- $\langle d_1, \ldots, d_n \rangle$ menetirány-vezérlő sorozat
- $\langle L, L, R, L, \ldots \rangle$ (több menetben járjuk be a fát, de nem biztos, hogy mindig balról jobbra. Ez a sorozat mutatja meg, hogy melyik menetben melyik stratégiát válaszzam.)
- $\langle A_1, \ldots, A_n \rangle$ az A halmaz partícionálása. A_i azon attribútumokat tartalmazza, amelyeket az i. menetben akarunk és tudunk kiértékelni

Output: a kiszámított attribútumos fa



9. ábra. R-L kiértékelő stratégia

Az algoritmus.

```
 1. Type node = ...
 2. Procedure Visit(N:node, i:Integer;
 3. Var start, limit, step:Integer;
 4. Begin
 5.    If di = L Then
 6.       Begin
 7.         start:=1;
 8.         step:=1;
 9.         limit:=np;
10.       End
11.    ELse
12.       Begin
13.         start:=np;
14.         step:=-1;
15.         limit:=1;
16.       End;
17.    For k:=start To Limit By step Do
18.       For minden Xk.a eleme AIx-re Do
19.         If Xk.a eleme Ai Then
20.           Evaulate(Nk.a);
21.        od
22.        If Xk eleme VN Then
23.           Visit(Nk.i);
24.    od;
25.    For minden Xk.b eleme ASX0 Do
26.        If Xk.b eleme Ai Then
27.           Evaulate(N0.b);
28.    od;
29. End {Visit}
30.
31. Begin
32.    For i:= 1 To n Do Visit(root,i);
33. End.
```

2.1.7. Példa attribútum nyelvtanra

A bináris törtszám decimális értékének meghatározása
(Knut 1968)

Attribútumok:

$X.v$: az X-ből levezethető string decimális értéke

$X.l$: az X-ből levezethető string hossza

$X.r$: az X-ből levezethető string legjobboldalibb helyiértéke

$AN = \{v, l, r\}$

$ANI = \{r\}$

$ANS = \{l, v\}$

$A_s = \{S.v\}$

$A_N = \{N.v, N.l, N.r\}$

$A_B = \{B.v, B.r\}$

Szabályok:

1. $S \rightarrow N_1\ '\cdot'\ N_2$

   $S.v := N_1.v + N_2.v$
   $N_1.r := 0$
   $N_2.r := -N_2.l$

2. $N_0 \rightarrow N_1 B$

   $N_0.v := N_1.v + B.v$
   $N_0.l := N_1.l + 1$
   $N_1.r := N_0.r + 1$
   $B.r := N_0.r$

3. $N \rightarrow \lambda$

   $N.v = 0$
   $N.l = 0$

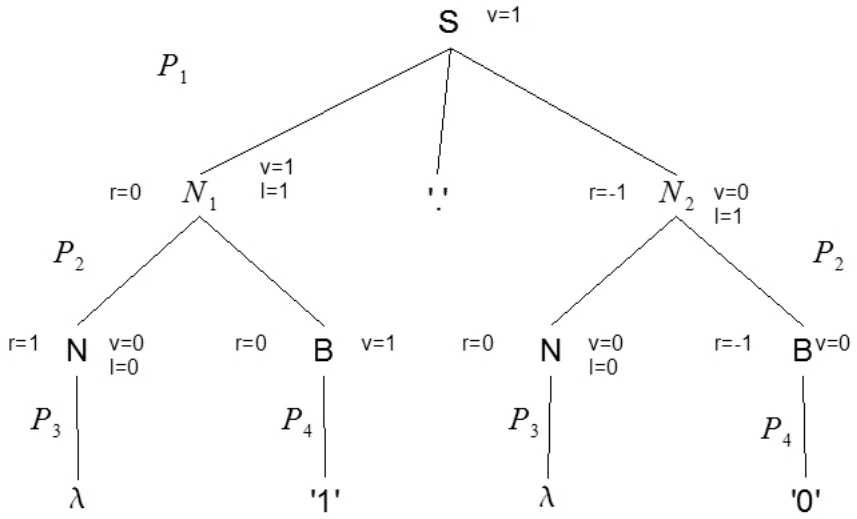
4. $B \rightarrow\ '1'$

   $B.v = 2^{B.r}$

5. $B \rightarrow\ '0'$

   $B.v = 0$

Észrevehető, hogy a balról-jobbra kiértékelésnél több menet szükséges a kiértékeléshez.



10. ábra. Bináris törtszám decimális értékének meghatározása

3. ASE - Alternating Semantic Evaluator

Az ASE nyelvtanhoz tartozó attribútum nyelvtanok specialitása, hogy kiértékelésük során alternáló sorozatot használunk, vagyis a *páratlan* körben *balról jobbra* végezzük az elemzést, még *páros* körben *jobbról balra*. Ezt addig csináljuk, ameddig minden attribútum ki nincs értékelve.
Gyakorlatban előfordulhat, hogy az elemzés során üres meneteket vagyunk kénytelenek végrehajtani, ám az is teljesül, hogy e nyelvcsaládba tartozó attribútum nyelvtanok biztosan kiértékelhetőek.

3.1. ASE algoritmus

Input: egy teszőleges attribútum nyelvtan
Output: annak eldöntése, hogy az attribútum nyelvtan eleme-e az ASE nyelvosztálynak. Továbbá, ha $AG \in ASE$, akkor az algoritmus megadja, hogy az egyes attribútumok *melyik menetben értékelhetők ki.*

```
 1. begin
 2.    m:=0;
 3.    B:=A;
 4.    repeat
 5.         m:=m+1;
 6.         Am:=B;
 7.         repeat
 8.              for minden X.a eleme Am-re (Xi=X) do
 9.                 if létezik(Xj.b, Xi.a), ahol Xj.b nem eleme Ak k<=m
10.                    vagy Xj.b eleme Am, de Xj.b az Xi.a után fordul elő az adott
11.                         kiértékelési menet szerint
12.                 then
13.                     X.a nem eleme Am;
14.         until nem törölhető több attribútum;
15.         B:=B-Am;
16.    until B=üres vagy (Am=Am-1=üres)
17. end;
```

Az algoritmus befejeztével ha:

1. $B = \text{üres} \Rightarrow AG \in ASE$
2. $A_m = A_{m-1} = \text{üres} \Rightarrow AG \notin ASE$

3.2. Példa

Szabályok:

1. $S \rightarrow N_1\ '\cdot'\ N_2$

   $S.v := N_1.v + N_2.v$
   $N_1.r := 0$
   $N_2.r := -N_2.l$

2. $N_0 \rightarrow N_1 B$

   $N_0.v := N_1.v + B.v$
   $N_0.l := N_1.l + 1$
   $N_1.r := N_0.r + 1$
   $B.r := N_0.r$

3. $N \rightarrow \lambda$

   $N.v = 0$
   $N.l = 0$

4. $B \rightarrow\ 'l'$

   $B.v = 2^{B.r}$

5. $B \rightarrow\ '0'$

   $B.v = 0$

Az ASE algoritmus végrehajtása:
A $\surd$-val jelzett attribútumokokat nem tudjuk kiértékelni az adott menetben, maradnak a halmazban.

1. $m = 1$

   $B = (N.l, N.v(\surd), N.r(\surd), S.v(\surd), B.r(\surd), B.v(\surd))$
   $A_1 = (N.l)$

2. $m = 2$

   $B = (N.v, N.r, S.v, B.r, B.v)$
   $A_2 = (N.v, N.r, S.v, B.r, B.v)$

3.3. Példa

Olyan attribútum nyelvtan, amely ugyan nem cirkuláris, de ugyannakor nem is ASE.

Szabályok:

1. $A \rightarrow A\ B$

   $B.b_i := a_i$
   $A.a_i := B.b_s$
   $a_s := A.a_s$

2. $A \rightarrow B\ A$

   $B.b_i := a_i$
   $A.a_i := B.b_s$
   $a_s := A.a_s$

3. $A \rightarrow\ 'x'$

   $a_s = a_i$

4. $B \rightarrow\ 'y'$

   $b_s = b_i$

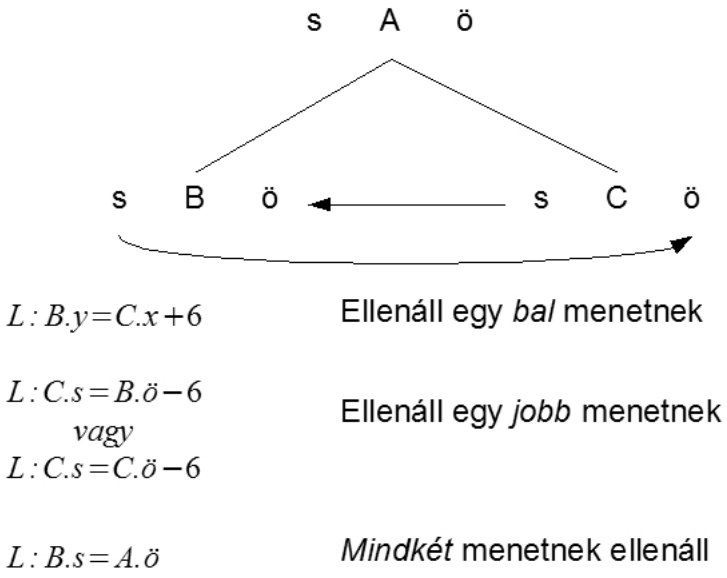
Az ASE algoritmus végrehajtása:
A $\surd$-val jelzett attribútumokokat nem tudjuk kiértékelni az adott menetben, maradnak a halmazban.

1. $m = 1$

   $B = (A.a_i(\surd), A.a_s(\surd), B.b_i(\surd), B.b_s(\surd))$
   $A_1 = ()$

2. $m = 2$

   $B = (A.a_i(\surd), A.a_s(\surd), B.b_i(\surd), B.b_s(\surd))$
   $A_2 = ()$



11. ábra. Hibák a menetek zavartalan lefolyásában

4. OAG - Ordered Atribute Grammars

A rendezett attribútum nyelvosztály egy viszonylag *új módszer* a szemantikus elemzésre. Bonyolult, de ugyanakkor igen *népszerű* is, hiszen igen *hatékony*an oldja meg az elemzést.

4.1. Jellemzői

- Több attribútumnyelvtant is tartalmaz
- *Nincs előre definiált fabejárás.* Eddig azt láttuk az ASE algoritmusoknál, hogy minden egyes csomópontot ugyanannyiszor járunk be, s így olykor üres menetek is sorra kerülhetnek. Ezt a hibát próbálja meg az OAG kiküszöbölni azzal, hogy az *egyes részfákat különböző számban* járja be. Minden egyes szabályhoz *kiértékelési stratégiát* rendelünk, amely *visit sorozatokat* eredményez.
- Az adott attribútumok között végzünk rendezést, osztályba soroljuk őket. Ezzel vezéreljük a bejárási stratégiát.
- Polinomiális algoritmussal eldönthető

4.2. Függőségi halmazok definiálása

4.2.1. Direkt függőségek

Ha addott egy P szabály:

$$P : X_0 \to X_1, \ldots, X_{n_p}$$

akkor

$$DP_P = \{(X_i.a, X_j.b) : \exists SF_P \; hogy \; X_j.b := f\,(\ldots X_i.a \ldots)\}$$

ahol $(X_i.a, X_j.b)$ a P szabály környezetében érvényes attribútum előfordulások (csomópontok a derivációs fában $X_i.a \to X_j.b$ függőséggel).
Tehát van olyan szemantikus függvény amely bal oldalán $X_i.a$, jobb oldalán pedig $X_i.b$ áll.

$$DP = \bigcup_{p \in P} DP_p$$

Példa. 3.3-as példa 1. szabályához tartozó direkt függőségek:
$DP_1 = \{(a_i, B.b_i), (B.b_s, A.a_i), (A.a_s, a_s)\}$

4.2.2. Indukált függőségek (*szabályokra*)

$IDP_P =$

$$DP_P$$
$$\bigcup$$

$$\left\{(X_i.a, X_j.b) : X_i \; p - beli \; szimb \; es \; \exists \; olyan \; Y_j \; q - beli \; szimb, \; h \; X_i = Y_j \; es \; (Y_j.a, Y_j.b) \in IDP_q^+\right\}$$

ahol IDP_q^+ a gráf pozitív lezártja, továbbá $X_i = Y_j$ jelenti azt, hogy ugyanazt a szimbólum előfordul másik szabályban is.
Tehát ha a és b attribútumok között van él, akkor bemásoljuk DP_P-be, s megkapjuk IDP_P-t.
Új éleket (függőségeket) csa egy adott szimbólum attribútumai között hozunk létre.

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

Példa. 3.3-as példa 1. szabályához tartozó indukált függőségek:
$IDP_1 = DP_1 \bigcup \{(a_i, a_s), (b_i, b_s)\}$

4.2.3. Indukált függőségek (*szimbólumokra*)

Az egyes szabályokban az attribútumhoz tartozó függőségeket kivesszük.

$$IDS_X = \{(X.a, X.b) : X_i = X \; es \; (X_i.a, X_i.b) \in IDP_P\}$$

$$IDS = \bigcup_{X \in V} IDS_X$$

Azok a függőségek, amelyek a különböző szabályokból összejöttek.

4.2.4. Partícionálás (*osztályozás*)

Tegyük fel, hogy IDS nem ciklikus, mert ha az lenne, akkor az adott attribútum nyelvtan nem lenne OAG, hiszen függőségek lennének benne, amelyek lehetetlenné tennék a kiértékelést.

$$A_{X,1} = \{X.a \in AS_X : \exists X.b \; hogy \; (X.a, X.b) \in IDS_X\}$$

Vagyis azok a *szintetizált* attribútumok, amelyekől *nem függ* semmi (X másik attribútuma).

$$A_{X,2n} = \{X.a \in AI_X : \exists (X.a, X.b) \in IDS_X \Rightarrow X.b \in A_{X.k} \; k \leq 2n\} - \bigcup_{k=1}^{2n-1} A_{X.k}$$

Ebben a páratlan menetben olyan örökölt attribútumokat veszünk bele a halmazba, amelyek a vizsgált attribútumtól függnek, vagy már benne vannak.

$$A_{X,2n+1} = \{X.a \in AS_X : \exists (X.a, X.b) \in IDS_X \Rightarrow X.b \in A_{X.k} \; k \leq 2n+1\} - \bigcup_{k=1}^{2n} A_{X.k}$$

Az előző lépés végrehajtása szintetizált attribútumokra.

Ezzel a lépéssel minden szimbólum attribútumait osztályokba soroltuk úgy, hogy az egyikben szintetizált, a másikban örökölt attribútumok helyezkednek el, s ez így halad páros-páratlan megosztásban.
A *kiértékelés* során először az *utolsó* partícióval kezdünk, hiszen ebben olyan attribútumok vannak, amik *nem függnek semmitől*.

4.2.5. Függőség szimbólumokra

A fentebb létrejött osztályok között vezetünk be függőségeket. *Szintetizáltból örököltbe, örököltből szintetizáltba.*

$$DS_X = IDS_X \bigcup \{(X.a, X.b) : X.a \in A_{X,k} \; es \; X.b \in A_{X,k-1}\}$$

4.2.6. Kiterjesztett függőségi gráf

Visszamásoljuk a bővített éleket vagyis képezzük DP és DS_X éleinek unióját.

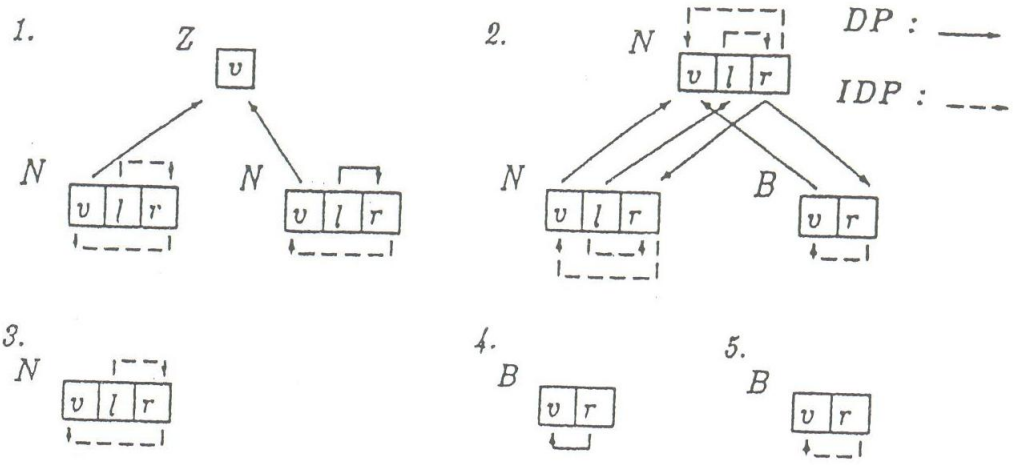
$$EDP_P = DP_P \bigcup \{(X_i.a, X_i.b) : X_i \; p - beli \; szimb \; es \; (X.a, X.b) \in DS_X, X = X_i\}$$

$$EDP = \bigcup_{p \in P} EDP_p$$

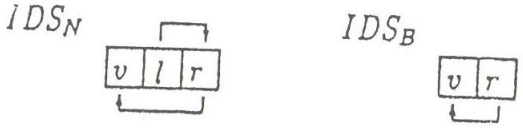
Ha EDP^+ nem ciklikus, akkor *AG OAG*.
Továbbá, ha bármely lépésben kapott gráf *cirkuláris*, megállunk.

Első lépés: direkt függőségek szabályokra (DP, az ábrán folyamatos vonal)

Második lépés: indukált függőségek szabályokra (IDP, az ábrán szaggatott és folyamatos vonal)



Harmadik lépés: redukált függőségek szimbólumokra (IDS)

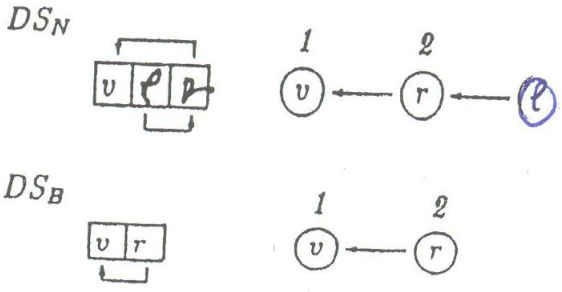


Negyedik lépés: partícionálás

$A_{N,1} = \{v\}$
$A_{N,2} = \{r\}$
$A_{N,3} = \{l\}$

$A_{B,1} = \{v\}$
$A_{B,2} = \{r\}$

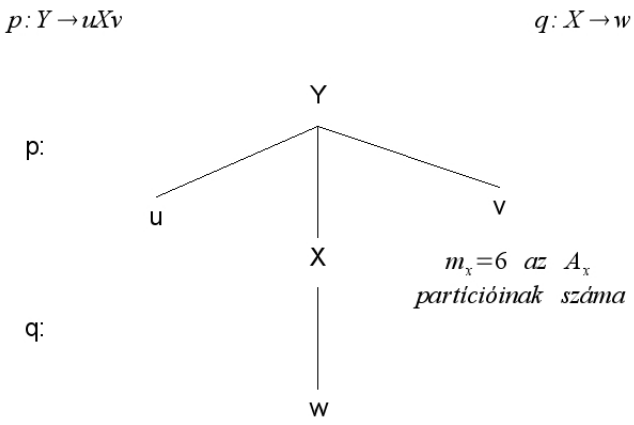
Ötödik lépés: függőségek szimbólumokra



A partícionálás nem hozott be új élekt, ezek már előbb is benne voltak.
Tehát a DS_N és DS_B ugyanaz, mint az IDS_N és IDS_B.

Akkor EDP=IDP, IDP nem ciklikus, akkor EDP sem, tehát OAG

12. ábra. OAG-e a fenti AG?



13. ábra. Visitsorozat működésének bemutatásához az alappélda

5. OAG visit sorozatok és implementációs sémák

Mint az korábban már láthattuk, az OAG esetén nincsenek előre rögzített bejárási stratégiák. Partícionálunk, s ezt követően *Visit sorozat*okat használunk a bejárás során.
Tehát minden $p \in P$-re egy VS_p visit sorozatot számolunk ki, ami akciók lineárisan rendezett sorozata.

Az alábbi eseteket különböztetjük meg:

- TC: elemzés után kezdődik a kiértékelés
- TU: top-down elemzéssel párhuzamos a kiértékelés
- BU: bottom-up elemzéssel párhuzamos a kiértékelés

5.1. Példa

Lásd:

13. ábra: az alapfeladat

14. ábra: TC és TU

15. ábra: BU

5.2. Összefüggések OAG esetén

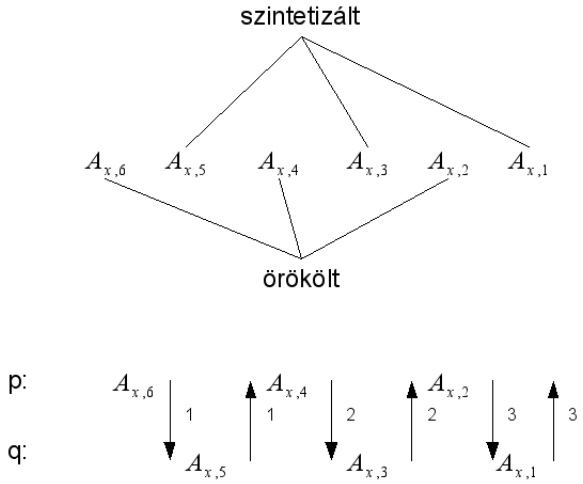
$$VS_P = AU_P \times AU_P$$

$$AU_P = AF_P \bigcup V_P$$

$$V_P = \{v_{k,i} | 1 \leq k \leq nv_X \ X_i = X\}$$

$V_{k,i}$: jelentése, hogy a i-edik nemterminálist k-adszorra járjuk be. Tehát $v_{k,0}$ a gyökér k-adik visitálását jelenti.

$p : X_0 \to X_1, \ldots, X_{np}$ esetén



14. ábra. TC és TU esetek

$$MAPVS\left(X_i.a\right) = \begin{cases} X_i.a & \text{ha } X_i.a \in AF_p \\ v_{k,i} & \text{ha } X_i = X \;\; X_i.a \in (A_{X,m} \bigcap AC_p) \;\; k = (f_X - m + 1) \, div \; 2 \; k > 0 \\ nem \; definialt & \text{ha } X_i = X \; X_i.a \in (A_{X,m} \bigcap AC_p) \;\; es \; k = 0 \end{cases}$$

$VS_p =$

$\{(MAPVS\left(X_i.a\right)), (MAPVS\left(X_j.b\right)) \,|\, (X_i.a, X_j.b) \in EDP_p\} \bigcup$

{ tetszőleges olyan él, hogy VS_p lineárisan rendezett legyen, és $v_{k,0}$ $(k = nv_X)$ a legnagyobb elem }

Ahol:

AC_p: az alkalmazott attribútum előfordulások

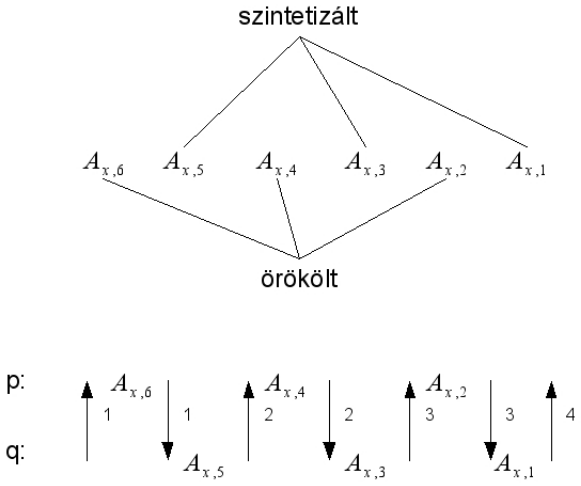
az unió második fele a VS_p-ben azért szükséges, mert a visitálás végén el kell hagynunk a derivációs fát a gyökérnél.

Továbbá látható, hogy az *alkalmazott előfordulású* attribútumokat visitekké képezzül le, hiszen ezek kiszámításához a fában valamerre el kell mozdulni, míg a *definiáló* előfordulásoknál nem.

5.3. VS implementálása rekurzív eljárásokkal

Az algoritmust *attribútumos fá*ra futtatjuk. Ennek csomópontja olyan adatszerkezet, ami tartalmazza:

- az attribútum példányokat
- referenciákat a leszármazottakra



15. ábra. BU eset

- egy indikátort (szám), hogy melyik sorszámú szabályt alkalmazzuk az adott csomópontnál (node)

Az alábbi visit-sorozatot bontjuk fel:

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

Annyi részre osztjuk, ahány gyökér visit van benne és minden részre egy-egy eljárást írunk.

```
 1. Procedure p_1(ref node X0)
 2.    Begin
 3.       .
 4.       .
 5.       .
 6.       Xi.a=f( )  {kiértékelés}
 7.       .
 8.       .
 9.       .
10.       q.k(Xj)    {visit a vk,j visit utasításra}
11.    End;
```

Ennél az algoritmusnál feltesszük, hogy az X_j szimbólum csak a q szabály baloldalán fordul elő. Itt már csak definiáló attribútum előfordulás, vagy $v_{k,i}$ $i > 0$ elem fordul elő.

Ha X_j baloldallal több szabály is előfordul, akkor az alábbi résszel módosul az algoritmus:

```
Case Xj.rule_indicator of
 q-1: q1-k(Xj)
 .
 .
 .
 q-n: qn-k(Xj)
Esac
```

5.4. Példa

A korábban már megismert bináris feladat kiértékelése az alábbi:

$$P_1 : v_{11}, N_{1.r}, v_{21}, v_{1,2}, N_{2.r}, v_{22}, v, v_{10}$$

$$P_2 : (v_{11}, l)\, v_{10}\, (N_{1.r}, v_{21}, B, v_{12}, v)\, , v_{20}$$

A visitsorozatokat részekre bontjuk, ahol van *gyökérre* való visszatérés, arra *eljárás*t írunk.

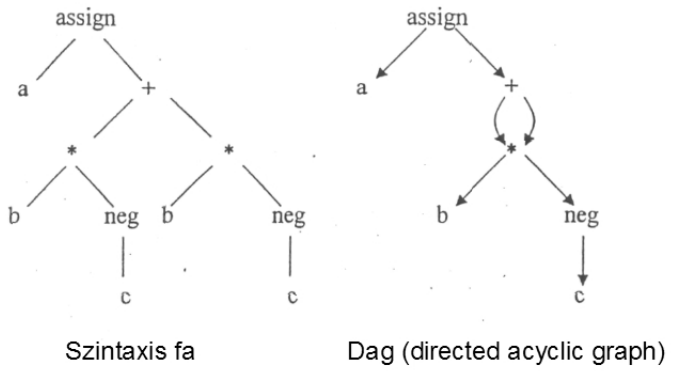
Első szabály:

```
 1. Proc P1(ref node Z)
 2. / visitek helyére eljárásthívás /
 3. / bemenetként a gyökeret kapja /
 4. / mivel két szabály van, case-t használunk /
 5. begin
 6.      case N1.rule_indicator of
 7.          P2: P2_1(N1) / 2-es szabály első része /
 8.          P3: P3_1(N1) / 3-as szabály első része /
 9.      esac
10.      N1.r := 0;
11. / N1.r áll a visitsorozatban. Ide írjuk az attribútum egyenletet. /
12. / az attribútum nyelvtant átmásoljuk a szabályba /
13.
14.      case N1.rule_indicator of {v21}
15.          P2: P2_2(N1) / 2-es szabály második része /
16.          P3: P3_2(N1) / 3-as szabály második része /
17.      esac
18.      case N2.rule_indicator of {v12}
19.          P2: P2_1(N2)
20.          P3: P3_1(N2)
21.      esac
22.      N2.r=-N2.l {N2.r}
23. / a szabályt átmásolása az attribútum nyelvtanból /
24.
25.      case N2.rule_indicator of {v22}
26.          P2: P2_2(N2)
27.          P3: P3_2(N2)
28.      esac
29.      v := N1.v + N2.v; {v}
```

Második szabály:
Mivel ké gyökér visit van, két eljárást kell írni.

```
30. proc P2_1(ref node N0)
31. begin
32.       case N1.rule_indicator of
33.           P2: P2_2(N1)
34.           P3: P3_2(N1)
35.       esac
36.       B.r := r;
37.
38.       case B.rule_indicator of {v12}
39.           P4: P4(B)
40.           P5: P5(B)
41.       esac
42.       v :=  N1.v + B.v;
43. end;
```

A további eljárásoknál már elegendő bemásolni az attribútum nyelvtan szabályait a *begin end* páros közé.



16. ábra. Szintaxis fa és dag

6. Kódgenerálás

A fordítás utolsó előtti fázisa, amely során jön létre a gépi kód. A mai fordítprogramoknál ez több lépésben végeztetik el.
Először egy *gépfüggetlen köztes kód* készül el, amelyből akár több, *különböző gépi kód* is generálható. Ilyen lehet pl. a szintaxis-fa, postfix jelölés, *háromcímes kód*.

6.1. Példa

Lásd 16. ábra.

Legyen a kifejezés az alábbi: $a := b * -c + b * -c$
Ekkor az alábbi szabályok adottak:

1. $S \rightarrow id\ '='\ E$

   $nptr := mknode('assign',\ mkleaf(id, id.place),\ E.nptr);$

2. $E \rightarrow E_1\ '+'\ E_2$

   $nptr := mknode('+',\ E_1.nptr,\ E_2.nptr);$

3. $E \rightarrow E_1\ '*'\ E_2$

   $nptr := mknode('*',\ E_1.nptr,\ E_2.nptr);$

4. $E \rightarrow '-'\ E_1$

   $nptr := mknode(neg,\ E_1.nptr);$

5. $E \rightarrow '('\ E_2\ ')'$

   $nptr := E_1.nptr;$

6. $E \rightarrow id$

   $nptr := mkleaf(id,\ id.place,);$

Ahol *mknode* egy csomópont létrehozásának parancsa, *mkleaf* pedig levelet hoz létre a fában. *E.nptr* a jobboldali részfa gyökere és *nptr* pedig egy csomópontra mutató pointer.

Ez az AG generálja a szintaxisfát. Ha azonos csomópontokra nem különböző csomópontokat hoznánklétre, nem referenciákat, akkor *dag*-et kapnánk.

6.2. Háromcímes kód

Mint korábban láttuk, ez is lehet a compiler gépfüggetlen, belső ábrázolási formálya.

6.2.1. Példa

Nézzük, hogy az előbb felírt kifejezés hogyan nézne ki háromcímes kóddal:
A szintaxisfából:
$t_1 := -c$
$t_2 := b * t_1$
$t_3 := -c$
$t_4 := b * t_3$
$t_5 := t_2 + t_4$
$a := t_5$

A dag-ból:
$t_1 := -c$
$t_2 := b * t_1$
$t_3 := t_2 + t_2$
$a := t_3$

6.2.2. Utasítások a háromcímes kódban

Háromcímes kódban az alábbi utasítások szerepelhetnek:

1. Bináris aritmetikai vagy logikai operátor (op): $x := y \ op \ z$
2. Unáris operátor: $x := \ op \ y$
3. Másolás operátora: $x := y$
4. Elágazások: $if \ x \ relop \ y \ goto \ L$
5. $param \ x \ es \ call \ p, n \ return \ y \ \ (ez \ opcionalis)$

   $param \ x_1$

   .

   .

   .

   $param \ x_n$ $p\,(x_1, \ldots, x_n)$-ből generálódik

   $call \ p, n$
6. $x := y\,[i]$, $x\,[i] := y$
7. x értéke y címe: $x := \&y$

   x étékre az y pointer által mutatott cím tartalmával egyenlő: $x := *y$

   az x pointer által mutatott objektum tartalma legy egyenlő y tartalmával: $*x := y$