Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Automatizálási és Alkalmazott Informatikai Tanszék

Lenkefi Péter

INKREMENTÁLIS GLR PARSER ÖSSZEHASONLÍTÁSA MÁS MÓDSZEREKKEL

KONZULENS

Somogyi Ferenc Attila

BUDAPEST, 2021

Tartalom

Összefoglaló

Manapság a fejlesztés szinte elképzelhetetlen anélkül, hogy gépelés közben a fejlesztőkörnyezet segítséget nyújtson kódírás közben. Ez a segítség lehet szemantikus színkiemelés, formázás, hibák aláhúzása, automatikus kiegészítési lehetőségek felajánlása, és így tovább. Mindezek alapja, hogy a háttérben valamilyen nyelvi segédeszköz fut, mely a gépelésre reagálva információt szolgáltat a kód jelenlegi állapotáról. A jó segédeszköz egyik fontos kritériuma, hogy az információ előállítása minél gyorsabb legyen, a felhasználó szempontjából szinte azonnalinak tűnjön.

A kód teljes egészében való újrafeldolgozása sok esetben megengedhetetlen, hiszen a kódfeldolgozás klasszikus lépései igen költségesek, számításigényesek. Szükség van tehát inkrementális módszerek bevezetésére, melyek a teljes újraszámítás helyett a korábbi eredmények módosításával dolgoznak, így sok esetben jelentősen csökkentve a költségeket.

A legtöbb nyelvi eszköz kulcslépése, hogy a szövegből szintaxisfát állít elő, mely egy strukturált, egyértelmű reprezentációja a kódnak. Ezt a lépést szokás szintaktikai elemzésnek hívni.

Dolgozatomban egy klasszikus elemzési módszer, az LR elemzők egy bővítését, a GLR elemzést és annak inkrementális adaptációját mutatom be. Kitérek a GLR előnyeire az LR elemzéssel szemben, illetve összevetem a módszert más lehetséges inkrementális megoldásokkal.

Abstract

Nowadays, it is almost impossible to imagine development without having a development environment, which aids us during typing. This help can take many forms, including semantic highlighting, formatting, underlining errors, autocomplete, and so on. This is usually achieved by a tool running in the background, that yielding information about the current state of the code as we type. A very important aspect of such a tool is a fast response time, ideally almost immediate from the perspective of the user.

Completely reanalyzing the code would be infeasible in most circumstances, as the classical steps of analysis are computationally intensive. We need to introduce some incremental methods, that try to adapt the previous results based on the changes made by the user, instead of redoing the entire computation.

The key step in most analysis tools is creating the syntax tree, which is a structured, unambiguous representation of the source code. This step is usually called syntax analysis.

In this thesis I will present an extension to the classical LR parsing method, the GLR parser and how that can be adapted to the incremental model. I will showcase the advantages of GLR over LR and compare the method with other incremental parsing solutions.

1 Bevezetés

A szoftverfejlesztésben egyre nagyobb szerepet kapnak a kódolást segítő különböző fejlesztőeszközök és az azok által nyújtott extra funkcionalitás. Ezzel párhuzamosan folyamatosak a törekvések az újabb és újabb programozási nyelvek kidolgozására, melyek változatos célkitűzésekkel próbálnak ki új megközelítéseket.

Bár a programozási nyelvek által nyújtott lehetőségek kulcsfontosságúak, azok elterjedéséhez és a valós életben való alkalmazásához mindenképp szükséges a megfelelő támogatás. A fejlesztőkörnyezetek elkészítése általában időigényes és nehéz, így rengeteg új nyelvnél tapasztalható, hogy igen keveset kapunk az alap funkciókon, például a szintaktikai kiemelésen kívül.

1.1 Motiváció

Az egyik legfontosabb információ, amit egy fejlesztőkörnyezet kinyerhet a kódból, az a program szintaxisfája, ezt a fordítási folyamatban a szintaktikai elemzés során teszi meg. A szintaxisfa alapján a segédeszköz képes szemantikai hibákat jelezni, szemantikus kiemelést adni, kódkiegészítést nyújtani és így tovább. Ez a szintaktikai elemzés működhet egy nyelvtani leírás alapján, így sokszor a parser megírása teljesen automatizálható.

Ennek az elemzésnek akkor is hatékonyan és gyorsan kell működnie, amikor a felhasználó több, apróbb módosítást végez a kódon. Ennek oka, hogy a fejlesztőkörnyezet akkor lehet igazán hasznos, ha például egy esetleges kódkiegészítést előbb kínál fel, mint ahogy azt a felhasználó legépelte volna. Emiatt a parsert érdemes inkrementális módszerekkel megvalósítani.

Számos módszer létezik, melyek gyökerei az 1960-as években kialakult elemzési technikákra épülnek. Ezek jelentősen különböznek működési elvben, illetve korlátokban. Egyeseket könnyű kézzel megvalósítani, míg másokat kódból generálnak. Egyesek az általánosságra törekszenek – hogy a lehető legtöbb nyelvet lefedjék –, míg mások a hatékonyságra törekszenek. Az egyik ilyen régmúltban gyökerező módszer a GLR parser, mely mind általánosságot, mind hatékonyságot igyekszik elérni.

1.2 Célkitűzés

A dolgozat célja az inkrementális GLR parser összevetése más automatizálható elemzési technikákkal. A GLR parsert két szempont miatt vettem a dolgozatom középpontjába. Egyrészt ez egy igen hatékony módszer, az LR parserek általánosítása. Másrészt mind az LR, mind pedig a GLR körül igen kevés az átfogó szakirodalom. Éppen emiatt a kapcsolódó irodalmi szakasszal szeretnék egy áttekintést adni erről.

Emellett szeretném pótolni azt, ami szerintem igen hiányosan jelenik meg a szakirodalomban, de fontos a GLR hatékony és helyes működéséhez: a Graph Structured Stack – vagy röviden GSS - műveletek algoritmusának egy lehetséges megvalósítása. Ez egy olyan adatszerkezet, mely biztosítja a GLR algoritmusának hatékony futását.

A munkám során kifejlesztettem egy keretrendszert, mely parserek statikus és dinamikus vizsgálatát segíti. Képes az elemzést lépésenként futtatni, illetve vizualizálni a különböző komponensek állapotait. Röviden ezekre a képességekre, illetve a keretrendszer felépítésére is kitérek. A keretrendszer célja részben az elemzési folyamat vizuális nyomon követése, részben pedig a különböző módszerek hatékonyságának mérése. A mérésekkel célom igazolni azt, hogy az inkrementális GLR elemzés sokkal hatékonyabb, mint a teljes folyamat újrafuttatása.

1.3 A dolgozat felépítése

A 2. fejezetben bemutatom azokat a szükséges alapokat, melyek az inkrementális GLR parser – és egyéb parserek – működésének megértéséhez szükségesek. A 3. fejezetben bemutatom a GLR működéséhez szükséges GSS adatszerkezet műveleteit – egy saját implementációval –, valamint az általam kifejlesztett keretrendszert. A végén meghatározom a mérések paramétereit, a mérőszámokat, illetve, hogy milyen eredményeket várok. A 4. fejezetben bemutatom a mérések menetét, illetve a különböző parserek által adott eredményeket. Rövid konklúziót is fűzök a különböző mérésekhez. Reményeim szerint a konklúzió némi iránymutatással szolgál annak, aki inkrementális parsert készít. Az 5. fejezet összefoglalja a dolgozatot, kitérve a továbbfejlesztési lehetőségekre is.

2 Kapcsolódó irodalom

2.1 Alapfogalmak

A további alfejezetek olvasásához szükséges néhány alapfogalom, illetve jelölési konvenció ismerete. Ezeket alább röviden ismertetem. A pontos definíciók megtalálhatók bármely formális nyelvekkel foglalkozó könyvben vagy jegyzetben [1], itt csak a lényeget emelem ki, mely a dolgozat megértésében segít.

A *szintaktikai elemző* – angolul parser – egy olyan programrész, mely a szöveges forráskódból *szintaxisfát* állít elő. A szintaxisfa a programkód szintaktikailag egyértelmű reprezentációja – például: precedencia feloldása, szóközök elvetése, stb.

A *tábla-vezérelt elemzők* olyan parserek, melyek az aktuális állapotuk és a bemenet alapján egy előre legenerált táblázat alapján döntik el, milyen művelet következik.

A kontextusfüggetlen – röviden CF – nyelvek a kontextusfüggetlen nyelvtanok által generált nyelvek.

A *kontextusfüggetlen nyelvtanok* szabályai a *produkciós szabályok* – nevezhetők még helyettesítési szabályoknak is –, melyek alakja általánosan jelölve $A \rightarrow \alpha$. Az A itt egy nemterminális, az α – és általában a görög betűk – pedig terminálisok és nemterminálisok tetszőleges láncát jelöli. A kontextusfüggetlen nyelvtanoknak van egy kezdőszabályuk, mely a levezetések kiindulási pontját jelöli. Például az 1. nyelvtan az összeadásokból, szorzásokból és zárójelezésekből álló kifejezéseket írja le.

$$E \rightarrow E + E$$
$$E \rightarrow E * E$$
$$E \rightarrow (E)$$
$$E \rightarrow \text{number}$$

1. nyelvtan: Összeadást, szorzást és zárójelezést támogató nyelvtan.

Ha egy produkciós szabály balján többször szerepel ugyanaz a szimbólum, például az 1. nyelvtan esetében az E, a jobb oldal kompaktabb formában írható a „|” jellel való összefűzéssel. Az 1. nyelvtan efféle rövidítését mutatja a 2. nyelvtan.

$$E \rightarrow E + E \mid E * E \mid (E) \mid number$$

2. nyelvtan: Az 1. nyelvtan rövidített formája.

A *terminálisok* olyan szimbólumok, melyek csak a produkciós szabályok jobb oldalán szerepelhetnek, és tovább nem helyettesíthetők.

A *nemterminálisok* olyan szimbólumok, melyek szerepelhetnek a szabályok bal oldalán. Ezek helyettesíthetők a jobb oldalukon található szimbólumláncra.

2.2 Top-down és bottom-up parsing

A parsereket általában két nagy csoportra bontják, az alapján, hogy milyen irányban járják be a nyelvtant, ami az általuk felismert nyelvet definiálja.

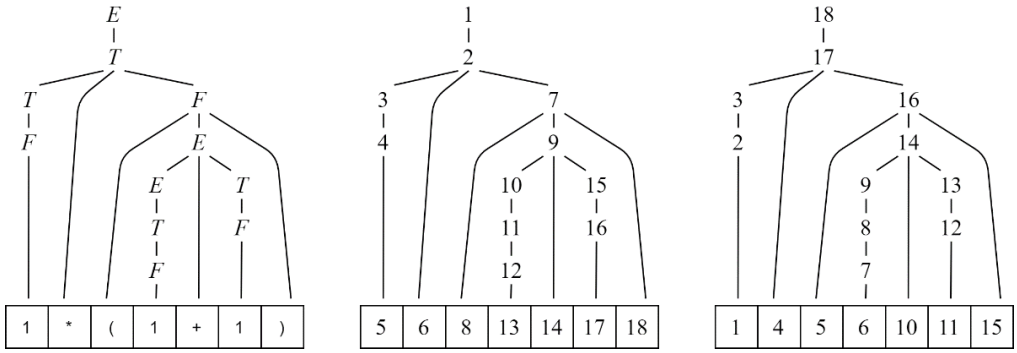
A *top-down parserek* a kezdeti nyelvtani szabályból kiindulva, egyre alacsonyabb szintű konstrukciókat ismernek fel, míg végül elérik a terminálisokat [2]. Előnyük, hogy emiatt igen könnyű kézzel implementálni őket, az esetek nagy részében elegendő a nemterminálisokat függvényhívássá, a terminálisokat pedig a bemenettel való egyeztetéssé fordítani – ez lényegében a recursive-descent parsing. Hátrányuk, hogy általában lassabbak, mint a bottom-up parserek – főleg, ha az említett naiv, kézzel írt implementációt használjuk -, illetve a balrekurziót manuálisan kell a nyelvtan transzformálásával eltüntetni. Balrekurziónak nevezzük az $A \rightarrow A\,\alpha$ alakú produkciós szabályokat. A balrekurzió recursive-descent parserre való leképezése végtelen, feltétel nélküli rekurzív hívásokhoz, ami pedig végül verem túlcsorduláshoz vezet.

A *bottom-up parserek* a terminálisokból kiindulva építenek egyre magasabb szintű konstrukciókat, míg végül elérik a kezdeti szabályt [2]. Általában a bemenetet egyszer olvassák, így igen gyorsak lehetnek. Bár igen sok variációjuk van, általánosságban azt mondhatjuk, hogy kézzel igen nehéz implementálni, általában ezt egy szoftverre bízzuk – ilyen például a Bison [3]. A két módszer összehasonlításához a 3. nyelvtan alapján a műveletek sorrendjét mutatja az 1. ábra.

Vannak technikák, melyek vegyítik a két módszer elemeit. Ilyen például, amikor egy recursive-descent parser a terminális alapján dönt a következő elemzési szabályról, ez az úgynevezett left-corner bottom-up filtering [4].

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid 1$$

3. nyelvtan: Egyszerű kifejezés nyelvtan.



1. ábra: Az 3. nyelvtan egy bemenetre való levezetése, a műveletek egy lehetséges sorrendje top-down, illetve bottom-up elemzés esetén.

Mivel a bottom-up parserek általában hatékonyabbak [5] és több, bevált továbbfejlesztési módszer van hozzájuk kidolgozva [6] [7], emiatt döntöttem úgy, hogy ezek irányába indulok el.

2.3 Shift-Reduce parsing

A bottom-up parserek egyik leggyakoribb megvalósítási módszere az úgynevezett shift-reduce parsing. Ezek általában tábla-vezérelt parserek, melyek pontosan egyszer dolgozzák fel az összes terminálist a bemenetből. A részeredményeket és aktuális állapotukat egy vermen tárolják [2].

Az elnevezésük a két fő műveletből áll, amit elemzés közben végeznek:

- Shift: A terminális és az új állapot a verem tetejére kerül.
- Reduce: Egy helyettesítési szabály alapján a verem tetején levő elemekből fát épít.

Például a 3. nyelvtan alapján az „1*1” bemenet elemzése:

Bemenet	Művelet	Verem
1*1	**shift** 1	
1*1	**reduce** F → 1	1
1*1	**reduce** T → F	F
1*1	**shift** *	T
1*1	**shift** 1	T *
1*1_	**reduce** F → 1	T * 1
1*1_	**reduce** T → T * F	T * F
1*1_	**reduce** E → T	T
1*1_	**accept**	E

2.1. táblázat: A 3. nyelvtan shift-reduce elemzése.

2.4 LR parsing

Az LR parserek a bottom-up parserek egy olyan csoportja, melyek egy determinisztikus veremautomata elvén készülnek. Általában tábla-vezéreltek és shift-reduce parserként valósulnak meg. Az LR elnevezés a ***L****eft-to-right,* ***R****ightmost derivation in reverse* kifejezésből jön, azaz a bemenetet balról jobbra olvasva, a nyelvtan szerinti jobboldali levezetést állítja elő, mindezt fordított sorrendben – tehát a levezetés (és szintaxisfaépítés) jobbról balra történik [2] [8].

Sok fajtájuk létezik, mindegyik a CF nyelvek 1-1 különböző méretű részhalmazát ismerik fel. Az LR(k) kifejezés arra utal, hogy az adott LR parser k darab terminálist tekinthet előre az automata konstruálásakor [2] [8].

Mivel a módszerek mind más és más részhalmazát ismerik fel a CF nyelveknek, így a parserek definiálják az általuk felismerhető nyelvek halmazát. Például az LR(0) parserekre azt mondjuk, hogy az LR(0) nyelveket képesek felismerni.

2.4.1 Az LR parserek működése

Bár a különböző LR parserek tábláinak konstrukciója másként történik, maga az elemzés a tábla elkészültével megegyezik. A táblák formátuma is ugyanaz, a tábla maga két félre bontható:

- *Action(S, T)*: A jelenlegi állapotban (S), az épp feldolgozott terminálisra (T) milyen művelet kerüljön végrehajtásra (Shift/Reduce).

- *Goto(S, N)*: A jelenlegi állapotban (S) egy redukciós lépés után (N) milyen állapotba ugorjon az automata.

A táblázatba – bár a nyelvtanban ezt explicit nem szokás jelölni – a bemenet vége szimbólumot is bevesszük, konvenció szerint $ jellel jelöljük.

Maga az algoritmus egy vermet kezel, melyre a részeredményeket és az aktuális állapotot rakja fel, illetve veszi le. A teljes működést a 2.1. kódrészlet mutatja be [2].

```
stack := Stack()
stack.Push(INITIAL_STATE)
term := ReadTerminal()
While True:
    state := stack.Top()
    action := Action(state, term)
    If action is Shift(s):
        stack.Push(s)
        term := ReadTerminal()
    If action is Reduce(L → α):
        stack.Pop(|α|)
        s := stack.Top()
        stack.Push(Goto(s, L))
```

2.1. kódrészlet: Az LR parserek algoritmusa.

2.4.2 LR(0)

Az LR(0) a legegyszerűbb az LR parser módszerek közül. Bár a gyakorlatban ritkán használjuk, mivel az általuk felismert halmaz igen kicsi, lényegében az összes többi módszer alapját képezi [2] [9]. A módszer ismertetéséhez szükség van néhány alapfogalomra:

LR elem: Egy LR elem a nyelvtan egy olyan produkciós szabálya, amely egy kurzorral (•) van ellátva annak jobb oldalán valamelyik szimbóluma előtt vagy után. A kurzor azt jelöli, hogy az automata hol tart az aktuális szabály elemzésével. Például a 3. nyelvtan első szabályához az $E \rightarrow \bullet\, E + T$, a $E \rightarrow E \bullet + T$ és $E \rightarrow E + T \bullet$ mind LR elemek. Azokat az elemeket, melyek elején van a kurzor, *kezdeti elemnek*, azokat, amelyek végén van pedig *végső elemnek* szokás hívni. Ez alapján a $E \rightarrow \bullet\, E + T$ kezdeti elem, míg a $E \rightarrow E + T \bullet$ végső elem.

LR elemhalmaz lezártja: Az LR elemeket a konstrukció során halmazokban kezelik. Az LR elemhalmaz lezártjának azt tekintjük, amikor minden olyan produkciós szabályt – kurzorával az elején – is hozzáveszünk az eredeti halmazhoz, melynek a bal oldala szerepel egy a már halmazban levő elem kurzorának jobb oldalán – mindezt rekurzívan addig

ismételve, amíg új elem kerül a halmazba. Például a 3. nyelvtan alapján a $\{ E \rightarrow \bullet E + T, E \rightarrow E \bullet + T \}$ lezártja a $\{ E \rightarrow \bullet E + T, E \rightarrow E \bullet + T, E \rightarrow \bullet T, T \rightarrow \bullet T * F, T \rightarrow \bullet F, F \rightarrow \bullet (E), F \rightarrow \bullet 1 \}$ halmaz.

A konstrukció menete

A konstrukció alapja, hogy az automata egyszerre követi az összes szabály összes lehetséges állapotát az LR elemek segítségével. Az automata egy állapotát így egy LR elemhalmaz képezi. Az állapotokból a kurzorok utáni szimbólumok alapján lehet tovább lépni, a konstrukció így deríti fel a további állapotokat [2] [9].

Az algoritmus menete LR(0)-hoz:

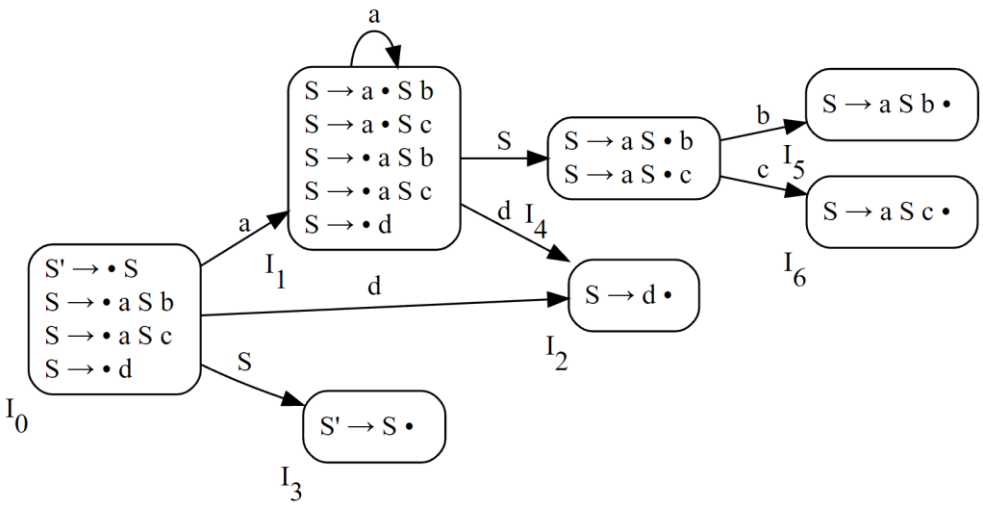
1. A G nyelvtanba kerüljön egy új kezdőállapot. Ha a korábbi kezdőállapot S volt, akkor kerüljön be egy $S' \rightarrow S$ szabály.
2. Legyen a kezdeti állapot a $\{ S' \rightarrow \bullet \mathrm{S} \}$ lezártja.
3. Legyen A egy még nem vizsgált állapot.
   a. Az A állapot azon LR elemei, melyek $X \rightarrow \alpha \bullet \mathrm{T}\beta$ alakúak és T egy adott terminális G-ben, a kurzorok előléptetésével képezzenek egy új halmazt. Ennek lezártja egy új állapot, B lesz. A terminális átlépése bekerül a műveleti táblába: *Action(A, T) = Shift(B)*.
   b. Az A állapot azon LR elemei, melyek $X \rightarrow \alpha \bullet \mathrm{N}\beta$ alakúak és N egy adott nemterminális G-ben, a kurzorok előléptetésével képezzenek egy új halmazt. Ennek lezártja egy új állapot, B lesz. A nemterminális átlépése bekerül az ugrás táblába: *Goto(A, N) = B*.
   c. Ha az A állapot tartalmaz $X \rightarrow \alpha \bullet$ végső elemet, akkor minden G-beli T terminálisra legyen *Action(A, T) = Reduce($X \rightarrow \alpha$)*.
4. Ha már nem keletkezik további állapot, a táblák összes többi helyére kerüljön hiba bejegyzés.

1. algoritmus: Az LR parserek tábláinak konstrukciója.

Az algoritmus felépítése ugyanez az összes többi LR módszernél, apróbb finomítások vannak benne csupán. Példaként a 4. nyelvtan konstrukciójához tartozó állapotgépet a 2. ábra tartalmazza.

$$S \rightarrow a\ S\ b$$
$$S \rightarrow a\ S\ c$$
$$S \rightarrow d$$

4. nyelvtan: Egyszerű nyelvtan LR(0) konstrukcióhoz.



2. ábra: A 4. nyelvtan állapotgépe az 1. algoritmus alapján.

Konfliktusok

Felmerülhet a kérdés, hogy az 1. algoritmus futtatása során történhet-e olyan, hogy egy műveleti bejegyzés felülír egy másikat. Ez mindegyik módszernél előfordulhat, és konfliktusnak nevezzük.

Két *Shift* művelet között természetesen nem fordulhat elő konfliktus, viszont *Shift* és *Reduce*, illetve két *Reduce* művelet között igen:

- A *Shift-Reduce* konfliktus abból fakad, hogy az automata nem képes eldönteni azt, hogy a következő terminális kerüljön-e föl a verem tetejére, vagy a vermen levő elemekre alkalmazzon redukciót. Tipikus példája ennek a *dangling-else* probléma, például a következő C kódrészletnél:

  ```
  if (x) if (y) a(); else b();
  ```

  Ebben az esetben az LR parserek nem képesek eldönteni, hogy az *else* kulcsszó a külső vagy belső elágazáshoz tartozik-e. A legtöbb parser-

generátor egyszerűen ilyen esetekben mindig a *Shift*-et választja – ami általában elegendő és helyes[1] [3].

- A *Reduce-Reduce* konfliktus azt jelenti, hogy a verem tetején levő elemek többféle helyettesítési szabálynak felelnek meg. A feloldásukra nincs egyszerű heurisztika, a nyelvtan átfogalmazására van szükség.

Ha egy módszer által készített táblában konfliktusok vannak, akkor azt mondhatjuk, hogy a nyelv nincs benne az adott LR módszer nyelvtanában. Például, ha egy nyelvtan LR(0) táblájában konfliktus van, akkor a nyelvtan nem LR(0). Ez egy biztos, statikus vizsgálata a nyelvtan osztályozásának [9].

Az LR(0) módszerrel generált táblák igen sok konfliktust okoznak, egy igen egyszerű okból kifolyólag: Egy végső elem hatására az adott állapot *Action* sorát mohón végig írja *Reduce* műveletekkel anélkül, hogy bármiféle tekintettel lenne arra, hogy ez a következő terminális alapján helyes-e. A további módszerek azon finomítanak, hogy hogyan lehet kevesebb konfliktust okozni, akár a nyelvtan finomabb elemzésével, akár az állapotok számának növelésével.

2.4.3 SLR

A **S**imple **LR** – vagy SLR – az LR(0) konstrukció egy nagy gyengeségét javítja ki. Ahelyett, hogy egy *Reduce* műveletet minden terminálishoz odaírna, a nyelvtan vizsgálatával kideríti, hogy milyen terminális követheti egyáltalán a redukciós szabály bal oldalán található szimbólumot [2] [9].

Például, ha az $X \rightarrow \alpha$ redukcióhoz kell *Reduce* műveleteket feljegyezni, akkor azt nem szükséges minden terminálishoz, elegendő csupán a nyelvtan szerint az X szimbólumot közvetlenül követő terminálisokra megtenni ezt. Ez például a 4. nyelvtan S szimbólumára a b és c terminálisok.

[1] Ha konfliktus esetén egy ilyen heurisztika szerint döntünk, általános esetben semmi sem garantálja, hogy a parser helyes marad. Előfordulhat, hogy a művelet után sokkal később derül ki, hogy a másik műveletet kellett volna választani, és a parser elakad.

Némi komplexitás árán – a követő terminálisok helyes meghatározása kissé bonyolult – jelentősen csökken a potenciális konfliktusok száma, így ez a módszer akár a gyakorlatban is használható.

2.4.4 LR(1), avagy CLR

Ahhoz, hogy még több nyelvet legyen képes elfogadni az LR parser, a konstrukciónál érdemes lehet egy terminálist előre tekinteni a nyelvtanban. Erre utal az LR(1)-ben a k=1 paraméter. Az LR(1)-et szokás még CLR-nek is nevezni, a **C**anonical **LR** kifejezésből [2] [9].

Az LR elemekhez a produkciós szabály és a kurzor mellé bekerül egy ún. *lookahead* terminális is. A lezárt képzése csupán annyiban változik, hogy a kurzor utáni részt vizsgálni kell az ott lehetségesen előforduló terminálisok miatt, hasonló módszerrel, mint SLR esetében. Mivel egy szabályt általában igen sokféle terminális követhet, az LR(1) állapotterek és táblák sokszor több nagyságrenddel nagyobbak, mint az LR(0) táblák. Már egy olyan igen egyszerű nyelvtanra is, mint a 3. nyelvtan, az LR(0) módszerre 12, míg LR(1)-re 22 állapot jön létre.

Mivel a gyakorlatban az LR(1) táblák óriásira nőhetnek, aminek már a tárolása is gondot okozhat, nem szokás használni, alapgondolata viszont megalapozza a következő módszert.

2.4.5 LALR

Az LR(1) táblák generálásakor megfigyelhető, hogy igen sok állapot szinte azonos, csupán az LR elemek *lookahead*-jei térnek el. A LALR módszer – mely elnevezése a **L**ook**A**head **LR**-re utal – ezekkel az állapotokkal kíván spórolni [2] [9]. A tábláját legegyszerűbb úgy előállítani, ha a legenerált LR(1)-es automatában az ilyen hasonló állapotok összevonásra kerülnek. Mivel az LR(1) tábla előállítása sokszor igen tárhelyigényes, a generátor eszközök egy alternatív módszert szoktak használni, ami jelentősen bonyolultabb.

A LALR állapottere lényegben ekvivalens az LR(0) – és emiatt az SLR – állapotterével, az LR(1)-nél gyengébb, viszont mind az LR(0)-hoz, mind az SLR-hez képest

jobb a stratégiája a *Reduce* műveletek megfelelő terminálishoz való bejegyzéséhez. Egy LR(1) nyelvtanhoz képest viszont behozhat *Reduce-Reduce* konfliktusokat [2].

A gyakorlatban ez az egyik leginkább alkalmazott módszer a kis állapottere miatt. Ezzel együtt a bemutatott módszerek között messze ennek a legnehezebb a tábláját előállítani.

2.4.6 Az LR nyelvosztályok hierarchiája

Mint arra már többszöri utalás is történt, a különböző LR módszerek különböző erősséggel bírnak. Az általuk felismert nyelvosztályok hierarchiája a következő [10] [11]:

$$LR(0) \subset SLR \subset LALR \subset LR(1) \subset LR(k), k > 1 \subset CF$$

Minden véges k-ra azonban az LR(k) nyelvosztály csupán valós részhalmaza a teljes CF nyelvosztálynak. Ez egyszerűen abból fakad, hogy az LR parserek az egyértelmű nyelvtanokat képesek felismerni, a CF nyelvtanok viszont lehetnek többértelműek – melyet hívunk *nemdeterminisztikusnak* is.

2.5 Generalized LR

Az LR parsereknek két hátrányuk van, melyek korlátozzák a használatukat a gyakorlatban:

- A determinisztikus CF nyelveknek csupán valós részhalmazát ismerik fel – hiszen a k korlát mindig adott. Ez gondot okozhat egy bonyolultabb programnyelv nyelvtanánál, még akkor is, ha az determinisztikus.
- Nem képesek felismerni a nemdeterminisztikus nyelveket. Ilyenek azok a nyelvek, melyeknek szintaktikai értelmezésük függ a kontextustól, például a C vagy C++ nyelvek bizonyos elemei.

Erre a két problémára nyújt megoldást a ***G**eneralized **LR*** – vagy röviden GLR – parsing [12] [13].

Az LR parserek csupán a valóságban tényleg lehetetlen eseteket tüntetik el a konfliktusok közül. A nemdeterminisztikus, illetve a *k*-nál több előre tekintést kívánó nyelvtanok azonban így is konfliktust fognak okozni a táblában.

A GLR alapötlete, hogy a konfliktusok felfoghatók nemdeterminisztikus lépésként is, tehát egy konfliktus esetén a lehetséges összes művelet párhuzamosan megtörténik. Ezután a további elemzés párhuzamosan folytatódik, ahol az ágak a további nemdeterminisztikus lépések során tovább ágazhatnak. Egy ágnak a futás végére két sorsa lehet:

- Ha egy ág valahonnan nem tudott tovább lépni, akkor hibás állapotba kerül, ezt az ágat nem szükséges folytatni. Ez azt jelenti, hogy az elágazás helyén azághoz tartozó művelet helytelen volt, és az LR parser limitált képességei miatt jött létre.
- Ha egy ág a bemenet végén elfogadta a bemenetet, az egy helyes elemzése az adott bemenetnek. Az összes ilyen ág ekkor az összes lehetséges, helyes elemzést tartalmazza. Ezzel a nemdeterminisztikus nyelvtanokra a GLR az összes eredményt képes megadni.

Ez az ötlet éppen az általánossága miatt az összes LR módszerre ráhúzható, akár LR(0), SLR, LALR vagy LR(1)-re. Természetesen a futás közbeni szétágazások és az érvénytelen, terminált ágak száma a módszertől függően változik. A GLR igazán hatékony működéséhez szükséges megismerni a *Graph Structured Stack* nevű adatszerkezetet, melynek alapgondolatát a 2.6 fejezet tárgyal részletesen, a 3.1 fejezetben pedig egy lehetséges implementációt adok rá.

2.6 Graph Structured Stack

A ***G**raph **S**tructured **S**tack* – röviden GSS – nevű adatszerkezet a GLR hatékony működését biztosítja [14] [15]. Bár a természetes nyelvek és programozási nyelvek elemzése és feldolgozása igen eltérő, a nemdeterminizmusból következő probléma könnyebben és tömörebben ábrázolható egy természetes nyelven. A megfigyelések – ha csak nagyobb bemeneten is, de – érvényesek az olyan kontextusfüggő szintaxissal rendelkező nyelvekre is, mint a C vagy C++.

Emiatt a további példákat az 5. nyelvtan fogja kísérni, mely egyszerű angol kijelentő mondatokat ír le [14]. A *det* a *determiner*, a *prep* pedig a *preposition* rövidítése. Az angol

nyelvben a *determiner* határozószó (például *a*, *the*, *every*), a *preposition* elöljáró szó (*to*, *in*, *with*), a *noun* főnév (*man*, *bed*, *telescope*), a *verb* pedig ige (*see*, *run*, *sat*).

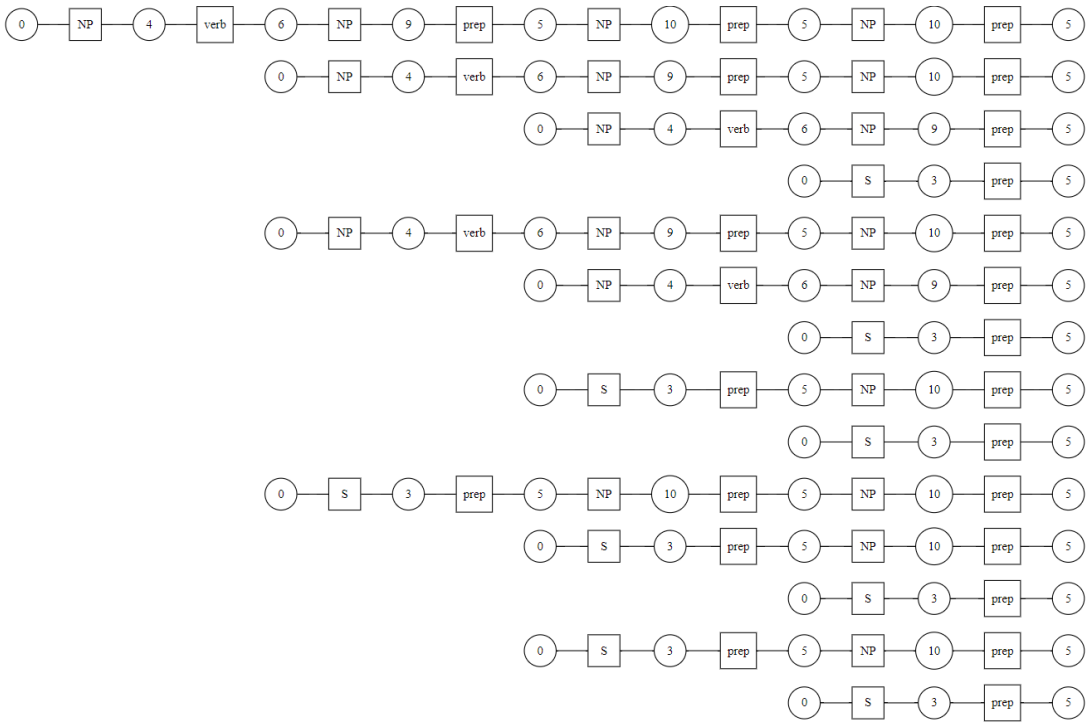
$$S \rightarrow NP\ VP \mid S\ PP$$
$$NP \rightarrow \text{noun} \mid \text{det noun} \mid NP\ PP$$
$$PP \rightarrow \text{prep}\ NP$$
$$VP \rightarrow \text{verb}\ NP$$

5. nyelvtan: Egy egyszerű nyelvtan angol kijelentő mondatokra.

Egy naiv GLR implementáció párhuzamosan vezeti a vermeit, melyeket lemásol, ha nemdeterminisztikus lépéshez ér, illetve megszüntet, ha érvénytelen állapotba kerül. A 3. ábra mutatja, hogyan néznek ki a párhuzamosan vezetett vermek az 1. példa „*with*” szavának feldolgozása után, LALR táblát használva a GLR parserhez.

„*I saw a man on the bed in the apartment with a telescope*”
noun verb det noun prep det noun prep det noun prep det noun

1. példa: Egy többértelmű angol kijelentő mondat, illetve az abból készült terminálisok.



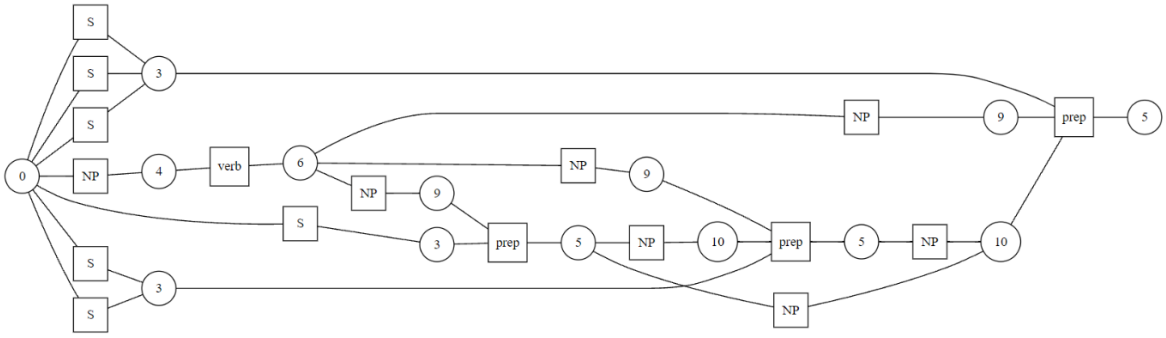
3. ábra: A GLR párhuzamosan vezetett vermei az 1. példa „with” szavának feldolgozása után.

Konvenció szerint az ábrán a vermek balról jobbra nőnek – az aktuális állapotok a jobb oldalon vannak –, a kör alakú elemek az állapotok, a négyzet alakúak pedig az elemzett szimbólumok (melyek a szintaxis részfát is tárolják).

A példán jól látszik, hogy már egy ilyen rövid bemenet esetén is 14 veremre ágazik szét a parser állapota. Ez a szétágazás és elkülönítés több szempontból hátrányos:

- Többletmunka a parser számára, hiszen minden egyes vermen végig kell vezetnie a műveleteket.
- A parser sok redundáns műveletet végez, hiszen a vermek sokszor jutnak ugyanabba az állapotba – a példa szerint jelenleg például minden verem tetején az 5-ös állapot van –, melyeket nem szükséges megkülönböztetni. Az egyidőben ugyanolyan állapotú vermek ugyanúgy fognak viselkedni.
- Az elkülönített állapotok mellett a részfák is el lesznek különítve, így várhatóan a parser többször is felépíti ugyanazokat a részfákat, ami jelentős memóriaallokációval járhat.

Mindezeken javíthat, ha az ugyanolyan legfelső állapottal levő vermek összevonásra kerülnek. Mivel így a verem elemei nem csak szétágazhatnak, hanem egyesülhetnek is, a struktúra így már egy irányított gráf lesz, emiatt kapja az adatszerkezet a *Graph Structured Stack* nevet. A 3. ábra állapotát ábrázolja újra, immár GSS-sel a 4. ábra.



4. ábra: A GLR GSS-e az 1. példa „with” szavának feldolgozása után.

A 126 elemből álló 14 párhuzamos veremből egy mindössze 31 elemből álló gráf lett, mely az adott pillanatban egyetlen felső állapottal rendelkezik. Mindemellett a részfák megosztása is sokkal könnyebbé válik. A GSS legfelső elemeire szokás fejekként (*head*) is

hivatkozni. Mivel immár inkább gráfról van szó, az elemeket is helyénvaló *csomópontoknak* hívni, melyekbe a fejek is beleértendők.

2.7 Incremental GLR

Az algoritmusok inkrementálissá tétele általában az eredmény, vagy részeredmények, illetve az algoritmus állapotának valamiféle tárolásával jár. A GLR-hez többféle ilyen módszer létezik, de talán a legkézenfekvőbb az, amit az ***I**ncremental **S**cannerless **GLR*** – röviden ISGLR – algoritmus ír le [16].

A bemenet immár nem terminálisokból, hanem részfákból áll. Ez kezdeti elemzésnél azt jelenti, hogy minden egyes terminális egy egyelemű fa. A fák építésénél minden egyes elem eltárolja, hogy a bemenet mely részéből származik, milyen állapot volt a verem tetején amikor elkészült, illetve, hogy újrafelhasználható-e.

Egy változtatásra az elemzés két lépésre bomlik:

- Egy előfeldolgozási lépés: A korábbi kielemzett fán megkeresi a törölt bemenetet reprezentáló levélelemeket, és azokat kitörli a fából. Az újonnan beszúrt terminálisokat a fa megfelelő helyein a levélelemek közé illeszti be. Ezen műveletek meggyorsítása miatt tárolják a fák azt, hogy a bemenet mely részéről származnak[2]. Minden egyes elemet, melyhez beszúrtak, vagy melyből eltávolítottak, nem újrafelhasználhatóként jelöli meg. Valamint minden nem újrafelhasználható elem szülőjét is nem újrafelhasználhatóként jelöli meg, egészen a gyökérelemig.
- A tényleges elemzési lépés: A tényleges elemzés kezdetben az előfeldolgozott fát helyezi a bemenetre. Amikor a parser olvasni akar és egy olyan részfát olvas, ami nem újrafelhasználhatóként van megjelölve, szétbontja azt a gyerek elemeire. Ezt egészen addig folytatja, amíg egy újrafelhasználható elemet nem sikerül olvasnia. Ekkor ellenőrzi, hogy az elemben tárolt állapot megegyezik-e az aktuális állapottal. Ha igen, a részfa újrafelhasználható – és

[2] A gyakorlatban érdemes inkább a fák szélességét tárolni terminálisokban mérve, így nem kell a szerkesztés miatt fellépő esetleges eltolódásokkal bajlódni.

egyszerűen a verem tetejére kerül –, különben ugyanúgy tovább kell bontani, mint a nem újrafelhasználható elemeket.

A módszer hátránya, hogy nem képes kezelni a nemdeterminisztikus környezetben keletkezett részfákat – tehát azokat, melyek többfele ágazó verem esetén keletkeztek. Ezeket ugyanis szigorúan nem újrafelhasználhatóként kell létrehozni, ellenkező esetben lehetséges, hogy egy korábban terminált ágat nem jár be újra a parser, ami ezúttal sikeres volna. Ettől függetlenül a módszer igen sokat javít a gyakorlatban [16].

Megjegyzés: Bár ez a módszer az ISGLR algoritmushoz született, bármely LR parser inkrementálissá tételéhez alkalmazható [17] [16].

Mivel a célom egy olyan módszer vizsgálata volt, mely automatikusan generálható egy nyelvtani leírásból, általános a teljes CF nyelvosztályra, mégis hatékony tud lenni fejlesztőkörnyezetekbe építve, a választásom erre az algoritmusra esett. A hatékonyságon kívül egy parsernek – bár erre a dolgozat nem tér ki – általában hibatűrőnek is kell lennie, a GLR – és ISGLR – erre több módszerrel is fel van készítve [7] [17].

3 Kontribúció

3.1 A GSS műveletei

A szakirodalom – bár a GSS alapötletét tárgyalja [15] [14] –, nem ismerteti a GSS műveleteinek részleteit, nem ad semmiféle referenciaimplementációt. Bár ezek nem túlzottan bonyolultak, könnyen lehet nehezen javítható, helyesnek vélt, de mégis hibás implementációt készíteni. Emiatt ebben a fejezetben egy lehetséges implementációt adok a műveletek ismertetése mellett: i) a bemenet szinkronizációija, ii) shift művelet, iii) reduce művelet.

Bemenet szinkronizáció

Azonos állapotok összevonása csakis akkor történhet meg két ág között, ha azok a bemenetben ugyanott tartanak, vagyis ugyanaz a terminális következik feldolgozásra. Mivel a bemenetben egyedül a *Shift* műveletek haladnak előre, egy nagyon egyszerű szabállyal szinkronban tartható az összes fej a veremben: minden lehetséges *Reduce* műveletet végrehajtunk a *Shift*-ek előtt.

Érdemes bevezetni egy új fogalmat, melyet *Reduce-Shift-ciklusnak* hívok a továbbiakban. Egy *Reduce-Shift-ciklus* az a folyamat, mikor az összes lehetséges Reduce műveletet végrehajtjuk, majd minden fejen végrehajtjuk az ott következő Shift műveletet.

Shift

A Shift művelet bemenete a fej, amire a művelet végrehajtandó, illetve a terminális, ami a bemenetből kiolvasásra került. A művelettől azt várjuk, hogy egy szimbólum és az új állapot csomópontja kerüljön a megadott fej fölé [15].

Abban az esetben, ha a jelenlegi Reduce-Shift-ciklusban már ugyanilyen állapot Shift-elve lett, az állapot újrafelhasználható, a Shift csupán hozzáköti a régi fejet ehhez az új állapot csomóponthoz. Ha ilyen állapot még nem lett Shift-elve a jelenlegi Reduce-Shift-ciklusban, akkor új állapot csomópontot kell neki létrehozni.

Egy állapot újrafelhasználása esetén a szimbólum csomópontja is újrafelhasználható, ha az már szerepel az állapot csomópontját megelőző csomópontok között. A teljes folyamatot a 2. algoritmus írja le.

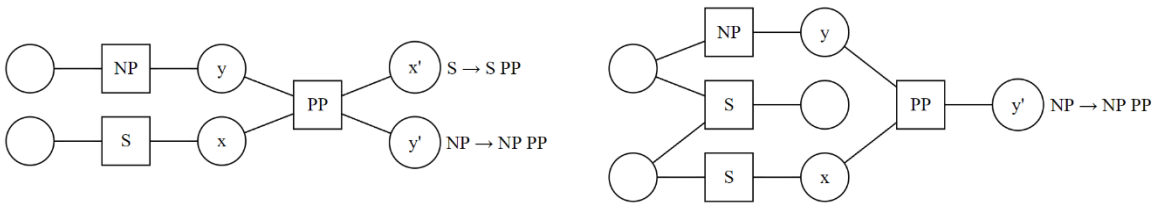
Bemenet: A fej csomópont, amin a művelet végrehajtandó (V), a bemenetről olvasott terminális (T) és az állapot, amibe kerül az új fej (S).

Kimenet: Az újonnan létrehozott állapot csomópont, vagy *NULL* érték, ha egy létező került újrafelhasználásra.

if jelenlegi Reduce-Shift-ciklusban már van V_S Shift-elve, melynek állapota S:
    if V_S-nek van olyan korábbi V_T szomszédja, melynek szimbóluma T:
        kössük V_T-hez V-t
    else:
        hozzunk létre egy új V'_T csomópontot, melynek szimbóluma T
        kössük V'_T-hez V-t, illetve V_S-hez V'_T-t
    return *NULL*
else:
    hozzunk létre egy új V'_S csomópontot, melynek állapota S
    hozzunk létre egy új V'_T csomópontot, melynek szimbóluma T
    kössük V'_T-hez V-t, illetve V'_S -höz V'_T-t
    V'_S fejnek fog számítani a következő Reduce-Shift-ciklusban
    return V'_S

2. algoritmus: A Shift algoritmus egy adott fejre, terminálisra és új állapotra.

Felmerülhet a kérdés, hogy miért csak újrafelhasznált állapotok esetén engedjük meg a szimbólumok újrafelhasználását, hiszen ezzel is spórolhatnánk a csomópontok mennyiségén, sőt, az algoritmus is sokkal egyszerűbb lenne. A problémát ezzel az ötlettel az 5. ábra szemlélteti.



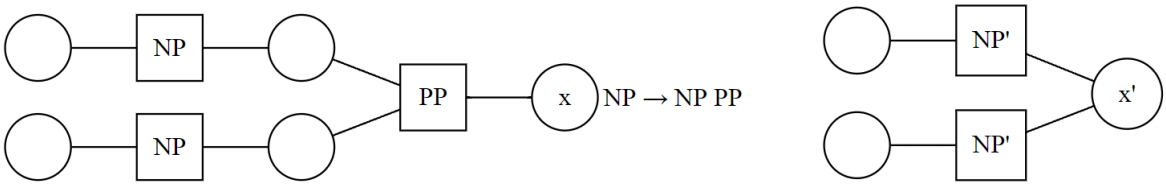
5. ábra: Helytelen redukció a szimbólumok újrafelhasználása miatt.

Mind az x és y-t követő szimbólum egy PP, azonban az x utáni állapot az x', az y utáni pedig az y' lenne. Ez az információ azonban elvész a közös PP szimbólum miatt. Míg

az x' csomópontból a $S \rightarrow S\ PP$, addig az y' csomópontból az $NP \rightarrow NP\ PP$ redukció lenne érvényes. Az ábra jobb oldalán azonban látható, hogy a PP szimbólum mindkét ágán megtörtént az $S \rightarrow S\ PP$ redukció, hiszen elveszett a pontos szomszédossági információ. Emiatt egy helytelen, $S \rightarrow S\ NP$ redukció is megtörtént, ami nem része a nyelvtannak.

Reduce

A Reduce művelet egy fejből kiindulva összegyűjti a redukcióhoz szükséges szimbólumokat, azokból felépíti a részfát, és ezt az új szimbólummal együtt a verembe helyezi. Ennek a nehézsége, hogy a szimbólumok összegyűjtése közben a verem szétágazhat, ekkor ugyanis az összes ágon, párhuzamosan kell folytatni a műveletet, és az összes ágra kell helyezni a felépített részfákat [15]. Egy kétfele ágazó redukciót ábrázol a 6. ábra.



6. ábra: Redukció több ágon.

Az algoritmus maga két fő lépésből áll: Először az összes útvonalon leszedi a megfelelő mennyiségű szimbólumot a fejből elindulva, majd a Shift-hez nagyon hasonló művelettel a veremre helyezi az épített fákat.

Készítsük el Shift' algoritmust a 2. algoritmus alapján, azonban az újrafelhasználás feltétele – ami az első sorban található – módosuljon az alábbira:

jelenlegi Reduce-Shift-ciklusban már van V_S ***Shift'****-elve, melynek állapota S*

Tehát az algoritmus csupán a Shift'-tel felhelyezett csomópontok között használhat fel újra csomópontokat. Erre azért van szükség, mert a Reduce műveletek között is történhet ugyan Reduce-Shift-cikluson belüli újrafelhasználás, ez nem keveredhet a Shift-ek által épített fejekkel, hiszen azok a veremnek már egy későbbi időpillanatára vonatkoznak.

Emellett a Shift' már nemcsak terminálisokkal, hanem részfákkal foglalkozik, így T bemenetének a helyén már részfa is állhat. Mivel a gyakorlatban a terminálisokat is már egy egyetlen elemű részfaként szokás a veremben tárolni, ez nem okoz jelentős változtatást.

A teljes Reduce folyamatát a 3. algoritmus írja le, az előbb definiált Shift' segítségével.

Bemenet: A fej, amin a művelet végrehajtandó (V) és a produkciós szabály, amivel redukálunk ($A \rightarrow \alpha$).

Kimenet: Az újonnan létrehozott állapot csomópont, vagy *NULL* érték, ha egy létező került újrafelhasználásra.

```
1  Gyűjtsük V-ből az összes |α| hosszú utat egy L listába
2  foreach a út az L listában:
3      if Goto(a végpontja, A) egy létező S állapot:
4          legyen T az a elemeiből készített fa (mely az A → α produkció eredménye)
5          legyen V' a Shift'(a végpontja, T, S) hívás eredménye
6          if V' nem NULL érték:
7              derítsük fel az összes végrehajtandó Shift és Reduce műveletet V'-n
```

3. algoritmus: A Reduce algoritmus egy adott fejre és produkciós szabályra.

3.2 A keretrendszer

A parserek összevetéséhez egy saját, nyílt forráskódú keretrendszert készítettem[3]. A keretrendszer célja az volt, hogy különböző konfigurációkban lehessen megvizsgálni a parserek statikus, illetve dinamikus tulajdonságait. Bár különböző eszközök eddig is léteztek LR parserek vizsgálatára [18] [19], ezek általában igen hiányosak: nem támogatnak inkrementális elemzést, GSS-t vagy bármiféle nemdeterminisztikus viselkedés szimulálását, vagy nem ábrázolják a folyamatot. A komolyabb eszközök, melyek parser kód generálására is képesek – mint a *Yacc*, *Bison* [20] vagy *Lemon* [21] – pedig csupán a konfliktusokra hívják fel a figyelmet, semmiféle vizualizációval nem segítenek.

A keretrendszer támogatja az LR(0), SLR, LALR és LR(1) táblák generálását CF nyelvtanból, mely lehet a szokásos produkciós szabályok specifikálása, vagy EBNF jelölés.

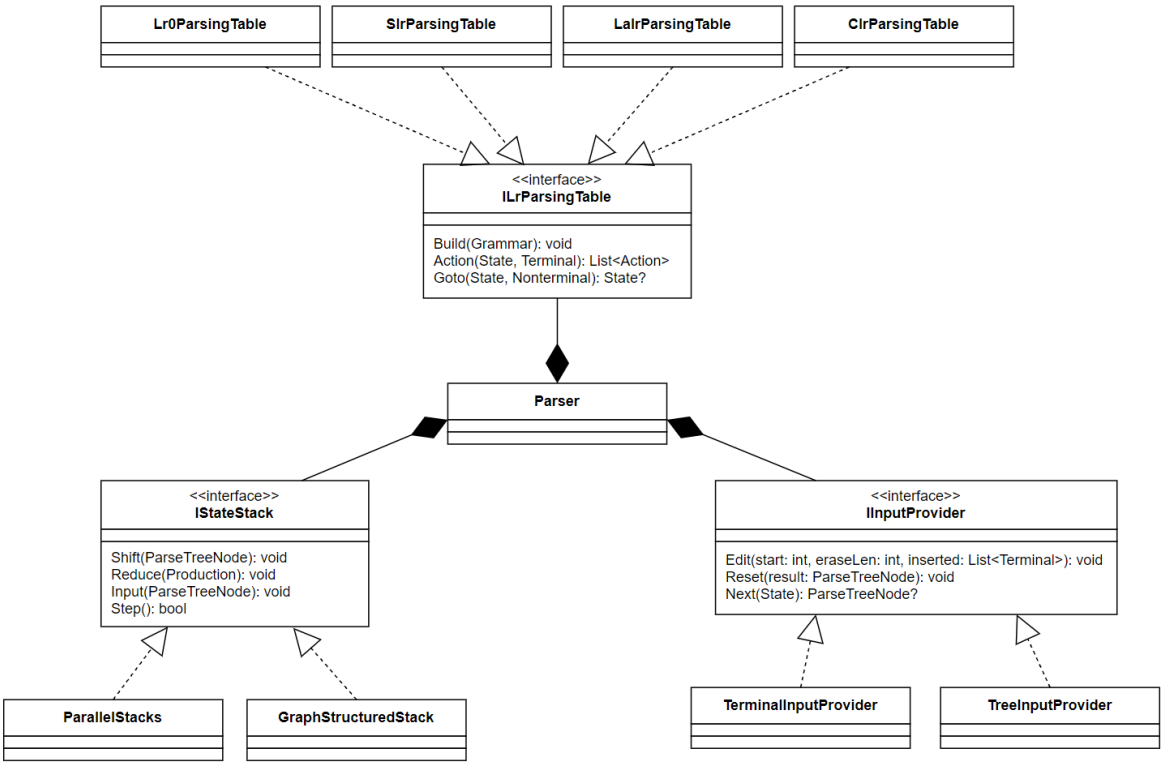
[3] A projekt a https://github.com/LanguageDev/Yoakke címen található. Ez egy általam fejlesztett, általános célú fordítóprogram keretrendszer, ami az ide tartozó munkámon kívül rengeteg mást is tartalmaz. Mivel itt már rengeteg segédeszközt fejlesztettem, ami hasznos volt a munkám során, a keretrendszer maga is ide került (jelenleg a „*grammar-library*" ágon).

Képes LR parser futtatására – akár műveletenként végigkövetve –, továbbá GLR elemzést párhuzamos vermek vagy GSS segítségével. Az elemzési mód változtatható inkrementális és teljes elemzés között. Az elemzési folyamatot ábrákkal képes illusztrálni.

3.2.1 Az architektúra

A keretrendszert lazán csatolt komponensek alkotják, melyek mind egy-egy aspektusát engedik külön vizsgálni, illetve lecserélni a parsernek. A komponenseket és relációjukat a 7. ábra mutatja, mely egy egyszerűsített osztálydiagramja a legfontosabb elemeknek. Az ábra közepén a parser található (*Parser* osztály), mely 3 komponensből tevődik össze:

- Parser tábla (*ILrParsingTable* interfész): képes legenerálni a nyelvtanhoz a megfelelő LR *Action* és *Goto* átmeneteket. Emellett vezeti a parsert az elemzési folyamat közben (2.1. kódrészlet). A dolgozatban említett négy módszer van implementálva (LR(0), SLR, LALR, LR(1)), de szabadon tovább bővíthető.
- Veremkezelő (*IStateStack* interfész): elemzés közben kezeli a számítási vermet. A párhuzamosan vezetett vermek módszere, illetve a *Graph Structured Stack* van implementálva. Szintén szabadon bővíthető, például a köztes megoldásként nevezett *Tree Structured Stack* [14] vizsgálatához izgalmas lehet.
- Bemenet kezelő (*IInputProvider* interfész): Az olvasandó bemenetet biztosítja a parsernek, illetve a bemenet szerkesztéséért felelős. Két stratégia van implementálva. Az egyik (*TerminalInputProvider*) mindig a teljes bemenetet adja át a parsernek, ami neminkrementális elemzéshez vezet. A másik (*TreeInputProvider*) az előzőleg elemzett fából az újrahasználható részeket adja a parsernek, mely inkrementális elemzést eredményez.



7. ábra: Egyszerűsített osztálydiagram a keretrendszer legfontosabb elemeiről.

3.2.2 Elemzési fa újrafelhasználása

Az inkrementális elemzés hatékony és helyes működéséhez az egyik legfontosabb lépés, hogy a korábbi eredményből a lehető legtöbbet megtartsuk egy változtatás után, de amit szükséges, dobjuk el, a beillesztett elemek pedig kerüljenek a beszúrás helyére.

A változtatást a keretrendszer egy közös művelettel jellemzi, mely az *Edit(startPosition, eraseLength, inserted)*. Ez a *startPosition* helytől *eraseLength* darab terminálist töröl, majd szintén a *startPosition*-tól kezdve beszúrja az *inserted* terminálisokat. Ezzel az összes gyakori szerkesztési művelet megvalósítható, például, ha csupán beszúrunk, akkor *eraseLength=0*, ha pedig csak törlünk, akkor az *inserted* üres.

Az inkrementális elemzésre alkalmas bemenetkezelő osztály (*TreeInputProvider* osztály, 7. ábra) alapötlete, hogy a bemenetet nem terminálisokként, hanem részfákként tárolja. Egy változtatás hatására megkeresi az összes érintett részfát, és megjelöli azokat nem újrafelhasználhatónak. A törölt intervallumban levő levélelemek törlésre kerülnek – hiszen azok terminálisokat reprezentálnak –, a beillesztett terminálisok pedig a megfelelő

csomópont levélelemeiként beszúrásra kerülnek. A teljes *UpdateTree* folyamatot a 4. algoritmus írja le.

Bemenet: A szerkesztés kezdete terminálisokban számolva a bemenet elejétől (S), a törölt terminálisok száma (E), illetve a beillesztett terminálisok (I). Opcionális paraméter a jelenleg feldolgozandó fa csomópont (N), mely
Kimenet: Igaz érték, ha I beillesztésre került, különben hamis.

1 Jelöljük meg N-t nem újrafelhasználhatónak
2 **foreach** N' gyerek eleme N-nek:
3 **if** N' intervalluma $\cap\ [S;\ S+E)\ \neq \emptyset$:
4 **if** N' levélelem:
5 Töröljük N'-t N gyerekei közül
6 **else**:
7 Rekurzívan hívjuk *UpdateTree*-t új $N = N'$ paraméterrel
8 **if** a rekurzív hívások mind hamis értékkel tértek vissza:
9 Illesszük be I-t N gyerekei közé a pozíciók sorrendjének megtartásával

4. algoritmus: A fa frissítése szerkesztésre (UpdateTree folyamat).

A fa frissítése után még egy fontos lépés, hogy hogyan adjuk vissza a parsernek a megfelelő, újrafelhasználható részeket. Ha egy részfa nem újrafelhasználhatónak lett megjelölve, már csak annak gyerek elemei kerülhetnek felhasználásra. Ha egy részfa nem lett megjelölve, de a parser állapota nem egyezik meg azzal az állapottal, ami a részfa elemzésekor volt, szintén csak annak gyerek elemei kerülnek újrafelhasználásra [16]. A teljes *NextTree* folyamatot az 5. algoritmus mutatja be, melyet a parser használ a következő bemenet kiolvasására Shift esetén.

Bemenet: A sor, amiben kezdetben a korábbi részfa *UpdateTree*-vel módosított változata van (Q), valamint a parser jelenlegi állapota (S).
Kimenet: A következő részfa, mely bemenete a parsernek.

1 Vegyük ki a Q elején levő elemet, tegyük N-be
2 **while** N nem levélelem vagy N nem újrafelhasználhatónak van jelölve vagy N felhelyezése előtti parser állapot nem S volt:
3 Tegyük Q elejére N gyerek elemeit
4 **return** N

5. algoritmus: Következő részfa lekérése a parsernek (NextTree folyamat).

3.2.3 Elemzés vizualizációja

A legtöbb eszköznél hiányoltam bármiféle vizualizációt az elemzés folyamatáról – illetve inkrementális elemzéshez egyáltalán nem volt ilyesmi. Emellett hibakeresésnél sokszor hasznos volt grafikusan látni a különböző komponensek állapotait, mivel azok szövegesen igen nehezen értelmezhetőek.

A következőkben a keretrendszer vizualizációs képességeit mutatom be.

Step-by-step végrehajtás

A keretrendszer a legtöbb műveletet lépésekre bontva hajtja végre, minden egyes lépés után visszaadva az irányítást a hívónak. Ez nagyban megkönnyíti a folyamat közbeni, akár lépésenkénti vizualizációk készítését. Léptethető a parser bemenetenként, a vermek műveletenként, vagy akár az inkrementálisan feldolgozott fa lebontásonként. Ez egy igen hasznos funkciónak bizonyult hibakereséshez. Ilyen step-by-step futtatás során készült például a 10. ábra és 11. ábra (melyek két közvetlenül egymásután történő lépést ábrázolnak), valamint egy szerkesztési lépés után, de az újra elemzés előtt készült a 12. ábra.

LR táblák ábrázolás

Bár a legtöbb eszköz ezt már támogatta, szükségem volt az LR táblák kirajzolására, főleg a helyesség ellenőrzése miatt. Ehhez egy egyszerű HTML dokumentumot generálok,

mely a szakirodalomban megszokott elrendezésben mutatja az *Action* és *Goto* táblákat. Például a 6. nyelvtan LR(0) táblájának a keretrendszerrel való ábrázolását a 8. ábra mutatja.

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid number$$

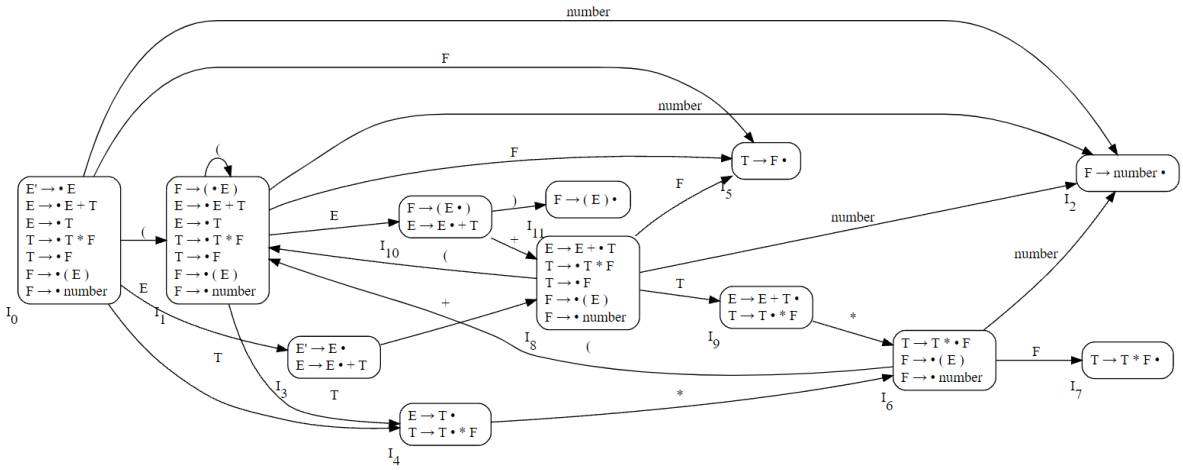
6. nyelvtan: Egyszerű, kétműveletes és zárójelezhető kifejezésnyelvtan.

	Action						Goto				
State	$	+	*	(	)	number	S	E	T	F	E'
0				s(1)		s(2)		3	4	5	
1				s(1)		s(2)		10	4	5	
2	r(F → number)	r(F → number)	r(F → number)	r(F → number)	r(F → number)	r(F → number)					
3	a	s(8)									
4	r(E → T)	r(E → T)	s(6) r(E → T)	r(E → T)	r(E → T)	r(E → T)					
5	r(T → F)	r(T → F)	r(T → F)	r(T → F)	r(T → F)	r(T → F)					
6				s(1)		s(2)				7	
7	r(T → T * F)	r(T → T * F)	r(T → T * F)	r(T → T * F)	r(T → T * F)	r(T → T * F)					
8				s(1)		s(2)			9	5	
9	r(E → E + T)	r(E → E + T)	s(6) r(E → E + T)	r(E → E + T)	r(E → E + T)	r(E → E + T)					
10		s(8)			s(11)						
11	r(F → (E))	r(F → (E))	r(F → (E))	r(F → (E))	r(F → (E))	r(F → (E))					

8. ábra: A 6. nyelvtan LR(0) Action és Goto táblája.

LR állapotgép ábrázolása

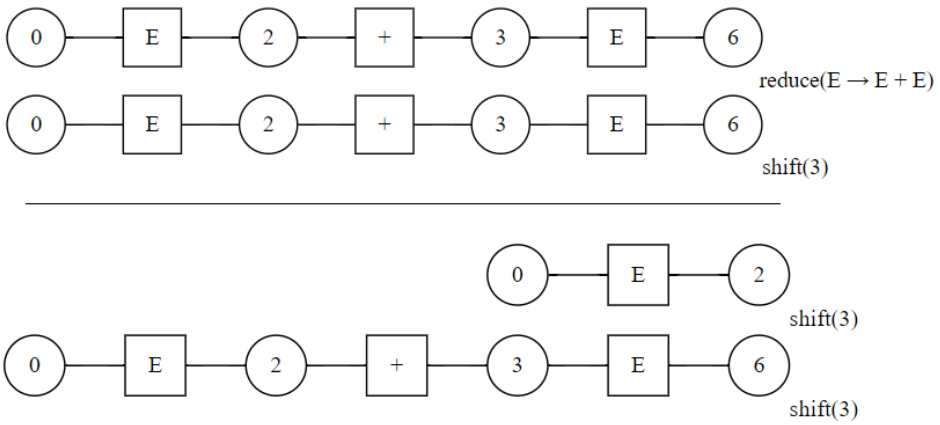
Kevesebb eszköz támogatta az LR állapotgépek kirajzolását, ami sokkal inkább szemlélteti az LR parserek állapotai közti összefüggéseket. Mivel ez alapvetően egy irányított gráf, kimenet formátumának a DOT gráfleíró nyelvet választottam, mely utána vizualizálható például a *Graphviz* nevű szoftverrel. A 6. nyelvtan LR(0) állapotgépét mutatja DOT formátumban a 9. ábra.



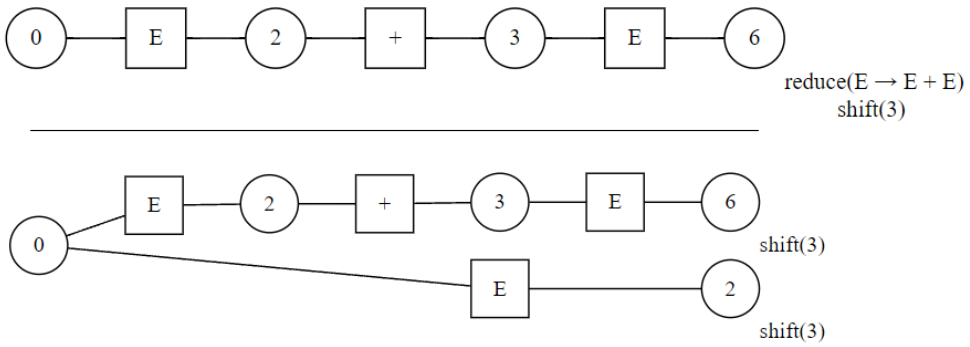
9. ábra: A 6. nyelvtan LR(0)/SLR állapotgépe DOT formátumban ábrázolva.

Verem ábrázolása

Mind a párhuzamos vermeknél, mind a GSS-nél hasznos volt az állapot ábrázolása, akár bizonyos hátralevő műveleteket ábrázolva. Ez könnyítette meg a folyamatok végigkövetését, a legjobb segédeszköz volt a parser dinamikus működésének a megfigyelésére. Az ábrázolások szintén DOT formátumban kerülnek előállításra. Ugyanazt a lépést ábrázolja párhuzamos vermekkel a 10. ábra, GSS-sel pedig a 11. ábra.



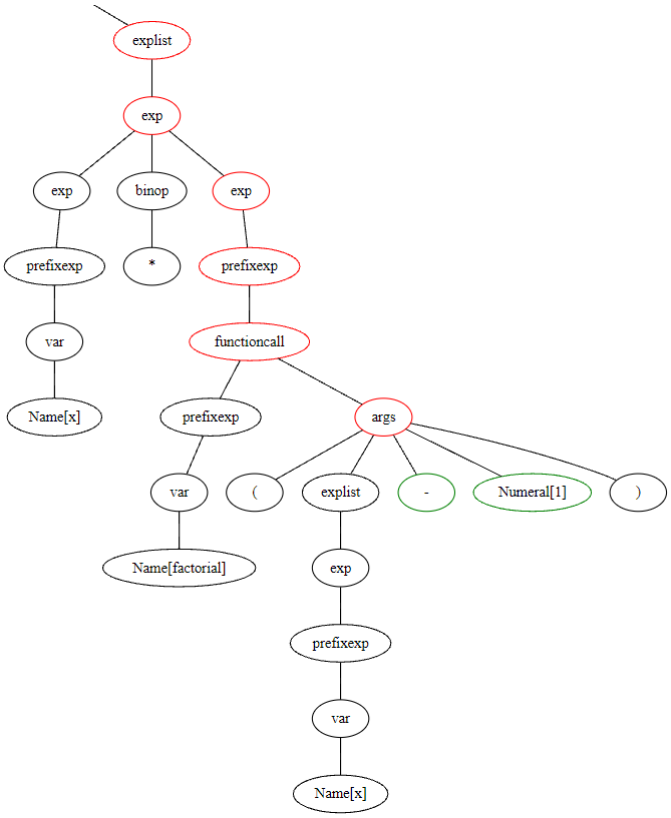
10. ábra: Párhuzamos vermek redukciós lépés előtt és után.



11. ábra: GSS redukciós lépés előtt és után.

Elemzési fa ábrázolása

Az elemzési fát már pusztán azért érdemes ábrázolni, mivel ez az elemzés végeredménye, a kívánt struktúra. Önmagában ez azonban kevés, főleg inkrementális elemzésnél, ahol szóba jön az is, hogy mely elemeket érintett a változás, melyek nem újrafelhasználhatók, illetve mely terminálisok kerültek beillesztésre. A keretrendszer a bemeneten való változtatás után piros színnel emeli ki a nem újrafelhasználható, zöld színnel pedig az újonnan beillesztett elemeket, mint például azt a 12. ábra mutatja.



12. ábra: Lua kód szintaxisfája, melyen megjelölésre kerültek az érintett elemek egy szerkesztés után.

3.3 Összehasonlítások alapja

Az eredmények kiértékeléséhez érdemes definiálnunk, hogy milyen mérést végzünk, milyen mérőszámokat használunk a különböző aspektusok vizsgálatához, illetve megközelítőleg milyen eredményeket várunk. Több izgalmas szempontból vizsgálhatók a parserek, így azt is érdemes tisztázni, mit és miért hasonlítunk össze.

Minden mérőszám meghatározása, illetve mérés a már említett, saját keretrendszerben történt. Ezzel elkerültem a különböző implementációk közti esetleges eltéréseket.

3.3.1 Mérőszámok

A mérés módjához és a mérőszámok meghatározásához érdemes a parser összes konfigurálható paraméterét figyelembe venni:

- Generált tábla: lehet LR(0), SLR, LALR és LR(1)
- Veremkezelési stratégia: lehet párhuzamosan vezetett vermek vagy GSS
- Korábbi eredmény újrafelhasználása: nincs (nem inkrementális), van (inkrementális)

Bizonyos paraméterekhez önmagukban is rendelhetünk mérőszámokat, melyek relevánsak lehetnek a végeredmények kiértékeléséhez, illetve a végső összehasonlításhoz. A következőkben ezeket mutatom be.

A *generált tábla* 3 statikus mérőszámot vezet be, melyek nem függenek a bemenettől: i) az állapotok számát, ii) a konfliktusos lépések számát, iii) illetve a konfliktusok számát. Utóbbi kettő között lényegi különbség, hogy az előbbi a nemdeterminisztikus lépések számát arányítja az összes lépéshez, az utóbbi egy nemdeterminisztikus lépés szétágazására utal.

A *veremkezelési stratégia* dinamikus mérőszámokat vezet be, melyet egy adott bemenet elemzéséhez lehet rendelni: i) a létrehozott elemek számát, ii) valamint a létrehozott élek számát. Bár a kettő szorosan összefügg, a két szám relációja jelentősen változhat a párhuzamos vermek és GSS veremkezelés között.

Magának az elemzésnek a lépésszámához igen kézenfekvő a két fő művelet összeszámolása: a Shift-ek és a Reduce-ok száma. Ha feltételezzük, hogy ezek költsége

parserről parserre közel azonos – ami a legtöbb esetben megengedhető –, ez egy jó mérőszám lesz a parserek sebességének összehasonlítására.

3.3.2 Az összehasonlítások célja

Bár az LR parserek között [22] és a GLR algoritmus különböző tábláival futtatott verziói között már végeztek összehasonlító elemzést [23], ebben több hiányosság is volt:

- Nem tartalmazott LALR táblával való futtatást, amire általában alapozzák a GLR parsereket. Így éppen a lényeges konklúzió nem vonható le belőle: érdemes-e a nagyobb táblájú LR(1)-es módszert használni, vagy a sokkal egyszerűbb SLR-t, vagy pedig maradjunk inkább a LALR-nál?
- Nem vizsgálta a GSS mellett a párhuzamos vermekkel való futtatást. Mivel a GSS jórészt a GLR algoritmus futtatásában kerül elő, érdemes volna a hatékonyságát alátámasztani azzal, hogy összevetjük a naiv, párhuzamos vermek módszerével.

Továbbá nincs olyan összehasonlítás, ami mindezeket az inkrementális GLR elemzéssel veti össze, mely szerintem egy igen hasznos módja volna a módszer hatékonyságának bemutatásához. Ezeket mind célom pótolni a saját méréseimben.

3.3.3 Várt eredmények

A mérései eredményektől két igen fontos konklúziót vártam el:

- A GSS egy hatékonyabb reprezentáció, mint a párhuzamos vermek. Emellett sejtésem szerint az, hogy mennyivel hatékonyabb, függ a nemdeterminisztikus lépések számától. Egy bonyolultabb LR módszerrel– például SLR az LR(0) helyett –, kevesebb konfliktussal ez a különbség szerintem csökkenthető. A mérések később ezt igazolták.
- Az ISGLR által bevezetett inkrementális módszer hatékony, és rengeteg műveletet spórolnak meg a kisebb szerkesztések hatására. Ezt a mérések később szintén igazolták. Meglepetésemre a Reduce műveleteken sokkal többet spóroltak, mint a Shift műveleteken, mely okára külön is kitérek az eredmények kiértékelésekor.

4 Mérési eredmények

Hogy a mérés gyakorlatibb legyen, egy létező programozási nyelv, a Lua 5.3 nyelvtanát választottam (Függelék: A Lua 5.3 hivatalos nyelvtana). A választás azért erre a nyelvre esett, mert a hivatalos nyelvtan teljesen publikus, a nyelv maga és a nyelvtana viszonylag egyszerű, de mégis bőven tartalmaz nemdeterminizmust – főleg az operátor precedenciák hiánya miatt.

4.1 A táblák összehasonlítása

A Lua 5.3 nyelvtan által generált négy táblázat adatait a 4.1. táblázat mutatja.

	Állapotok	**Konfliktusos lépések**	**Konfliktusok**
LR(0)	206	685	686
SLR	206	69	69
LALR	206	44	44
LR(1)	1628	900	900

4.1. táblázat: Az LR táblák mérőszámai.

Jól látható, hogy – amint arra a 2.4 fejezet utalt –, az LR(0), SLR és LALR módszerek ugyanannyi állapotból állnak, viszont a konfliktusok száma jelentősen csökken a bonyolultabb módszer irányába. A kezdeti LR(0) kimagasló konfliktusszáma nem meglepő, hiszen minden produkciós szabály felismerésekor minden egyes terminálisra Reduce műveletet ír elő (részletesebben a 2.4.3 fejezet magyarázza), ezzel rengeteg Shift-Reduce konfliktust okozva. Az SLR ezzel szemben majdnem megtizedelte a konfliktusok számát. Figyelembe véve az SLR egyszerűségét, nyugodtan kijelenthető, hogy az LR(0)-val szemben mindig érdemes az SLR-t választani. Az SLR és LALR között már nincsenek ilyen nagyságrendi különbségek, jelen esetben az LALR az SLR-hez képest megszabadult a konfliktusok 37%-ától. Egyszerűbb nyelvtanoknál ez akár elhanyagolható is lehet. Mivel az LALR-t igen bonyolult egyszerre hatékonyan és helyesen megvalósítani [2], ha a nyelvtan elég egyszerű és kevés többlet konfliktust okoz az SLR a LALR-hez képest, érdemes lehet az SLR-t választani. Kész, tesztelt implementációk esetén azonban nincs miért az SLR-t választani az LALR felett.

Az LR(1) sokkal több állapotból áll, mint a korábbi módszerek. Ez várható volt, hiszen az LR(1) automata elemei már egy terminálist előretekintenek a nyelvtanban, és a legtöbb produkciós szabályt általában több, mint egyféle terminális követheti.

Felmerülhet a kérdés, hogy ha az LR(1) nagyobb nyelvosztályt ismer fel, mint a korábbi módszerek, akkor miért van több konfliktusa még az LR(0)-nál is? A válasz nem magától értetődő. Az LR(1) sokkal nagyobb állapotteret fed le, az LALR konfliktusmentes átmenetei ugyanúgy megjelennek az LR(1)-ben is, viszont az LR(1) rengeteg olyan állapotot modellez, melyeket például az LALR egyáltalán nem. A rengeteg új konfliktus ezekből a nem modellezett állapotokból keletkezett. A későbbi mérések is ezt igazolják majd.

4.2 Veremkezelési stratégiák összehasonlítása

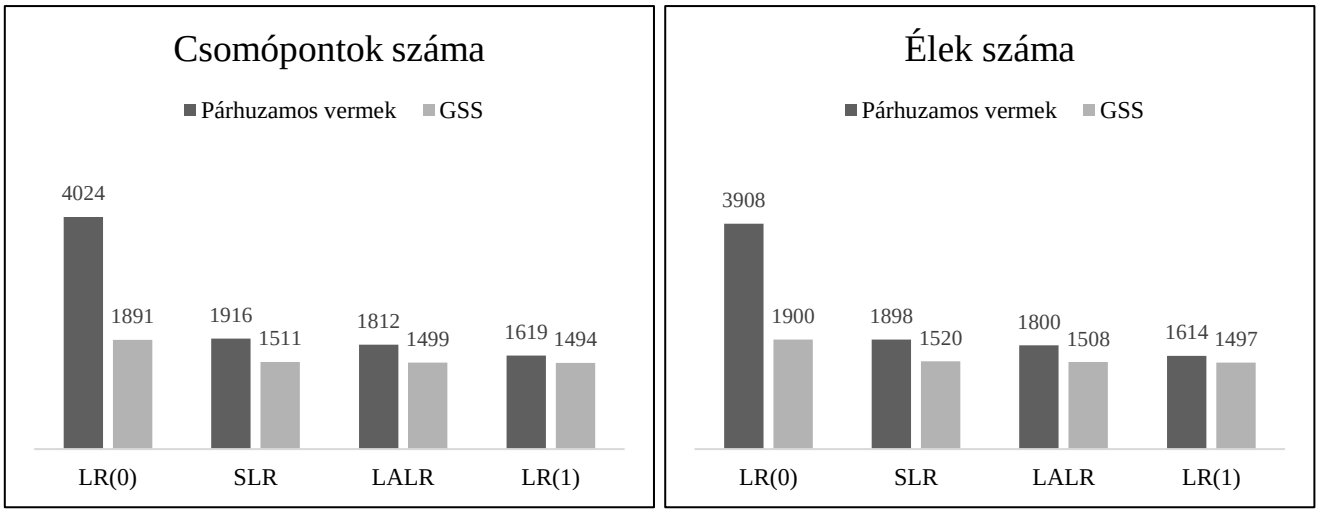
Bár a GSS intuitívan egy hatékonyabb adatszerkezet, mint a párhuzamos vermek vezetése, gyakorlatban előforduló adatokkal – mint például egy konkrét programkód – nincsenek összehasonlító mérések. Emiatt érdemes kitekintést tenni arra, hogy egy elemzés során mekkora a két stratégia közti különbség. A 4.2. táblázat a két stratégia által használt csomópontokat, a 4.3. táblázat pedig a használt éleket ábrázolja a különböző LR táblák viszonylatában, a függelékben található Lua kód elemzése közben. A méréseket grafikon formájában is megjeleníti a 13. ábra.

	LR(0)	**SLR**	**LALR**	**LR(1)**
Párhuzamos vermek	4024	1916	1812	1619
GSS	1891	1511	1499	1494

4.2. táblázat: Lua kód elemzése közben keletkezett verem csomópontok.

	LR(0)	**SLR**	**LALR**	**LR(1)**
Párhuzamos vermek	3908	1898	1800	1614
GSS	1900	1520	1508	1497

4.3. táblázat: Lua kód elemzése közben keletkezett verem élek.



13. ábra: A csomópontok és élek száma LR tábla módszerekre lebontva grafikonon.

Igen jól látszik a korreláció a nemdeterminisztikus lépések aránya, és a GSS nyújtotta előny között. Az LR(0) esetén – mely nagyon sok nemdeterminisztikus lépést tartalmaz – sokkal kisebb helyet foglal el, mint a párhuzamos vermek. Ez az előny azonban rohamosan csökken az egyre kevésbé konfliktusos módszerek esetén. Ismét jól látható, hogy a legnagyobb ugrást az LR(0) és SLR közti váltás okozta, míg az SLR és LALR közti különbség sokkal inkább elhanyagolható. Egy rendkívül nemdeterminisztikus nyelvtan esetén – mint például egy természetes nyelv – tehát mindenképp érdemes GSS-t használni, hiszen az LR módszerek a nemdeterminisztikus nyelvtan esetén mindig konfliktusosak lesznek. Egy konfliktusos – de alapvetően determinisztikus – nyelvnél jó eséllyel a párhuzamos vermek is megfelelnek, ha legalább SLR-t használunk.

Megfigyelhető még, hogy míg a párhuzamos vermek esetében a csomópontok száma dominál az élek száma fölött, ez GSS esetén éppen fordítva van, az élek száma mindig több a csomópontok számánál. Ez arra utal, hogy a GSS sokszor képes spórolni az új csomópontokon, és csupán két létező csomópont között húz be egy élet.

4.3 Inkrementális elemzés műveletszáma

Az inkrementális elemzés hatékonyságának megállapításához érdemes megnézni, hogy egy-egy apróbb változásra – például egy paraméter beillesztése vagy név átírása – hány művelettel képesek előállítani az eredményt a teljes elemzéshez képest. A méréshez a bemenetet a Lua hivatalos példakódjaiból választottam, melyen némileg változtattam

(Függelék: Markov láncok Lua-ban). A teljes elemzés után egy *Name* terminális kerül beszúrásra a 2. sor függvényhívásába, így a 2. sorban az `io.read(STDIN)` szerepel. Ez egy olyan valós, kisebb módosítást hivatott utánozni, amilyet a felhasználó folyamatosan végez a forráskódon, és amire a fejlesztőkörnyezetnek minél gyorsabban reagálnia kell.

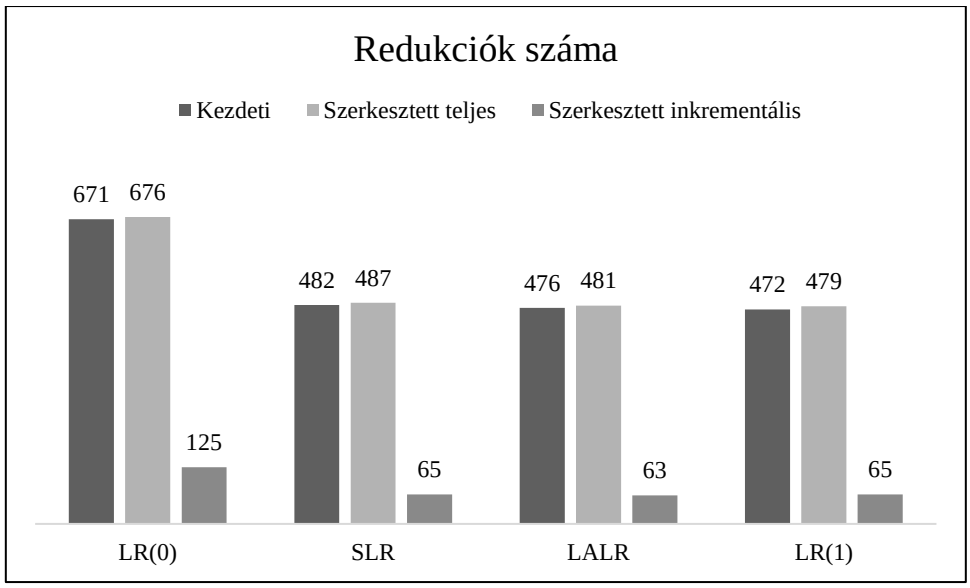
A 4.4. táblázat mutatja, hány művelet szükséges a kezdeti teljes elemzéshez, a változtatott kód inkrementális elemzés nélkül, illetve inkrementális elemzéssel a különböző LR táblák használatával, GSS mellett.

		LR(0)	**SLR**	**LALR**	**LR(1)**
Kezdeti	Shift	268	268	268	273
	Reduce	671	482	476	472
Szerkesztett teljes	Shift	271	271	271	276
	Reduce	676	487	481	479
Szerkesztett inkrementális	Shift	75	75	75	75
	Reduce	125	65	63	65

4.4. táblázat: Kezdeti, szerkeszett teljes, illetve inkrementális elemzések lépésszáma.

Mivel egy Shift művelet körülbelül megfeleltethető egy bemeneti elem feldolgozásának, egy jó kiindulási alapot nyújt az összehasonlításhoz. Kezdeti, illetve a szerkesztett teljes elemzésnél ezek száma éppen a terminálisok száma, ez magyarázza, hogy a beillesztés után ez a szám megnőtt. Inkrementális elemzésre ez a szám a feldolgozott részfák száma, mely jól mutatja, hogy jelentős számú részfát sikerült egy az egyben újra felhasználni a korábbi bemenetből.

A Reduce műveletek némileg izgalmasabbak. Számuk egy új faelem létrehozásával egyeztethető. Az LR konfliktusok miatt ez a szám sokkal inkább változik az LR módszerek között is, mint a Shift-ek száma. Ezt a különbséget a 14. ábra hivatott szemléltetni.



14. ábra: Redukciók száma a kezdeti, szerkesztett teljes, illetve szerkesztett inkrementális elemzésnél.

Ismét szembetűnő, hogy a bonyolultabb LR módszerek felé haladva csökken a szükséges Reduce műveletek száma, ami a konfliktusok számának csökkenésére utal.

Az inkrementális módszer itt még inkább jelentős javítást mutat, mint a Shift műveletek esetén, minden LR módszer esetén. Ez részben annak is köszönhető, hogy a nyelvtanban igen sok az olyan produkciós szabály, melynek jobb oldalán egyetlen szimbólum található, ezzel sok egyelemű redukciót előidézve az elemzés során. Emiatt egy részfa újrafelhasználása sokszor csak egyetlen Shift-et spórol meg, de jelenthet akár 4-5 Reduce megspórolást is. Egy olyan nyelvtanban, ahol kevesebb ilyen szabály van, tehát minden Reduce művelet több szimbólum redukcióját jelenti, a Shift és Reduce műveletek száma valószínűleg sokkal közelebb volna egymáshoz.

Elmondható, hogy a módszer képes igen hatékonyan elvégezni a szintaktikai fán való módosítást, és újraelemzéskor képes felhasználni annak részeit, jelentősen lecsökkentve az elvégzendő munkát és memória allokációkat.

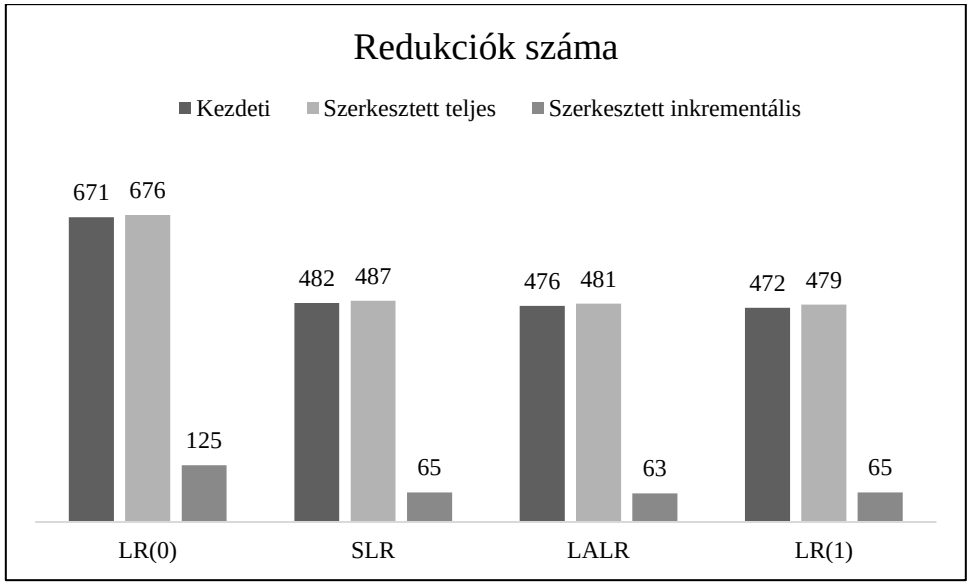
4.3.1 Inkrementális elemzés nagyobb változásra

Mint látható volt, az algoritmus jól teljesít olyan változásokra, melyeket a felhasználó kód gépelése közben ejthet, például beszúr egy paramétert vagy átír egy azonosítót. Felmerülhet azonban, hogy előnnyel jár-e az inkrementális algoritmus akkor is, ha egy sokkal nagyobb változtatást hajtunk végre a kódon? Ennek vizsgálataképp az előző kísérletet

elvégeztem úgy is, hogy a kódba nem beillesztettem, hanem kitöröltem a teljes *prefix* nevű függvényt. A – korábbival egyező formátumban a – műveletszámokat a 4.5. táblázat, a Reduce műveleteket LR módszerre lebontva pedig a 15. ábra mutatja.

		LR(0)	**SLR**	**LALR**	**LR(1)**
Kezdeti	Shift	268	268	268	273
	Reduce	671	482	476	472
Szerkesztett teljes	Shift	252	252	252	257
	Reduce	639	459	453	449
Szerkesztett inkrementális	Shift	51	51	51	51
	Reduce	83	39	37	37

4.5. táblázat: Kezdeti, szerkeszett teljes, illetve inkrementális elemzések lépésszáma.



15. ábra: Redukciók száma a kezdeti, szerkesztett teljes, illetve szerkesztett inkrementális elemzésnél.

Igen jól látszik a műveletek csökkenéséből, hogy az algoritmus nagyobb változtatásokra is képes újrafelhasználni a szintaxisfa jelentős részeit. Bizonyára lehetséges olyan „gonosz” szerkesztési műveletet előállítani, ahol a fa teljes egészében értelmét veszti – például a fa összes csomópontját érinti –, de a való életben az ilyenek valószínűleg ritkábban fordulnak elő.

4.3.2 Inkrementális elemzés párhuzamos vermekkel?

Felmerülhet a kérdés, hogy miért nem végeztem méréseket a párhuzamos veremkezelési stratégiával. A válasz igen egyszerű: az ISGLR által prezentált inkrementális algoritmus nem alkalmas rá. Mint arról a bevezetőben szó volt, az ISGLR csakis olyan részeredményeket képes újrafelhasználni, amelyek determinisztikus elemzés közben keletkeztek – tehát a veremnek egyetlen fej állapota van [16].

Mivel a párhuzamos vermek sosem egyesítik az állapotaikat, ez már egyetlen nemdeterminisztikus lépés esetén azt jelenti, hogy a további eredmények mind nemdeterminisztikus elemzésből származnak. Létezik egy köztes adatszerkezet a GSS és a párhuzamos vermek között, az úgynevezett *Tree Structured Stack* [14], mely a fej állapotokat igyekszik a vermek között megosztani, így valamivel alkalmasabb inkrementális elemzéshez, mint a párhuzamos vermek.

A konklúzió tehát az, hogy a párhuzamos vermek és az ISGLR természetéből adódóan nem produkáltak volna izgalmas eredményeket, gyakorlatilag a teljes neminkrementális elemzés értékeit produkálta volna.

5 Összefoglalás

A dolgozat elején röviden ismertetve lettek az LR parserek, a GLR és egy lehetséges inkrementális adaptációja, az ISGLR. A GLR hatékony működéséhez bemutattam a GSS nevű adatszerkezet egy lehetséges implementációját, melyet hiányoltam a szakirodalomból. Utána ismertettem a keretrendszert, amivel a méréseket és összehasonlításokat végeztem. Végül elvégeztem a méréseket és kiértékeltem a különböző módszerek teljesítményét. Az eredményekkel kifejezetten elégedett voltam, pontosan alátámasztották a szakirodalmat.

Mivel a mérésekkel pótoltam a számomra hiányos mérési szempontokat – az elterjedt LALR tábla összehasonlítása LR(0), SLR és LR(1) táblákkal, párhuzamos vermek és GSS összevetése, illetve az inkrementális módszer hatékonyságának vizsgálata -, a dolgozat és az abban bemutatott mérések egy jó alapként szolgálnak elemzési módszerek kiválasztására.

A dolgozat és az általa lefektetett munka alapvetően több irányba vihető tovább:

- Az egyik legcélszerűbb irány a keretrendszernek egy webes felületet készíteni, ahol a különböző parsereket lehet tanulmányozni a keretrendszer által generált ábrákkal. Szerintem ez egy igen hasznos oktatási eszköz lehetne, főleg, hogy GLR-hez vagy inkrementális elemzéshez ilyet egyáltalán nem lehet találni.
- Egy másik irányzat további komponensek implementálása lehetne. Ilyen lehetne az inkrementális lexikai analízis és kapcsolata az inkrementális elemzéssel, vagy a már említett *Tree Structured Stack*.
- Összehasonlításokat lehetne végezni teljesen más megközelítésű parserekker. Ilyen például az inkrementális *Packrat* parser [24], mely egy top-down megközelítés, és akár kézzel is könnyű implementálni.

Irodalomjegyzék

[1] B. Iván, Formális nyelvek, Typotex Elektronikus Kiadó Kft., 2005.

[2] A. V. Aho, M. S. Lam, R. Sethi és J. D. Ullman, Compilers: Principles, Techniques and Tools (Second Edition), 2006.

[3] „Bison 3.8.1 Manual," Free Software Foundation, Inc., 10 September 2021. [Online]. Available: https://www.gnu.org/software/bison/manual/bison.html.

[4] P. Blackburn és K. Striegnitz, „Union College," 29 August 2002. [Online]. Available: https://cs.union.edu/~striegnk/courses/nlp-with-prolog/html/node53.html.

[5] D. Flodin, A Comparison Between Packrat Parsing and Conventional Shift-Reduce Parsing on Real-World Grammars and Inputs, 2014.

[6] A. Sorkin és P. Donovan, LR(1) Parser Generation System: LR(1) Error Recovery, Oracles, and Generic Tokens, 2010.

[7] A. Lavie és M. Tomita, GLR* - An Efficient Noise-skipping Parsing Algorithm For Context Free Grammars, 1995.

[8] A. V. Aho és J. D. Ullman, The theory of parsing, translation, and compiling, Englewood Cliffs, NJ: Englewood Cliffs, N.J., Prentice-Hall, 1972.

[9] D. Grune és C. Jacobs, Parsing Techniques - A Practical Guide, Springer-Verlag New York, 2008.

[10] D. E.Knuth, On the translation of languages from left to right, 1965.

[11] M. D. Mickunas, R. L. Lancaster és V. B. Schneider, Transforming LR(k) Grammars to LR(1), SLR(1), and (1,1) Bounded Right-Context Grammars, 1976.

[12] M. Tomita, Generalized LR Parsing, Springer US, 1991.

[13] T. A. Wagner és S. Graham, „Incremental Analysis of Real Programming Languages," in *PLDI97: Conference on Programming Language*, Las Vegas, 1997.

[14] M. Tomita, „An efficient Augmented-Context-Free Parsing Algorithm," *Computational Linguistics,* %1. kötet13, %1. szám1-2, pp. 31-46, 01 January-June 1987.

[15] M. Tomlta, „Graph-structured Stack and Natural Language Parsing," in *ACL '88: Proceedings of the 26th annual meeting on Association for Computational Linguistics*, Buffalo, 1988.

[16] M. P. Sijm, „Incremental Scannerless Generalized LR Parsing," in *SPLASH '19: 2019 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, Athens, 2019.

[17] M. Brunsfeld, „Tree-Sitter - A new parsing system for programming tools," in *Strange Loop Conference*, St. Louis, 2018.

[18] „JSmachines - LALR(1)," [Online]. Available: http://jsmachines.sourceforge.net/machines/lalr1.html.

[19] Z. Kincaid és S. Zhu, „LR(0) Parser Visualization," [Online]. Available: https://www.cs.princeton.edu/courses/archive/spring20/cos320/LR0/.

[20] „Bison as a Grammar Checker," LRDE, [Online]. Available: https://www.lrde.epita.fr/~tiger/doc/gnuprog2/Bison-as-a-Grammar-Checker.html.

[21] D. R. Hipp, „The Lemon Parser Generator," [Online]. Available: https://sqlite.org/src/doc/trunk/doc/lemon.html.

[22] X. Chen és D. Pager, „Full LR(1) parser generator Hyacc and study on the performance of LR(1) algorithms," in *Fourth International C* Conference on Computer Science & Software Engineering*, Montreal, 2011.

[23] A. Johnstone, E. Scott és G. Economopoulos, „The Grammar Tool Box: A Case Study," *Electronic Notes in Theoretical Computer Science*, pp. 97-113, 2004.

[24] P. Dubroy és A. Warth, „Incremental packrat parsing," in *SPLASH '17: Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, Vancouver BC, 2017.

[25] R. Ierusalimschy, L. H. d. Figueiredo és W. Celes, „Lua 5.3 Reference Manual," 2020. [Online]. Available: https://www.lua.org/manual/5.3/manual.html.

[26] R. Ierusalimschy, „Programming in Lua: Markov Chain Algorithm," 2003-2004. [Online]. Available: https://www.lua.org/pil/10.2.html.

Függelék: A Lua 5.3 hivatalos nyelvtana

A teljes EBNF jelölésű nyelvtan megtalálható a Lua 5.3 hivatalos útmutatójában [25].

```
chunk ::= block

block ::= {stat} [retstat]

stat ::=  ';' |
          varlist '=' explist |
          functioncall |
          label |
          break |
          goto Name |
          do block end |
          while exp do block end |
          repeat block until exp |
          if exp then block {elseif exp then block} [else block] end |
          for Name '=' exp ',' exp [',' exp] do block end |
          for namelist in explist do block end |
          function funcname funcbody |
          local function Name funcbody |
          local namelist ['=' explist]

retstat ::= return [explist] [';']

label ::= '::' Name '::'

funcname ::= Name {'.' Name} [':' Name]

varlist ::= var {',' var}

var ::=  Name | prefixexp '[' exp ']' | prefixexp '.' Name

namelist ::= Name {',' Name}

explist ::= exp {',' exp}

exp ::=  nil | false | true | Numeral | LiteralString | '...' |
         functiondef | prefixexp | tableconstructor | exp binop exp |
         unop exp

prefixexp ::= var | functioncall | '(' exp ')'

functioncall ::=  prefixexp args | prefixexp ':' Name args

args ::=  '(' [explist] ')' | tableconstructor | LiteralString

functiondef ::= function funcbody

funcbody ::= '(' [parlist] ')' block end

parlist ::= namelist [',' '...'] | '...'
```

```
tableconstructor ::= '{' [fieldlist] '}'

fieldlist ::= field {fieldsep field} [fieldsep]

field ::= '[' exp ']' '=' exp | Name '=' exp | exp

fieldsep ::= ',' | ';'

binop ::=  '+' | '-' | '*' | '/' | '//' | '^' | '%' |
           '&' | '~' | '|' | '>>' | '<<' | '..' |
           '<' | '<=' | '>' | '>=' | '==' | '~=' |
           and | or

unop ::= '-' | not | '#' | '~'
```

Függelék: Markov láncok Lua-ban

Az eredeti megtalálható a Lua hivatalos honlapján [26].

```
function allwords()
    local line = io.read() -- current line
    local pos = 1 -- current position in the line
    return function() -- iterator function
        while line do -- repeat while there are lines
            local s, e = string.find(line, "%w+", pos)
            if s then -- found a word?
                pos = e + 1 -- update next position
                return string.sub(line, s, e) -- return the word
            else
                line = io.read() -- word not found; try next line
                pos = 1 -- restart from first position
            end
        end
        return nil -- no more lines: end of traversal
    end
end

function prefix(w1, w2)
    return (w1 .. ' ') .. w2
end

local statetab

function insert(index, value)
    if not statetab[index] then
        statetab[index] = {
            n = 0
        }
    end
    table.insert(statetab[index], value)
end

local N = 2
local MAXGEN = 10000
local NOWORD = "\n"

-- build table
statetab = {}
local w1, w2 = NOWORD, NOWORD
for w in allwords() do
    insert(prefix(w1, w2), w)
    w1 = w2;
    w2 = w;
end
insert(prefix(w1, w2), NOWORD)
-- generate text
w1 = NOWORD;
w2 = NOWORD -- reinitialize
for i = 1, MAXGEN do
```

```
    local list = statetab[prefix(w1, w2)]
    -- choose a random item from list
    local r = math.random(table.getn(list))
    local nextword = list[r]
    if nextword == NOWORD then
        return
    end
    io.write(nextword, " ")
    w1 = w2;
    w2 = nextword
end
```