

TDK dolgozat

Hollósi János

SZÉCHENYI ISTVÁN EGYETEM
MŰSZAKI TUDOMÁNYI KAR
MATEMATIKA ÉS SZÁMÍTÁSTUDOMÁNY TANSZÉK

Véges automaták és reguláris nyelvek kapcsolata

TMDK dolgozat

Hollósi János

IV. éves Mérnök Informatikus BSc hallgató

Konzulens:
Dr. Kallós Gábor
egyetemi docens

Lezárva: Győr, 2014. június 20.

Tartalomjegyzék

1. Célkitűzés	2
2. Az elméleti háttér bemutatása.....	3
2.1. Generatív nyelvtanok.....	3
2.1.1. Levezetés	3
2.1.2. Generált nyelv	3
2.1.3. Chomsky-féle kategorizálás	3
2.2. Automaták	4
2.2.1. Rabin-Scott automata	4
2.3. Nyelvosztályok ekvivalenciája	5
2.3.1. Láncszabályok	5
2.3.1.1. Láncszabályok törlése	6
2.3.1.2. Szabályredukció	6
2.3.2. 3-típusú nyelvtanok és véges elfogadó automatak ekvivalenciája	7
3. Egy már létező megoldás bemutatása.....	8
3.1. A JFLAP bemutatása	8
3.2. A JFLAP használata	8
4. A szoftver bemutatása.....	14
4.1. A matematikai és formális nyelvi alapok megvalósítása.....	14
4.1.1. Set osztály.....	14
4.1.2. Pair osztály	15
4.1.3. Rule osztály	16
4.1.4. Configuration osztály	16
4.1.5. Language osztály	16
4.1.6. ListOfRules osztály	17
4.1.7. ListOfConfigurations osztály	18
4.1.8. TransitionFunction osztály	18
4.1.9. SetOfTransitionFunctions osztály	19
4.1.10. Grammar osztály	19
4.1.11. Machine osztály.....	22
4.1.12. Osztály diagram.....	25
4.2. A grafikus reprezentáció megvalósítása	25
4.2.1. State osztály.....	25
4.2.2. Arrow osztály	27
4.2.3. MachineElement osztály	30
4.2.4. MachineDrawer osztály.....	31
4.2.5. GrammarDrawer osztály	32
4.2.6. RulesDrawer osztály.....	33
4.3. A szoftver használata.....	34
5. Összehasonlítás	42
6. Összefoglalás.....	45
7. Irodalomjegyzék	46

1. Célkitűzés

A formális nyelvek elméletének egyik alappillére az a fontos tétel, miszerint a reguláris kifejezésekkel megadható, a véges automátákkal felismerhető és a 3-típusú nyelvtannal generálható nyelvek osztálya megegyezik.

Dolgozatomban e tétel vizsgálatával foglalkozom, különösen tekintettel a 3-típusú nyelvtanok és a véges elfogadó automáták kapcsolatára.

Ennek érdekében először röviden ismertetem a megértéshez szükséges matematikai és formális nyelvi elméleti hátteret, majd bemutatom az általam készített szoftvert, mely segítségével nyelvtanokat és véges automátákat hozhatunk létre, jeleníthetjük meg azt látványos grafikával, és elvégezhetjük az ekvivalencia vizsgálatot.

A dolgozat elkészítéséhez az elmúlt féléves „Formális nyelvek és automáták” c. tantárgy nyújtotta az inspirációt, amikor is hallgatói szemmel figyelhettem meg, hogy a tananyag elsajátítása számos hallgatónak nehézséget okoz a témakör matematikai mivolta végett.

Rövid Internetes kutakodás után megállapítottam, hogy nehéz a témával foglalkozó és jól használható olyan szimulációs szoftvert találni, mely oktatási célzattal is alkalmazható. A fellelhető tananyagok nagy része kimerül a statikus ábrák használatában.

Célul tűztem ki, hogy létrehozzak egy olyan szoftvert, mely segítséget nyújthat a téma iránt érdeklődő vagy a témát tanuló egyetemi hallgatók számára. Az alkalmazást szemléletes grafikai kimenettel képzeltem el, amivel könnyebben elsajátíthatóak és megérthetőek a tankönyvekben olykor kevésbé beszédes elméleti alapok.

A későbbiekben a munkám során – a közben fellelt robusztusabb, hasonló alkalmazások hatására – ezt a célkitűzést finomítottam, és hangsúlyozottan törekedtem arra, hogy olyan funkciókat fejlesszek ki, amelyek még a legjobb fellelt alkalmazásban sem szerepeltek. Erről még később részletesebben is írok.

Mindeközben a kutatásban is egyre fejlettebb megközelítést alkalmaztam, a magyar nyelvű irodalmaktól fokozatosan áttértem az angol könyvek és cikkek elemzésére, mivel fontos volt számomra, hogy a munkám a lokálisnál határozottabban nagyobb hangvételű legyen, és az érdeklődő szakmai közönség körében is felhívja magára a figyelmet.

2. Az elméleti háttér bemutatása

A következőkben tisztázom a feladat megértéséhez szükséges matematikai és formális nyelvi elméleti hátteret. A dolgozat olvasójáról feltételezem, hogy van némi jártassága a témakör kapcsán, így az elmélet taglalása során az egyszerűbb részekre (pl. halmazelmélet) nem térnék ki.

Az elméleti összefoglalást [5] és [7] alapján készítettem el.

2.1. Generatív nyelvtanok

A $G = (N, T, P, S)$ rendezett négyest generatív nyelvtannak (vagy generatív grammatikának) hívjuk. Az N elemeit nemterminális jeleknek nevezzük, és általában nagybetűkkel jelöljük. A T elemeit terminális jeleknek nevezzük, és általában kisbetűkkel jelöljük. N és T diszjunkt véges ábécé. A P elemeit képező (p, q) rendezett párokat átírási (helyettesítési) szabályoknak nevezzük, ahol $p, q \in (N \cup T)^*$, de p -ben van legalább egy nemterminális betű, és általában $p \rightarrow q$ alakban írjuk. Az S egy kitüntetett nemterminális jel, melyre igaz, hogy $S \in N$. S a generálás kiinduló vagy kezdő eleme, más néven mondatszimbóluma (kezdő szimbóluma).

2.1.1. Levezetés

Egy G generatív nyelvtanban az $r \in (N \cup T)^*$ szóból egy lépésben, vagy közvetlenül levezethető a $t \in (N \cup T)^*$ szó, ha van olyan $p \rightarrow q$ helyettesítési szabály a P -ben és $p_1, p_2 \in (N \cup T)^*$ úgy, hogy $r = p_1 p p_2$ és $t = p_1 q p_2$. Jelölés: $r \Rightarrow_G t$ vagy $r \Rightarrow t$.

Egy G generatív nyelvtanban az r szóból (valahány lépésben) levezethető a t szó, ha van olyan $r_0, r_1, \dots, r_n \in (N \cup T)^*$ véges szósorozat, amelyre teljesül, hogy $r_0 = r, r_n = t$ és $r_i \Rightarrow r_{i+1}$ ($i = 0, 1, \dots, n-1$). Ezt a relációt $r \Rightarrow^* t$ -vel jelöljük.

2.1.2. Generált nyelv

A $G = (N, T, P, S)$ generatív nyelvtan által generált nyelven azon szavak halmazát értjük, amelyek valahány lépésből levezethetők a kezdőszimbólumból:

$$L(G) = \{w \mid S \Rightarrow^* w, w \in T^*\}.$$

2.1.3. Chomsky-féle kategorizálás

Az osztályozás alapját a helyettesítési szabályok alakjára vonatkozó megszorítások képezik abban a hierarchiában, amelyet az elmélet egyik megalapozója, N. Chomsky vezetett be.

Azt mondjuk, hogy $G = (N, T, P, S)$ generatív nyelvtan:

- Mondatszerkezetű nyelvtan (0-típusú), ha nincs rá vonatkozó külön korlátozás.
- Környezetfüggő nyelvtan (1-típusú), ha minden P -beli szabály $\beta A \gamma \rightarrow \beta \alpha \gamma$ alakú, ahol $\alpha \neq \lambda$, vagy $S \rightarrow \lambda$ alakú.
- Környezetfüggetlen nyelvtan (2-típusú), ha minden P -beli szabály $A \rightarrow \alpha$ alakú.

- Reguláris nyelvtan (3-típusú), ha minden P -beli szabály $A \rightarrow xB$ vagy $A \rightarrow x$ alakú.

Fontos, hogy reguláris nyelvtanok esetén beszélhetünk jobb- és ballineáris esetről, attól függően, hogy a nemterminális szimbólum a szabályok jobb oldalának elején vagy végén található. Itt most csak a jobblineáris esettel foglalkozom.

Dolgozatom további részében csak a reguláris nyelvtanok esetével foglalkozom, mert ezek hozhatók kapcsolatba a véges elfogadó automatákkal. De ugyanakkor megjegyezném, hogy más típusú nyelvtanok és automaták között is felfedezhető a párhuzam, pl. a környezetfüggetlen nyelvtanok a veremautomatákkal.

2.2. Automaták

Automatán egy olyan absztrakt rendszert értünk, mely egy diszkrétnek képzelt időskála időpillanataiban érkezett ingerek hatására ezen időpillanatokban válasszal reagál, miközben belső állapotát megadott szabályok szerint változtatja a külső ingerek hatására. Az ingerekre adott válasz függ mind az ingerektől mind pedig a pillanatnyi belső állapottól. Automatákat tulajdonságaik alapján általában az alábbi kategóriákba szokták sorolni, de fontos megjegyezni, hogy ezek nem diszjunkt halmazok, az egyes fajták között lehetségesek átfedések: Mealy, Moore, kimenő jel nélküli, iniciális, parciális, teljesen definiált, nemdeterminisztikus, determinisztikus, sztochasztikus, Rabin-Scott. A továbbiakban csak a Rabin-Scott típusú automatákkal foglalkozom, mert ez a fajta automata alkalmas a reguláris nyelvek elfogadására, és a tartalmazás kérdésének megválaszolására.

2.2.1. Rabin-Scott automata

A Rabin-Scott féle automatát felismerő vagy elfogadó automatáknak is nevezzük. A Rabin-Scott féle automata egy $A = (Q, T, q_0, d, F)$ rendezett ötös, ahol Q a nem üres állapothalmaz, T a nem üres bemenő jelhalmaz, $q_0 \in Q$ a kezdőállapot, $d: Q \times T \rightarrow Q$ az átmeneti függvény, $F \subseteq Q$ pedig a végállapotok nem üres halmaza. ($q_0 \in F$ megengedett.)

A $d(B, a) = A$ mozgási szabály azt jelenti, hogy az automata a B állapotban van, és az olvasott karakter az a , akkor az új állapot A lesz.

A $d(q, a) = \{q_1, q_2, \dots, q_m\}$ mozgási szabály azt jelenti, hogy az automata a q állapotban van, és az olvasott karakter az a , akkor az új állapot $q_1, q_2, \dots, q_m$ állapotok valamelyike lesz.

Azt mondjuk, hogy az $A = (Q, T, q_0, d, F)$ automata determinisztikus, ha minden $q \in Q$ és $a \in T$ esetén a $d(q, a)$ halmaz legfeljebb egy elemű.

Ha minden $q \in Q$ és $a \in T$ esetén van $d(q, a) \neq \emptyset$ mozgási szabály, akkor az automata teljesen definiált.

Ha létezik legalább egy olyan (q, a) rendezett pár, melyre $d(q, a) = \emptyset$, akkor az automata parciális.

Teljesen definiált automata a bemenő jelsorozatot elfogadja, vagy visszautasítja. Egy nem üres, $a_1, a_2, \dots, a_n$ bemenő jelekből álló $a_1 \dots a_n$ bemenő szó esetén akkor mondjuk, hogy a

tekintett $A = (Q, T, q_0, d, F)$ automata felismeri, ha alkalmas $q_1, q_2, \dots, q_n$ állapotaira $d(q_0, a_1) = q_1, \dots, d(q_{n-1}, a_n) = q_n$ teljesülése mellett $q_n \in F$.

Parciális automata esetén előfordulhat, hogy az automata nem tudja végig olvasni a szót, ilyenkor az automata „lehal”, ezért a nem teljesen definiált automatákra nem igaz a fenti állítás, de bármely automatahoz képezhető egy vele ekvivalens teljesen definiált automata. Ezért a parciális automaták esetével külön nem foglalkozunk.

Az automata mozgási folyamatát mint úgynevezett konfigurációk egymásutánjaként követhetjük nyomon. Valamely automata konfigurációja tulajdonképpen pillanatfelvétel az automata működéséről.

A $k = (q, w)$ konfiguráció azt jelenti, hogy az automata a $q \in Q$ állapotban van és a $w \in T^*$ még el nem olvasott jelsorozatot kapja inputként.

Tetszőleges (q, w) és (q', w') konfigurációk esetén $(q, w) \mapsto (q', w')$ reláció akkor és csak akkor áll fent, ha $w = aw'$, valamely $a \in T$ -re, és fennáll $q' \in d(q, a)$ tartalmazás.

Egy automata által felismert $L(M)$ nyelve alatt azon jelsorozatok halmazát értjük, amelyeket az automata elfogad: $L(M) = \{w \mid (q_0, w) \mapsto^* (p, \lambda) \wedge p \in F\}$.

2.3. Nyelvosztályok ekvivalenciája

Bizonyítható, hogy tetszőleges ábécé felett a 3-típusú nyelvek osztálya, a véges automatákkal felismerhető nyelvek osztálya és a reguláris nyelvek osztálya egymással megegyezik.

Ennek bizonyítását 3 részre lehet bontani, melyek a következők:

- Tetszőleges T ábécé esetén minden T feletti L reguláris nyelv generálható 3-típusú nyelvtannal.
- Tetszőleges T ábécé feletti 3-típusú nyelv felismerhető véges elfogadó automatával.
- Tetszőleges T ábécé feletti, automatával felismerhető nyelv reguláris.

A továbbiakban csak a 3-típusú nyelvtanok és a véges elfogadó automaták közti kapcsolat vizsgálatára összpontosítok (2. állítás a felsorolásban), mert gyakorlati szempontból ezen két nyelvosztály kapcsolata bír a legnagyobb jelentőséggel.

A bizonyítás előtt két fontos lemmát kell megemlíteni.

2.3.1. Láncszabályok

Tetszőleges $G = (N, T, P, S)$ nyelvtan esetén az $A \rightarrow B$ alakú szabályokat láncszabályoknak nevezzük. Továbbá amennyiben P -ben nincsenek láncszabályok, úgy azt mondjuk, hogy G láncszabálymentes.

1. Lemma. Legyen $G = (N, T, P, S)$ környezetfüggetlen nyelvtan. Megadható olyan $G' = (N, T, P', S)$ láncszabálymentes környezetfüggetlen nyelvtan, amelyre $L(G') = L(G)$. Amennyiben G 3-típusú, úgy G' is 3-típusú lesz.

2.3.1.1. Láncszabályok törlése

Először minden $A \in N$ -re meghatározzuk azon $B \in N$ nemterminálisok halmazát, melyekre $A \Rightarrow^* B$ úgy, hogy a levezetés közben csak láncszabályokat alkalmaztunk. Jelöljük ezt a halmazt N_A -val.

Az N_A halmaz kiszámítását a következő iterációval végezzük:

- (i) Legyen $N_0 = \{A\}$ és legyen $i = 0$.
- (ii) Legyen $N_{i+1} = N_i \cup \{C \in N \mid \exists B \in N_i \wedge B \rightarrow C \in P\}$.
- (iii) Amennyiben $N_i = N_{i+1}$ legyen $N_A = N_i$, különben $i = i + 1$ és ismételjük meg az (ii) ponttól.

Most konstruáljuk meg P' -t a következőképpen:

$$P' = \emptyset.$$

Minden $A \in N$ -re,

Minden $B \in N_A$ -ra,

Minden $B \rightarrow \alpha \in P$ szabály esetén:

Ha $B \rightarrow \alpha$ nem láncszabály, akkor
vegyük fel az $A \rightarrow \alpha$ szabályt P' -be.

Nyilvánvaló, hogy P' -ben nem lesznek láncszabályok, továbbá teljesülni fog, hogy $L(G') = L(G)$.

2.3.1.2. Szabályredukció

2. Lemma. Legyen $G = (N, T, P, S)$ 3-típusú láncszabálymentes nyelvtan. Akkor van olyan $G' = (N', T, P', S)$ 3-típusú nyelvtan, hogy $L(G') = L(G)$ és P' -ben minden szabály $A \rightarrow aB$ vagy $A \rightarrow \lambda$ alakú, ahol $A, B \in N$ és $a \in T$.

Konstruáljuk meg P' -t a következőképpen:

- (i) Minden $A \rightarrow aB$ vagy $A \rightarrow \lambda$ alakú P -beli szabályt vegyünk fel P' -be.
- (ii) Minden $A \rightarrow a_1 a_2 \dots a_n B$, P -beli szabály esetén (ahol $n > 1$, $a_1, a_2, \dots, a_n \in T$) vegyünk fel P' -be az $A \rightarrow a_1 A_1$, $A_1 \rightarrow a_2 A_2$, ..., $A_{n-1} \rightarrow a_n B$ szabályokat, ahol $A_1, A_2, \dots, A_{n-1}$ új nemterminális szimbólumok.
- (iii) Minden $A \rightarrow a_1 a_2 \dots a_n$, P -beli szabály esetén (ahol $n > 1$, $a_1, a_2, \dots, a_n \in T$) vegyünk fel P' -be az $A \rightarrow a_1 A_1$, $A_1 \rightarrow a_2 A_2$, ..., $A_{n-1} \rightarrow a_n A_n$, $A_n \rightarrow \lambda$ szabályokat, ahol $A_1, A_2, \dots, A_{n-1}, A_n$ új nemterminális szimbólumok.
- (iv) Legyen továbbá N' az N halmaz és az előbbi konstrukció során keletkezett új nemterminálisok halmazának egyesítése.

Ekkor nyilvánvaló, hogy P' -ben csak a kívánt alakú szabályok fognak szerepelni. Az is könnyen belátható, hogy $L(G') = L(G)$.

2.3.2. 3-típusú nyelvtanok és véges elfogadó automaták ekvivalenciája

Legyen $G = (N, T, P, S)$ 3-típusú nyelvtan. Megadható olyan $A = (Q, T, q_0, d, F)$ véges elfogadó automata, melyre $L(A) = L(G)$.

Az eddigiek alapján feltehető, hogy G nyelvtan láncszabálymentes és minden P -beli szabály $A \rightarrow aB$ vagy $A \rightarrow a$ alakú. Konstruáljuk meg A -t a következőképpen. Legyen

- $Q = N$,
- $q_0 = S$,
- minden $A \in N$ és $a \in T$ esetén $d(A, a) = \{B \in N \mid A \rightarrow aB \in P\}$,
- $F = \{A \in N \mid A \rightarrow \lambda \in P\}$.

A konstrukció fordított esetben is létezik, tehát adott A véges elfogadó automatához megadható olyan G 3-típusú nyelvtan, melyre $L(G) = L(A)$.

Ezen esetben a konstrukció a következőképpen zajlik. Legyen

- $N = Q$,
- Ha d tartalmaz $d(q_i, a) = q_j$ átmenetet, akkor P -be vegyük fel a $q_i \rightarrow aq_j$ szabályt. Amennyiben $q_k \in F$, akkor P -be vegyük fel a $q_k \rightarrow \lambda$ szabályt.
- $S = q_0$.

3. Egy már létező megoldás bemutatása

A bevezető fejezetben már említést tettem arról, hogy munkám megkezdése előtt Internetes kutatást végeztem. Sikerült fellelnem több a témával foglalkozó alkalmazást is. A következőkben bemutatom az egyik ilyen, véleményem szerint legjobb megvalósítást, mely a JFAP nevet viseli.

3.1. A JFLAP bemutatása

A JFLAP (Java Formal Languages and Automata Package) egy Java programozási nyelven íródott, összetett formális nyelvi és automata szimulációs alkalmazás, mely támogatja a nem determinisztikus véges automatákat, determinisztikus veremautomatákat (pushdown automata), Turing gépeket, reguláris nyelveket és környezetfüggetlen nyelveket.

A szoftvercsomag fejlesztése a 90-es évek elejére nyúlik vissza. A fejlesztés kezdetben a New York-i Rensselaer Polytechnic Institute-ban zajlott Susan Rodger irányítása alatt. 1994-ben Susan Rodger a Durham-i Duke University-n kezdett el dolgozni, ekkor a projekt is átkerült erre az egyetemre. A fejlesztés azóta is ott zajlik.

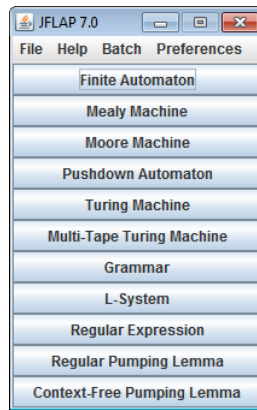
Jól látható, hogy a JFLAP fejlesztése immáron 24 évre nyúlik vissza, ami idő alatt több nemzetközileg is elismert intézmény számos hallgatója vett részt a kutatásban. Mindezt jól tükrözi a szoftver által nyújtott szolgáltatások széles skálája és a megvalósított funkciók komplexitása is. Úgy gondolom, hogy ezek a tények alkalmassá teszik a szoftvert arra, hogy összehasonlítási alapként szolgáljon az általam megvalósított alkalmazáshoz.

3.2. A JFLAP használata

A továbbiakban bemutatom a JFLAP használatát, különösen azokat a funkciókat, melyeket vagy melyekhez hasonlókat magam is megvalósítok a szoftveremben.

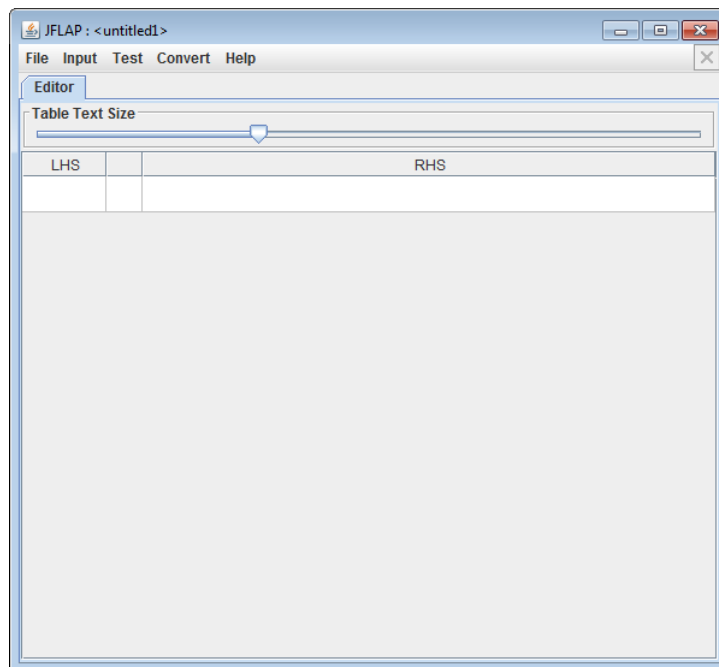
A szoftver használatához először navigáljunk a www.jflap.org honlapra, ahol a bal oldali menüből válasszuk a „Get JFLAP” lehetőséget. A megjelenő oldalon kattintsunk a felajánlott kérdőívre. A kérdőív megfelelő kitöltése után az oldal átvezet minket a letöltési oldalra, ahol időrendi sorrendben találjuk a szoftver egyes verzióit. Célszerű a program legfrissebb verzióját választani. Ha rákattintunk valamelyikre, akkor elindul annak letöltése. Letöltés után az alkalmazás futtatható.

A szoftver indítása után egy kezdőképernyőn választhatjuk ki, hogy mit szeretnénk modellezni.



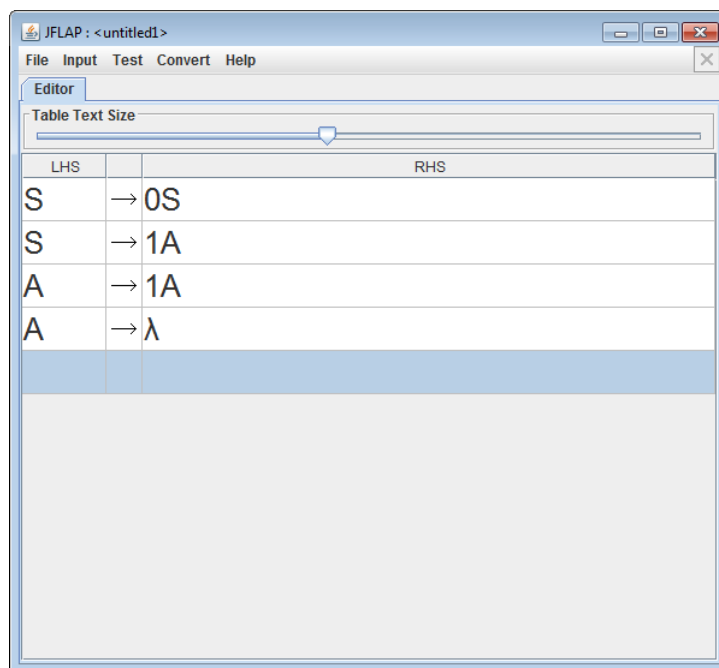
1. ábra: A JFLAP kezdőképernyője

Ha rákattintunk valamely nyomógombra, akkor létrehoz egy új példányt a választott elemből, ami egy új ablak formájában jelenik meg. Most egy új nyelvtan létrehozását fogom bemutatni.



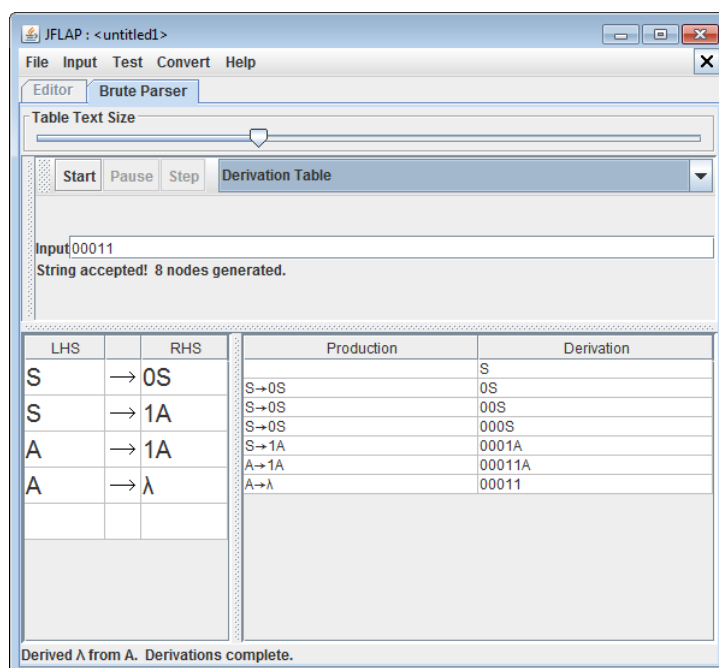
2. ábra: Nyelvtan szerkesztőablak

A megjelenő felületen egy csúszka segítségével lehetőség van módosítani a betűméreten. A csúszka alatti táblázatba kattintással vihetők fel a nyelvtanhoz tartozó átírási szabályok. Az LHS oszlopba kerül a szabály bal oldala, míg az RHS oszlopba a szabály jobb oldala.



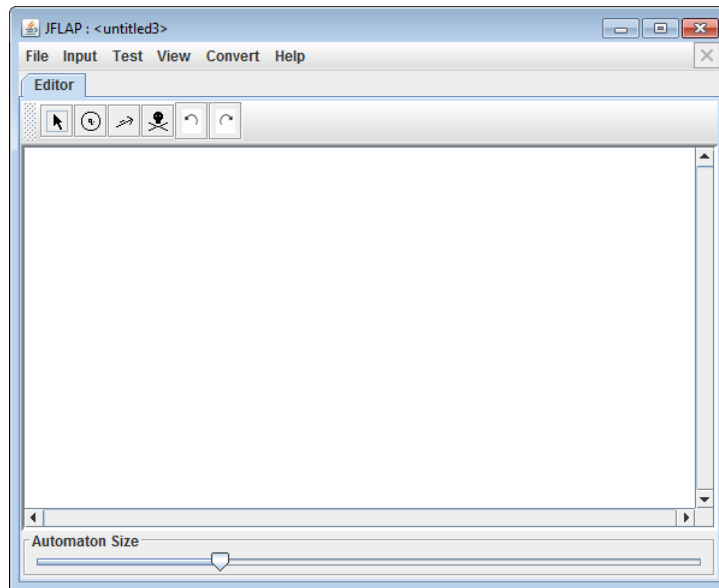
3. ábra: Átírási szabályok felvétele a nyelvntanba

Ha felvettük a nyelvntan szabályait, lehetőség van tetszőleges szó levezethetőségének vizsgálatára. A szó megadása után a szoftver tájékoztat a levezetés sikerességéről, illetve amennyiben a levezetés lehetséges, megadja, hogy mely átírási szabályok egymás utáni alkalmazásával juthatunk el a kezdőszimbólumtól a levezetni kívánt szóig.



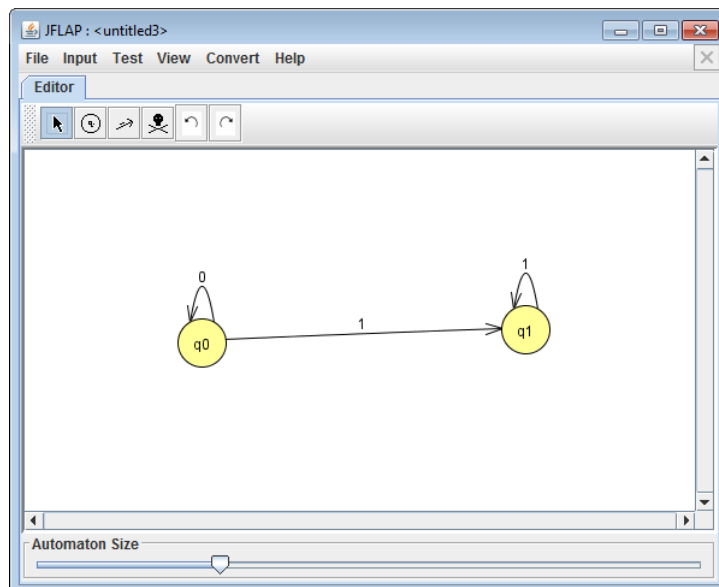
4. ábra: Szó levezetése

A JFLAP segítségével véges automatákat is megkonstruálhatunk. Ehhez ismét a kezdőképernyőre kell navigálnunk és a megfelelő nyomógomb segítségével létrehozni egy új automatát. Ekkor megjelenik az automata szerkesztő ablak.



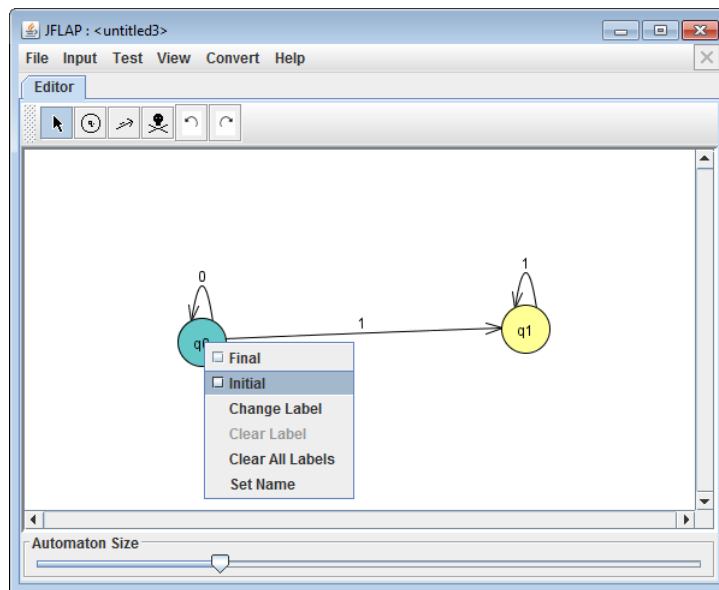
5. ábra: Automata szerkesztő ablak

A felületen egyszerű rajzeszközökkel tudjuk megszerkeszteni a kívánt automatát. Lehetőség van állapotok felvételére és a köztük lévő átmenet megrajzolására, illetve az átmenethez tartozó bemeneti jel megadására.



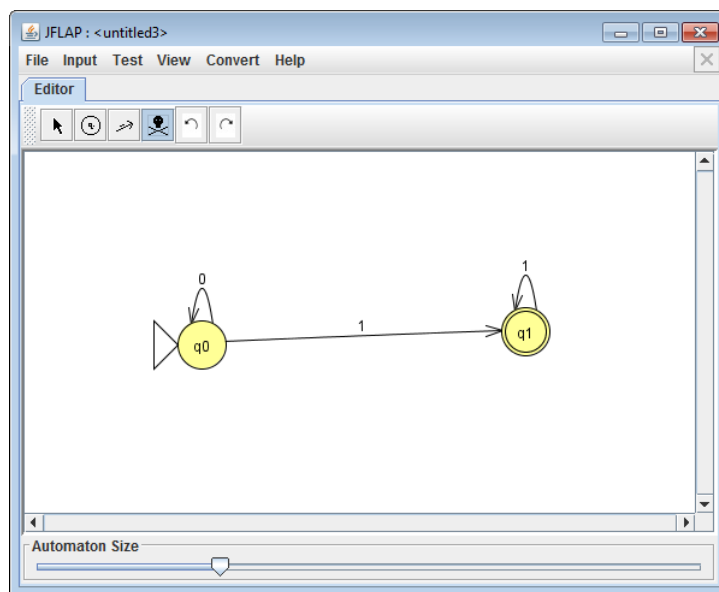
6. ábra: Automata szerkesztése

Állapotok felvétele után mód nyílik kezdőállapot és végállapotok megjelölésére. Ehhez a kívánt állapotot reprezentáló grafikus objektum helyi menüjén belül kell kiválasztani a megfelelő opciót.



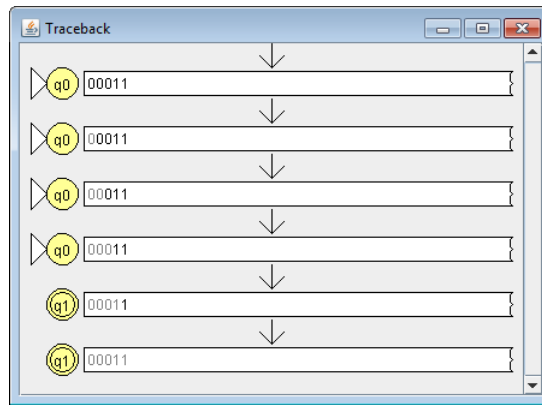
7. ábra: Kezdőállapot beállítása

A kezdőállapot és végállapotok megadása után a megszokott jelöléssel tünteti ki a szoftver ezeket az állapotokat.



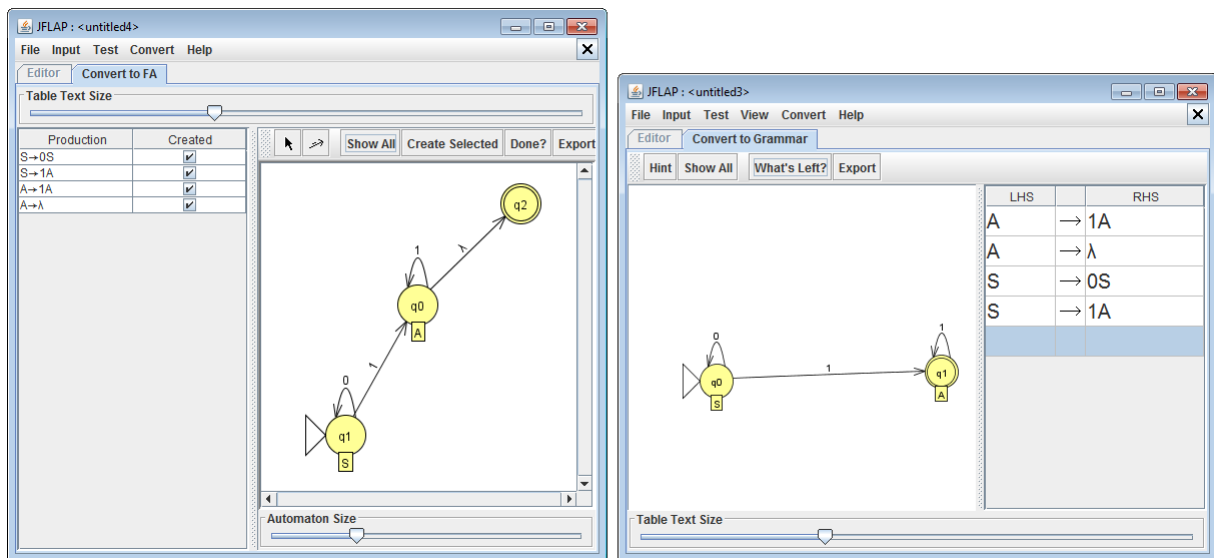
8. ábra: Kezdő és végállapotok megkülönböztetése

A nyelvtanokhoz hasonlóan lehetőség van tetszőleges bemeneti jelsorozat elfogadásának vizsgálatára. Megfelelő bemenet esetén egy új ablakban jeleníti meg a jelsorozat hatására egymás után bekövetkező állapotokat.



9. ábra: Elfogadás lépései

Nyelvtan szerkesztése esetén lehetőség van ekvivalens automata generálására. Hasonló módon az automata szerkesztő felületen kreálhatunk egyenértékű nyelvtant az automatahoz.



10. ábra: Ekvivalens automata és nyelvtan generálása

A bemutatott funkciókon felül még számos lehetőséget rejt az alkalmazás, amikre most nem térnék ki. Már ezekből is jól látszik, hogy valóban egy gondosan megtervezett és implementált szoftverrel állunk szemben, ami remek szimulációs eszközként szolgál a formális nyelvek és automata témaköréhez.

Azonban JFLAP tüzetes áttekintése után azt tapasztaltam, hogy a szoftver mutat némi hiányosságot, ami megnehezíti az elmélet elsajátítását, illetve nem teljesen illeszkedik a tankönyvekben megszokott módszerekhez, jelölésmódokhoz. Ezek hátráltathatják a hallgatókat a tananyag megértésében, rosszabb esetben eltántoríthatják őket a témakör tanulásától.

Mindezen kiindulva kitaláltam egy meglátásom szerint jobb és hatékonyabb megvalósítási módot, ami remélhetőleg hatékonyabb eszközként szolgál a formális nyelvek és automata elméletének elmélyítése során a hallgatóság számára.

4. A szoftver bemutatása

A második fejezetben tárgyalásra került fontos matematikai és formális nyelvi alapismeretek nyomán megvalósítottam a szükséges modelleket reprezentáló, az objektum orientáltság szemléletét követő alapvető osztályokat. Az osztályokat oly módon készítettem el, hogy azok a lehető leghűbben tükrözzék a valós elméleti feltevéseket. Így folyamatosan nyomon követhető az egyes osztályok közötti kapcsolat, úgy, mint öröklődés, tartalmazás. A fejezet során részletesen bemutatom a szoftver összes általam megvalósított osztályát, kitérve azok működésére, s néhány példán át a kívánt funkció prezentálására.

A szoftver összetevőit két nagy csoportra lehet osztani. Első részbe tartoznak azok az osztályok melyek az elméleti alapokat kívánják reprezentálni. S ezekre építkezve készítettem el az alkalmazás második felét, mely a grafikus megjelenésért felel. A két halmaz létjogosultsága ugyanannyira fontos, hiszen a szükséges matematikai és nyelvi alapokat tartalmazó osztályok nélkül a szoftver eleve nem lenne életképes, a grafikus reprezentáció hiányában pedig elveszne a szoftver eredeti célkitűzése, miszerint az alkalmazás legyen képes az elméleti háttér gördülékeny megértetésére.

4.1. A matematikai és formális nyelvi alapok megvalósítása

A következőkben bemutatom a megvalósított formális nyelvi elméleti alaposztályokat és azok kapcsolatát. A tárgyalás során jól megfigyelhető az osztályok hierarchikus egymásra épülése. Kezdetben szemléltetem a legalapvetőbb funkciókat megvalósító típusokat, s ahogy haladunk előre, az ismertett osztályok egyre összetettebbé, komplexebbé válnak. A tárgyalás végül a reguláris nyelvtan és véges elfogadó automata modelljével válik teljessé. Ezzel a megközelítéssel elértem, hogy minden feladat csak a maga szintjén legyen jelen, magasabb szinteken ne forduljanak elő alacsonyabb szinten már tisztázott problémák. Ez az elgondolás nagymértékben leegyszerűsíti a szoftver fejlesztést, megkönnyíti az olyan komplexitású modellek létrehozását, mint a tárgyalt formális nyelvi témakör.

4.1.1. Set osztály

A klasszikus értelemben vett matematikai halmaz modelljét képeztem ezzel az osztállyal. Programozás-technikailag annyi különbség megfigyelhető az osztály és a matematikai értelemben vett halmaz között, hogy itt a halmaz elemeire vonatkozó szabály helyett azt kell megadni, hogy a halmaz milyen típusú objektumok tárolására legyen képes. A hagyományos halmazoknál ezt az elemeket jellemző tulajdonság(ok) megadásával tehetjük meg, itt ez a megadás kissé elvonatkoztatva jelenik meg. Az osztály példányának létrehozása során a tartalmazandó objektumok típusát kell a konstruktor rendelkezésére bocsátani. Ezt a megoldást a manapság nagy népszerűségnek örvendő *generikus osztályok* modelljével készítettem el. Ebben a megközelítésben az osztály definíciója során nem kötjük ki, hogy az pontosan milyen típusú objektumokkal képes dolgozni, csupán egy jelölést (általában nagy „T” betű, a *template* szóra utalva) használunk ott, ahol a megfelelő osztálynevek állnának. Mindössze az osztály példányosítása során kerül meghatározásra, hogy az adott példányban milyen típust jelöljön a fent nevezett szimbolikus „T” típus. Ez a megközelítés azért előnyös, mert olyan típusokat hozhatunk létre, amik több különböző fajta objektummal is kapcsolatban

állhatnak a megoldandó feladattól függően és nem kell minden feladatra külön osztályt létrehozni. Elegendő egy generikus osztályt úgy megvalósítani, hogy az hatékonyan tudjon dolgozni bármilyen várható körülmény között. Erre kitűnő példa az általam létrehozott halmaz osztály, mely bármilyen adattípus tárolása esetén meg tudja valósítani az elvárt funkciókat.

Az újonnan létrehozott objektum inentől már csak a megadott típussal tud együtt dolgozni. A példányosítás után lehetőség adódik elemek felvételére a halmazba. A tartalmazás tekintetében ismét egy apró különbség vehető észre a matematikai értelemben vett halmazokhoz képest. A halmazok alap tulajdonsága, hogy elemeik nem rendezettek, s egy elemet csupán csak egyszer tartalmazhat. Programozás-technikailag lehetetlen az a megközelítés, hogy az elemek ne legyenek valamilyen módon rendezve, így a rendezettség tulajdonság az általam megvalósított halmaz esetén is megfigyelhető, de mindezt úgy valósítottam meg, hogy ez a szükséges „rossz” tulajdonság minél inkább rejtve maradjon a felhasználó szeme elől. A másik probléma a többszöri tartalmazás. Az elemek halmazon belüli tárolását tekintve lehetséges ugyan, hogy egy elem többször is szerepeljen az adatstruktúrában, de ezt megint csak úgy valósítottam meg, hogy ezt a felhasználó ne tudja kihasználni. Ily módon, ha egy új elem kerül felvételre a halmazba, akkor a tényleges hozzáadás előtt az osztály példánya mindenképp ellenőrzi, hogy az adott halmaz objektum tartalmaz-e már az újonnan felvenni kívánt objektummal egyenértékű elemet. Amennyiben nem, csak abban az esetben adja hozzá a halmazhoz az új elemet. Megjegyzendő, hogy ez a két apró megkötés nem egyedi, ezt más (nagyobb) alkalmazások esetén is így valósítják meg.

```
Set<int> h1 = new Set<int>(); //Egészek tárolására alkalmas példány.
h1.Add(7); //Elem felvétele a halmazba.
h1.Add(42);
h1.Add(7); //Ez már nem fog bekerülni, mert egy ekvivalens érték már szerepel a
halmazban.

Set<string> h2 = new Set<string>(); //Szó tárolására alkalmas példány.
h2.Add("alma");
h2.Add(13); //Hibás, nem fog lefordulni.
```

Az osztályban megvalósítottam számos, a feladat szempontjából nélkülözhetetlen, halmazokra vonatkozó műveleteket, úgy mint két halmaz uniója, metszete és különbsége. Illetve megvizsgálhatjuk, hogy két adott halmaz diszjunkt-e, valamint képezhetjük két halmaz Descartes szorzatát is, de lehetőség van két halmazról eldönteni, hogy ekvivalensek vagy sem. Ezek a műveletek a matematikában megszokott módon működnek, így részletezésüket nem tartom szükségszerűnek.

```
Set<int> h3 = new Set<int>();
h3.Add(42);
h3.Add(10);
h3.Add(2);
Set<int>.Unions(h1, h3); //A h1 és h2 halmaz unióját képezi.
```

4.1.2. Pair osztály

A matematikából ismert párt, azon belül is a rendezett pár modelljét valósítottam meg az osztály segítségével. Akárcsak a Set osztály esetén, itt is a generikus osztálymodell alapján

terveztem meg a működést. Ennek megfelelően a *Pair* objektum egy-egy „T” típusú elemet tartalmaz, melyek sorrendje is fontos. Az osztályon műveletek nincsenek értelmezve, mivel funkciója egyedül az adatok tárolása és azok szolgáltatása. Természetesen az ekvivalencia vizsgálatot ezen osztály esetén is megvalósítottam, így két *Pair* típusról eldönthető, hogy megegyeznek-e. Az osztály önállóan csupán a *Set* osztály Descartes-szorzatot számító függvényének visszatérési értékében jelenik meg, hiszen ennek eredménye rendezett párok halmaza. De az osztály létjogosultsága nem áll meg ennyiben, több magasabb szintű osztály használja a *Pair* típust, mint ősoosztályt. Ezek a későbbiekben tárgyalásra kerülnek.

```
Pair<int> p1 = new Pair<int>(3, 5); //Egész pár érték tárolására alkalmas
példány. A tárolt érték (3, 5).
p1.GetA(); //A pár bal elemének lekérdezése.
p1.GetB(); //A pár jobb elemének lekérdezése.
p1.SetA(12); //A tárolt érték megváltoztatása. Az új érték a (12, 5) lesz.
Console.WriteLine(p1); //Kimenet: (12, 5).
```

4.1.3. Rule osztály

A *Pair* osztály egy speciális megjelenése, annak egy leszármazottja. A nyelvtanok esetén használt átírási szabályhalmaz elemeinek típusát írom le vele, ahol a pár első eleme az átírandó részszó, második eleme pedig az új részszó.

```
Rule r1 = new Rule("A", "ab"); //Itt már nem kell megadni az elem típusát, hisz
csak String-eket képes tárolni.
Console.Write(r1); //Kimenet: A->ab.
```

4.1.4. Configuration osztály

Szintén a *Pair* osztály utódja. Itt az automaták állapotának leírására szolgáló konfigurációk tárolására alkalmas osztályról van szó, ahol az adott pár első eleme az automata jelenlegi állapota, míg második eleme az adott állapotban inputként kapott jelsorozat.

```
Configuration c1 = new Configuration("q", "abc"); //Az aktuális állapot "q", az
olvasott szó "abc".
Console.WriteLine(c1); //Kimenet: (q, abc).
```

4.1.5. Language osztály

A nyelvtanok által generált nyelv tárolására alkalmas osztály. Ezt a típust a *Set* osztály kiegészítéseként hoztam létre, annak egy leszármazottja oly módon, hogy az csak *String* (szó) típusú elemeket tartalmazhat. Ennek megfelelően minden halmazművelet értelmezve van a nyelv osztályon is, ahogy ez a matematikai értelemben is megfigyelhető, azonban ezeket kiegészítettem néhány plusz, a feladat szempontjából elkerülhetetlen, nyelvi funkcióval. Fontos megjegyezni, hogy nem minden nyelvi funkció került megvalósításra, lévén, hogy jelen feladat megoldásához nem minden nyelvi műveletre van szükség. Így a program felesleges bonyolítását elkerültem.

```
Language l1 = new Language();
l1.Add("ab"); //Szó felvétele a nyelvbe.
l1.Add("cd");
Console.WriteLine(l1); //Kimenet: {ab, cd}.
```

Az osztály segítségével képezhetjük két nyelv konkatenációját, mely azok összes szavára nézve előállítja a konkatenált szavakat és visszaadja egy új nyelvként. Ez a művelet talán a legfontosabb mind közül, mert az összes többit erre építettem vagy valamilyen módon felhasználtam létrehozásuk során ezt a funkciót.

```
Language l2 = new Language();  
l2.Add("q");  
Console.WriteLine(Language.Concatenation(l1, l2)); //Kimenet: {abq, cdq}.
```

Lehetőség van egy halmaz n -edik hatványát képezni, mely a kitevőtől függően visszaad egy üres halmazt ($n = 0$), a bemeneti nyelvet ($n = 1$), vagy a nyelv többször egymás után önmagával vett konkatenáltját ($n > 1$).

```
Console.WriteLine(Language.Power(l1, 2)); //Kimenet: {abab, abcd, cdab, cdcd}.
```

A Kleene-plusz és Kleene-csillag műveleteket sajnos kissé át kellett értelmezni, a megvalósíthatóság érdekében. Ezek a műveletek egy-egy végtelen szókészletű nyelvet generálnak, ami programozás-technikailag nem túl szerencsés. Ezért ezt a két funkciót úgy készítettem el, hogy azoknak nem csak a kiindulási nyelvet kell megadni, hanem egy egész értéket is, mely arra utal, hogy hány iteráció erejéig generálja az új nyelv szavait. Ez a megvalósítási módszer nem egyedi, ilyen esetben általában ehhez a megkötéshez folyamodnak a fejlesztők. Ettől eltekintve a megvalósítás ugyanazt a modellt követi, mint amit matematikai értelemben értünk a műveletek alatt. A Kleene-plusz műveletben itt is jól megfigyelhető a konkatenáció és a *Set* osztályból örökölt unió műveletek felhasználása, a Kleene-csillag művelet pedig gyakorlatilag a Kleene-plusz kiegészülése az üres szó hozzáadásával, ezért itt is ugyanúgy megjelenik a konkatenációra való visszavezetés.

```
Console.WriteLine(Language.KleenePlus(l2, 3)); //Kimenet: {q, qq, qqq}.  
Console.WriteLine(Language.KleeneStar(l2, 3)); //Kimenet: {EMPTY_STRING, q, qq, qqq}.
```

Megvizsgálhatjuk, hogy egy forrás szó szuffixe egy megadott szó vagy sem. Ezt úgy valósítottam meg, hogy válaszként nem logikai értéket szolgáltat a függvény, hanem visszaadja a szuffix kezdő indexét. Illetve ha nem szerepel a szóban a részszó, akkor -1 értékkel tér vissza. Itt beszélhetnénk prefixről is, de ez a feladat szempontjából nem szükséges funkció, így nem került kifejtésre.

```
Console.WriteLine(Language.isSuffix("abba", "ba")); //Kimenet: 2.  
Console.WriteLine(Language.isSuffix("abba", "aba")); //Kimenet: -1.
```

4.1.6. *ListOfRules* osztály

Egy szó levezetéséhez szükséges egymás után alkalmazott helyettesítési szabályok listáját modelleztem. Ennek megfelelően tartalmazza az egymás után következő *Rule* objektumok listáját, és egy másik listát, mely az alkalmazott szabályok közötti aktuális szavakat foglalja magában. Lehetőség van következő szabály és szó felvételére, azonban ügyelni kell arra, hogy a legelső (kiindulási) szó esetén még nem beszélhetünk alkalmazott szabályról. Ezt figyelembe véve a hozzáadó metódus létrehozásánál felhasználtam a többalakúság (*polimorfizmus*) lehetőségét. Ebben a megközelítésben egy adott függvénynek több megvalósítása is lehet, ahol a függvények bemeneti paraméterei egyértelműen

különböznek. Így készítettem egy olyan hozzáadó függvényt, mely csak egy *String* objektumot vár, ez lesz a kezdőszó. Illetve megvalósítottam egy másikat, mely a következő szó (*String*) és a hozzá vezető szabály (*Rule*) objektumokat várja. Minden további alkalmazott szabály és így létrejött szó ez utóbbi metódussal vehető fel.

```

ListOfRules lor1 = new ListOfRules();
lor1.Add("S"); //A kezdőszimbólum felvétele, itt még nincs szabály.
lor1.Add("abB", new Rule("S", "abB")); //További levezetések felvétele.
lor1.Add("abba", new Rule("B", "ba"));
Console.WriteLine(lor1); //Kimenet: S->abB->abba.
Console.WriteLine(lor1.ToStringWithChain()); //Kimenet: S->[S->abB]->abB->[B-
>ba]->abba.

```

4.1.7. *ListOfConfigurations* osztály

Egy konfigurációval egy automata aktuális állapotát figyelhetjük meg, több egymás utáni konfigurációval pedig egy automata mozgását lehet rekonstruálni. Az osztály feladata az egymás utáni konfigurációk listában történő tárolása. Ezt azért valósítottam meg, hogy egy adott szó elfogadásának a lépéseit tároljuk.

Módunk nyílik következő konfiguráció felvételére, illetve az egyes konfigurációk lekérdezhetőek. Mivel a feladat megoldása során szinte mindig a listában szereplő legutóbbi konfiguráció lekérdezése a releváns, ezért ennek szolgáltatására létrehoztam egy külön függvényt. Így nem kell azzal foglalkozni, hogy éppen hányadik elem az utolsó a listában, ezt az osztály elvégzi.

```

ListOfConfigurations loc1 = new ListOfConfigurations();
loc1.Add(new Configuration("q", "abc")); //Következő konfiguráció felvétele.
loc1.Add(new Configuration("a", "bc"));
loc1.Add(new Configuration("b", "c"));
loc1.Add(new Configuration("c", ""));
Console.WriteLine(loc1); //Kimenet: (q, abc)->(a, bc)->(b, c)->(c, λ).
loc1.GetLastElement(); //A sorban az utolsó konfigurációt adja vissza.

```

4.1.8. *TransitionFunction* osztály

Automaták esetében az átmeneti függvény tárolását megvalósító osztály. Itt a tárolást úgy valósítottam meg, hogy az objektumban mentésre kerül az automata aktuális állapota, az aktuális állapotban feldolgozandó bemenő jel, és a bemenő jel hatására bekövetkező állapotátmenetek halmaza. A megfelelő függvényekkel lehetséges bármely adat lekérdezése és módosítása. Mivel az újonnan bekövetkező állapotok halmaza több elemet is tartalmazhat, ezért itt nem kell egyszerre megadni az összes állapotot, lehetőség van egyenként felvenni őket a halmazba.

```

TransitionFunction tf = new TransitionFunction();
tf.SetActState("q"); //Aktuális állapot megadása.
tf.SetInput("a"); //Bemenő jel megadása.
tf.AddNextState("w"); //Jel hatására bekövetkező új állapot felvétele.
Console.WriteLine(tf); //Kimenet: d(q, a)={w}.
tf.AddNextState("e");
Console.WriteLine(tf); //Kimenet: d(q, a)={w, e}.

```

4.1.9. SetOfTransitionFunctions osztály

Átmeneti függvények halmazát tároló típus. Az általános halmazhoz képest beleépítettem két plusz funkciót, mely az átmeneti függvények létrehozása és tárolása szempontjából fontos.

Tartalmaz egy átmeneti függvény felvételére alkalmas metódust, melynek meg kell adni az átmenet paramétereit, úgy mint aktuális állapot, bemenő jel és következő állapot. Itt most nem a következő állapotok halmazát kell megadni, csupán egy elemet. Ha több átmeneti állapot is van, akkor egymás után többször kell meghívni a függvényt, mindig a felvenni kívánt új állapottal. Felvételkor a metódus tudni fogja, ha már létezik olyan átmeneti függvény, melynek az aktuális állapota és a bemenő jele megegyezik a felvenni kívánttal, ezért nem egy új *TransitionFunction* típust hoz létre, hanem a meglévőnek az új állapotok halmazához írja hozzá az felvenni kívánt állapotot.

```
SetOfTransitionFunction sotf1 = new SetOfTransitionFunction();
sotf1.Add(tf); //TransitionFunction felvétele.
sotf1.Add("w", "b", "q"); //Új elem megadható paramétereivel is, az osztály
gondoskodik a TransitionFunction létrehozásáról.
sotf1.Add("q", "a", "x"); //Tudni fogja, hogy "q" állapotból "a" jel hatására már
van átmenet, nem hoz létre új TransitionFunction típust.
Console.WriteLine(sotf1); //Kimenet: {d(q, a)={w, e, x}, d(w, b)={q}}.
```

Megvizsgálhatjuk, hogy az átmeneti függvények halmaza tartalmaz-e átmenetet adott állapot és bemenő jel hatására. Ez a funkció az automaták során lesz hasznos, amikor levezetést akarunk vizsgálni, vagy el szeretnénk dönteni, hogy adott automata parciális vagy sem.

```
Console.WriteLine(sotf1.IsContainsTransition("q", "b")); //Kimenet: False.
Console.WriteLine(sotf1.IsContainsTransition("q", "a")); //Kimenet: True.
```

4.1.10. Grammar osztály

A *Grammar* osztály felel a formális nyelvi értelemben vett nyelvtan modellezéséért. A feladat ugyan a 3-típusú (jobblineáris) grammatikák vizsgálatára összpontosul, mindazonáltal az osztályt úgy valósítottam meg, hogy azzal bármilyen nyelvtan tárolható, viszont bizonyos (főbb) funkciók csak az említett reguláris nyelvtanok típusára alkalmazhatók.

Az osztály adattagjainak tárolását ugyanúgy valósítottam meg, mint ahogy az a valós nyelvtanok esetén is elvárt, így az eddigiek alapján különösebb nehézséget nem okoz annak példányosítása. Szükség van a nemterminális és terminális elemek halmazára, az átírási szabályok halmazára, és a kezdőszimbólumra. Ezeket a már korábban megvalósított alapvető osztályok (halmaz, nyelv, stb.) segítségével tároltam.

```
Language N = new Language(); //Nemterminális elemek halmaza.
Language T = new Language(); //Terminális elemek halmaza.
Set<Rule> P = new Set<Rule>(); //Átírási szabályok halmaza.
string S; //kezdőszimbólum.

//Nemterminális elemek felvétele.
N.Add("A");
N.Add("B");
N.Add("S");
N.Add("Q");
```

```

//Terminális elemek felvétele.
T.Add("a");
T.Add("b");

//Átírási szabályok felvétele.
P.Add(new Rule("A", "a"));
P.Add(new Rule("B", "b"));
P.Add(new Rule("A", "aQ"));
P.Add(new Rule("B", "bQ"));
P.Add(new Rule("S", "A"));
P.Add(new Rule("S", "B"));
P.Add(new Rule("Q", ""));
P.Add(new Rule("Q", "S"));

S = "Q"; //Kezdőszimbólum megadása.

Grammar G = new Grammar(N, T, P, S); //Nyelvtan típus létrehozása.
Console.WriteLine(G); //Kimenet: ({A, B, S, Q}, {a, b}, {A->a, B->b, A->aQ, B->bQ, S->A, S->B, Q->λ, Q->S}, Q).

```

Mivel megvan rá a mód, hogy olyan nyelvtant generáljunk az osztállyal, mely nem elégíti ki a grammatikákkal és főleg a jobblinéaris nyelvtanokkal szemben támasztott követelményeket, ezért megvalósítottam két olyan metódust a típuson belül, melyekkel ellenőrizhetjük a nyelvtan egyes adattagjainak helyességét. Az első ilyen metódus azt ellenőrzi, hogy a példányt egyáltalán lehet-e nyelvtannak nevezni, tehát eleget tesz-e a szükséges követelményeknek. Ennek feltételeit a második fejezetben már részletesen tárgyaltam, így itt ennek részleteire nem térek ki. A másik metódussal kifejezetten a 3-típusú, azon belül is a jobblinéaris nyelvtanokra vonatkozó szabályok meglétét ellenőrizhetjük, tehát hogy az egyes átírási szabályok alakjai sorra a 3-típusú nyelvtanok esetén megszokottak-e. Ha csak egy szabály nem tesz eleget ennek a feltételnek, akkor a nyelvtanról azt mondhatjuk, hogy nem 3-típusú.

```

Console.WriteLine(G.IsRegular()); //Kimenet: True.

```

A *Grammar* osztály keretei közt készítettem el az egyik legfontosabb funkciót az egész művön belül, mellyel megvizsgálhatjuk bármilyen tetszőleges szó esetén, hogy az része-e a nyelvtan által generált nyelvnek, tehát a vizsgált szó levezethető-e az adott nyelvtan szabályain keresztül. A metódus elkészítése során nem csak a szóprobléma megoldásával foglalkoztam. Az algoritmust úgy készítettem el, hogy az konkrét megoldást is talál, tehát a szóprobléma szintaktikai elemzését is elvégeztem. A témával foglalkozó fellelt szakirodalmak kivétel nélkül az erős normálformában megadott reguláris nyelvtanok esetén, vagy ekvivalens véges automata megadásával vizsgálják a szóproblémát, ezzel lényegesen leegyszerűsítve a feladatot. Én úgy készítettem el az algoritmust, hogy az bármilyen jobblinéaris nyelvtan esetén működik, nem szükséges azt normál formára hozni, meg tudja állapítani a levezethetőséget, és amennyiben az lehetséges, ajánlást is ad a levezetésre. A következőkben röviden ismertetem az általam kifejlesztett algoritmus menetét.

Az elgondolásom az volt, hogy legyen egy szavak tárolására alkalmas H halmaz, mely kezdetben tartalmazza az adott nyelvtan kezdőszimbólumát, és legyen w bemeneti szó, amit le szeretnék vezetni a nyelvtan szabályain keresztül.

A program iteratív módon a következő folyamatot végzi. H minden eleme esetén, tehát kezdetben csupán a kezdőszimbólum kapcsán, megvizsgálja, hogy a felbontható-e két részszoóra oly módon, hogy a prefix szó csak terminális elemekből álljon, a szuffix szó pedig maximum egy nemterminális szimbólum legyen. Tehát ha az adott szó xB alakú, akkor a kapott két részszoó x és B legyen. Megengedve azt, hogy mind az x mind a B lehet λ . Könnyen belátható, hogy ez a felbontás mindig sikeres lesz, hiszen kikötöttük, hogy a nyelvtan legyen jobblinéaris.

A felbontás után eldönti, hogy a P átírási szabályok halmaza tartalmaz-e olyan szabályt, melyre igaz az, hogy annak bal oldalán a H halmaz vizsgált elemének B szuffixe található, jobb oldalán pedig egy olyan yA (vagy y , ha $A = \lambda$) alakú szó, melyre teljesül, hogy ha a w szóból levonjuk az x prefixet, akkor az így megmaradt részszoó valamekkora prefixe megegyezik az y szóval.

Ha talál ilyen szabályt, akkor egy (kezdetben üres) H' halmazba felveszi a xyA szót. Amikor H összes elemén végig ért, akkor H elemeit törli és felveszi a H' elemeit a H halmazba, majd a folyamatot kezdi előlről.

Mindezt addig ismétli, míg végül már nem talál H halmazban olyan elemet, amivel dolgozni tudna. Tehát a halmaz vagy üres lesz, mert H' halmazba végül nem tudott semmit felvenni, vagy H minden szava már egyenlő lesz a keresett w szóval, így további átírás már nem lehetséges. Ez utóbbi esetben talált helyes megoldást a levezetésre.

A működés során a program H minden eleméhez egy listát társít, melyben gyűjti az alkalmazott átírási szabályokat. Így végül ha H valamely eleme w szóval lesz ekvivalens, akkor az adott elemhez tartozó átírási szabályok rendezett listája fogja alkotni a levezetést. Így tehát az általam kifejlesztett algoritmus választ ad a levezethetőségre és helyes levezetési megoldást ad, amennyiben az lehetséges.

```
ListOfRules c1 = new ListOfRules();

Console.WriteLine(G.Matching("a", out c1)); //Az "a" szó levezethetőségének
vizsgálata. A levezetés, ha létezik, a c1 típusba kerül.
Console.WriteLine(c1); //Kimenet: Q->S->A->a.

Console.WriteLine(G.Matching("ba", out c1));
Console.WriteLine(c1); //Kimenet: Q->S->B->bQ->bS->bA->ba.
```

Két nagyon fontos metódust terveztem még az osztályhoz, melyek szintén nagy szerepet játszanak a feladat megoldásában. Ezek a láncszabályok eltávolítására szolgáló függvények, melyeket akkor alkalmazunk, amikor ekvivalens automatát szeretnénk konstruálni a nyelvtanhoz. Egyenként lehetőség van az $A \rightarrow B$, illetve az $A \rightarrow xB$ és $A \rightarrow x$ alakú szabályok eltüntetésére. Ezek algoritmizálását az elméleti alapok során tárgyalt módon végeztem el, ezért itt már nem részletezném.

```
Grammar G1 = new Grammar(N1, T1, P1, S1);

Console.WriteLine(G1); //Kimenet: ({S, A, B}, {a, b, c}, {S->abaB, A->B, A->cacb,
B->bA, B->S, B->\lambda}, S).
```

```
G1.deleteChainRules1(); //A->B alakú szabályok törlése.

Console.WriteLine(G1); //Kimenet: ({S, A, B}, {a, b, c}, {S->abaB, A->cacb, A->bA, A->λ, A->abaB, B->bA, B->, B->abaB}, S).

G1.deleteChainRules2(); //A->xB és A->x alakú szabályok törlése.

Console.WriteLine(G1); //Kimenet: ({S, A, B, C, D, E, F, G}, {a, b, c}, {S->aC, bC->bD, D->aB, A->cE, E->aF, F->cG, G->b, A->bA, A->λ, A->aC, B->bA, B->λ, B->aC}, S).
```

A szoftver egyik fő feladata az ekvivalens nyelvtanok és automaták megszerkesztése. Ennek érdekében készítettem el egy metódust a *Grammar* osztályon belül, ami tetszőleges bemeneti automata alapján elkészíti a vele ekvivalens nyelvtan objektumot. Ennek megvalósítását az elméleti fejezetben leírtak szerint végeztem el.

```
Machine M = new Machine(Q, I, d, q, F); //Automata objektum.
Grammar G = Grammar.ConvertFrom(M); //Ekvivalens nyelvtan megszerkesztése
automata alapján.
```

4.1.11. Machine osztály

Automata modell tárolására alkalmas osztály. A nyelvtani (*Grammar*) osztállyal ellentétben ezt az osztályt speciálisan a véges elfogadó automaták reprezentálására készítettem el. Ennek fő oka, hogy a különféle automata típusuk felépítése nagymértékben eltér egymástól, így feleslegesen bonyolította volna a feladatot egy olyan automata osztály leprogramozása, mely nem lesz teljes mértékben kihasználva a feladat során.

Ennek megfelelően az osztály halmazok formájában várja az egyes állapotokat, bemenő jeleket és elfogadó állapotokat. Valamint átmeneti függvények halmazaként (*SetOfTransitionFunction*) tárolja a szükséges függvényeket, és meg kell neki adni a kezdőállapotot is.

```
Set<String> Q = new Set<String>(); //Állapotok halmaza.
Set<String> I = new Set<String>(); //Bemenő jelek halmaza.
SetOfTransitionFunction d = new SetOfTransitionFunction(); //Átmeneti függvények
halmaza.
String q; //Kezdőállapot.
Set<String> F = new Set<String>(); //Végállapotok halmaza.

//Állapotok felvétele.
Q.Add("0");
Q.Add("1");
Add("2");

//Bemenő jelek felvétele.
I.Add("a");
I.Add("b");

//Átmeneti függvények felvétele.
d.Add("0", "a", "1");
d.Add("0", "a", "2");
d.Add("0", "b", "2");
d.Add("1", "a", "0");
d.Add("2", "b", "1");

q = "S"; //Kezdőállapot felvétele.
```

```

//Végállapotok felvétele.
F.Add("0");
F.Add("2");

Machine M = new Machine(Q, I, d, q, F); //Automata létrehozása.

Console.WriteLine(M); //Kimenet: ({0, 1, 2}, {a, b}, {d(0, a)={1, 2}, d(0,
b)={2}, d(1, a)={0}, d(2, b)={1}}, 0, {0, 2}).

```

Megvizsgálhatjuk, hogy a létrejött automata felépítése helyes-e, tehát kielégíti-e az automatákkal szemben támasztott általános követelményeket. Ezek az általunk vizsgált automata osztályt tekintve a második fejezetben részletezésre kerültek. Ugyanakkor készítettem metódusokat annak eldöntésére is, hogy a vizsgált automata determinisztikus-e, illetve parciális-e. Az automata determinisztikus voltát vizsgáló függvényem végig nézi annak összes átmeneti függvényét, és megvizsgálja, hogy azok hány átmenetet tartalmaznak. Amennyiben az átmenetek száma egy függvényre nézve több mint egy, vagy tartalmaz üres szó átmenetet, akkor azt mondhatjuk, hogy az automata nem determinisztikus. Amennyiben minden átmeneti függvény átment a rostán, az automata determinisztikus. Az automata parciális tulajdonságát vizsgáló metódusom képi az állapotok és a bemenő jelek összes lehetséges kombinációját, és mindegyik esetre megvizsgálja, hogy az átmeneti függvények halmaza tartalmaz-e az adott állapotra és bemenetre átmenetet. Az első olyan kombinációnál, ami hamis eredményt ad, a metódus megállapítja, hogy az automata parciális, azaz nem teljesen definiált.

```

Console.WriteLine(M.isCorrect()); //Kimenet: True.
Console.WriteLine(M.isDeterministic()); //Kimenet: False.
Console.WriteLine(M.isPartial()); //Kimenet: True.

```

Akárcsak a nyelvtan osztályban, itt is elkészítettem egy tetszőleges bemeneti szó felismerhetőségét elvégző metódust. A metódus szintén nem csak arra ad választ, hogy adott szó felismerhető-e vagy sem, hanem ajánlást is ad annak megoldásaként. Az algoritmust úgy terveztem meg, hogy az összes lehetséges levezetési módot szolgáltatassa.

A működés valamelyest hasonló mint a nyelvtanok esetén. Vettem egy H listát, mely konfigurációk listáját képes tárolni (*ListOfConfiguration*), és kezdetben hozzá adtam egy elemű konfigurációs listát, melynek kezdeti állapota legyen a q_0 kezdőállapot, olvasott szava pedig a vizsgált w szó.

Itt is iteráció formájában történik a megoldás. Az algoritmus végigmegy a H lista összes elemén és megnézi, hogy az aktuális konfigurációs lista már befejeződött e, tehát hogy a lista utolsó konfigurációjában az olvasott szó üres szó-e.

Amennyiben igen, akkor azzal nem foglalkozunk, áthelyezésre kerül a kezdetben üres, szintén konfigurációs listák tárolására képes H' listába. Azonban ha nem üres szó az utolsó konfigurációban olvasott szó, akkor az algoritmus megpróbál tovább haladni a felismerés felé.

Ennek első lépéseként megnézi, hogy a d átmeneti függvények halmazában van-e olyan átmenet, amivel tovább tud dolgozni. Akkor tud tovább haladni, ha létezik olyan átmenet,

melyre igaz, hogy a kiindulási állapota megegyezik a vizsgált konfiguráció állapotával, és a bemeneti jel pedig az adott konfigurációban olvasott szó első karaktere.

Amely átmeneti függvény megfelel ennek, azoknak az összes átmenete esetén felvesszünk egy új konfigurációs listát, mely legyen egyenlő az éppen vizsgált listával, és adjunk hozzá egy új konfigurációt, melyben az aktuális állapot legyen az adott átmeneti függvény vizsgált új állapota, az új állapotban olvasott szó pedig legyen az aktuális konfigurációs listában szereplő utolsó konfiguráció esetén olvasott szó olyan szuffixe, melyből csak az eredeti szó első karaktere hiányzik.

Az így kapott új konfigurációs listákat is hozzá adja H' listához. Ha H lista végére ért, akkor annak tartalmát üríti és betölti H' elemeit H listába. Az egész folyamatot kezdi előlről és mindaddig ismétli, míg H lista összes elemének utolsó konfigurációjának olvasott szava nem lesz üres szó.

Ha ez a feltétel teljesül a H listára, akkor meg kell vizsgálni, hogy tartalmaz-e elemeket. Amennyiben nem, w szó nem vezethető az automatával. Ha tartalmaz elemeket, akkor minden H listabeli konfigurációs listára meg kell nézni, hogy annak utolsó konfigurációjának aktuális állapota egyben végállapot is. Amelyre nem igaz, azt törli a H listából.

Ha H -nak még így is maradtak elemei, akkor a w szó vezethető és a megmaradt konfigurációs listák jelentik az összes lehetséges levezetését az adott szónak.

```
List<ListOfConfigurations> rv = new List<ListOfConfigurations>(); //Lehetséges
levezetések ide tárolja.

Console.WriteLine(m.Matching("aabbbaa", out rv)); //Kimenet: True.

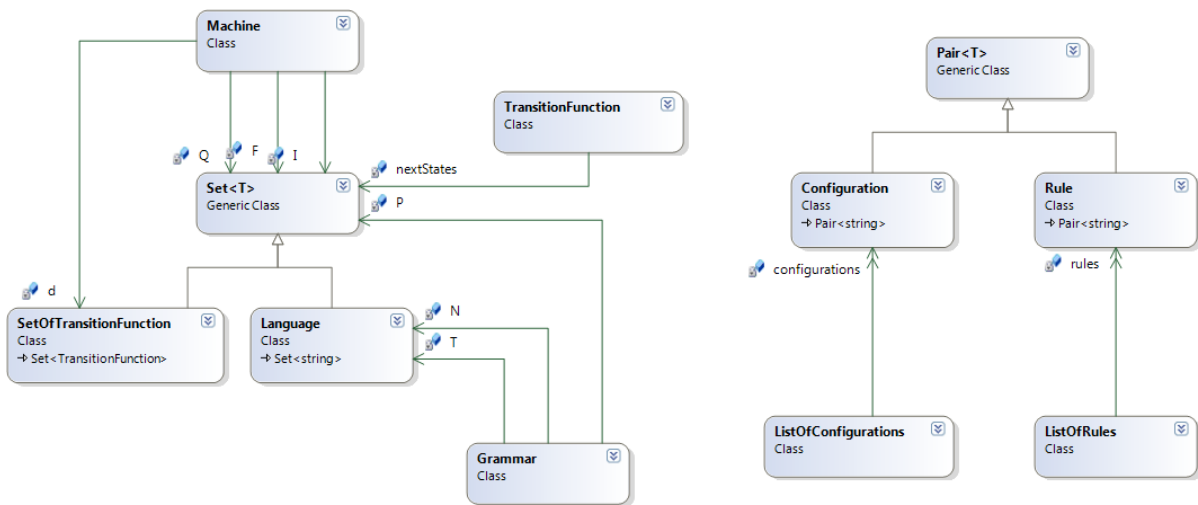
for (int i = 0; i < rv.Count; ++i) //Összes lehetséges levezetésnek a kiírása.
    Console.WriteLine(rv[i]);

//Kimenet: (0, aabbbaa)->(1, abbaa)->(0, bbaa)->(2, baa)->(1, aa)->(0, a)->(2, λ).
```

Akárcsak a nyelvtanok esetén, itt is megvalósítottam az ekvivalens 3-típusú nyelvtan megkonstruálásának lehetőségét. Ezt egy olyan függvény formájában hoztam létre, mely bemenetként vár egy *Grammar* objektumot, és kimenetként szolgáltatja a vele ekvivalens automatát.

```
Grammar G = new Grammar(N, T, P, S);
Machine M = Machine.ConvertFrom(G); //Ekvivalens automata megszerkesztése
nyelvtan alapján.
```

4.1.12. Osztály diagram



11. ábra: A matematikai és formális nyelvi alaposztályok UML diagramja

4.2. A grafikus reprezentáció megvalósítása

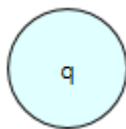
A 4.1.-es alfejezetben megvalósított alaposztályokra építkezve készítettem el a program grafikai megjelenéséért és használhatóságáért felelős típusokat. Az elkészítéshez a C# nyelv WPF (Windows Presentation Foundation) grafikus alrendszerét használtam fel. Választásom azért a WPF-re esett és nem a sokkal népszerűbb és könnyebben használható Windows Forms-ra, mivel ez a környezet számottevően hatékonyabban támogatja az olyan bonyolult grafikai elemek valós idejű megjelenítését, mint például a feladatban megvalósított dinamikus automata ábrák. A következőkben részletezem a grafikai osztályokat és azok alapvető működését, egymásra épülését.

4.2.1. State osztály

A State osztály feladata az automata egyes állapotainak megjelenítése. Az osztály egy példánya egy automata állapotot reprezentál, így tároltam benne az ahhoz szükséges adatokat.

Egy állapotot két fő összetevőből alkottam meg, ezek az állapot teste és felirata. A test a formális nyelvekben megszokott módon egy kör objektum, melyet a C#-ban megtalálható *Ellipse* osztály felhasználásával valósítottam meg és tároltam el az általam létrehozott State osztályban. Az *Ellipse* osztály segítségével tetszőleges méretű és színű ellipszis alakzatokat rajzolhatunk az ablak bármely pontjára. Az állapot felíratához a beépített *Label* osztályt használtam fel, mellyel tetszés szerinti pozícióra helyezhetünk formázott szöveget a képernyőn. A State osztályt úgy valósítottam meg, hogy a felhasználónak nem kell foglalkoznia sem az *Ellipse*, sem a *Label* példányok tulajdonságaival. A State osztály példányában csupán a kívánt pozíciót kell megadni, és ez alapján az osztály felhelyezi a megfelelő helyre az ellipszis objektumot, majd annak középpontjára a felíratot.

```
State s1 = new State(grid1, 30, "q", Brushes.LightCyan, 70, 100, false, false);
```



12. ábra: A State osztály egy példányának grafikus megjelenése

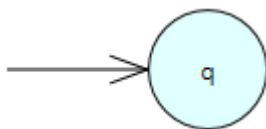
Az osztályt úgy valósítottam meg, hogy azzal nem csak statikus állapotokat lehet ábrázolni. Lehetőséget biztosítottam az egyes állapotok áthelyezésére is. A fellelhető automata modellek általában mind statikusak, ami olykor nehezítheti annak megértését. Az általam létrehozott modell ettől sokkal szemléletesebbé és átláthatóbbá vált.

Az egyes állapotok mozgatásához a programozási nyelv nyújtotta *eseménykezelést* használtam fel. Események segítségével olyan funkciókat rendelhetünk bizonyos objektumokhoz, melyek funkciók esetén kikötjük, hogy azok csak bizonyos esemény bekövetkezte után fussanak le. A mozgatást egy olyan metódussal hajtottam végre, mely meghívása esetén az adott állapot követi a kurzor mozgását. Ezt a metódust pedig csak akkor futatja le a program, ha bekövetkezik az az esemény, hogy a felhasználó az állapot fölé viszi a kurzort, tehát a kurzor pozíciója beleesik az állapot ellipszisének területébe és mindemellett nyomva tartja a bal egérgombot. Amíg ez az esemény tart, addig folyamatosan fut a metódus, tehát az állapot folyamatosan követi a kurzort. Amint az esemény leáll, a metódus sem fut tovább. Így könnyűszerrel és más programokból már jól megszokott, úgynevezett *drag-and-drop* (*fogd és vidd*) módszerrel lehet az automata grafikus megjelenését személyre szabni.

Annak érdekében, hogy az osztály minél hűbben visszaadja a valós automaták állapotát, megvalósítottam két fontos funkciót a *State* típuson belül. A formális nyelvi automaták állapotait két „kitüntetéssel” lehet ellátni. Az egyik az, hogy az állapot legyen kezdőállapot, a másik pedig, hogy az állapot legyen végállapot. Illetve a kettő nem zárja ki egymást, tehát egy állapot lehet kezdő és vég is. Ezt a két funkciót hoztam létre magam is az állapotokon belül.

Ha egy állapotot kezdőállapotként jelölünk meg, akárcsak a formális nyelvekben, itt is egy az állapotra bal oldalról vízszintesen mutató nyíl jelzi ezt. Ezt egy logikai érték ki- és bebillentésével lehet megjeleníteni és eltávolítani. A megjelenő nyíl grafikájáért az *Arrow* osztály felel, mely a következő alfejezetben kerül kifejtésre.

```
State s1 = new State(grid1, 30, "q", Brushes.LightCyan, 70, 100, false, true);
```



13. ábra: Kezdőállapot grafikus megjelenése

Végállapotot úgy különböztetünk meg, hogy annak kontúráját vastagabbra szedjük, mint az automata többi állapotáét. Ennek a megvalósítására gyakorlatilag kész megoldást kínál az *Ellipse* osztály, ugyanis a típuson belül megadható, hogy az adott ellipszis példány körvonala milyen vastag legyen. Így itt már csak annyi volt a dolgom, hogy egy szintén logikai érték változtatásával két vastagsági érték közül lehessen választani.

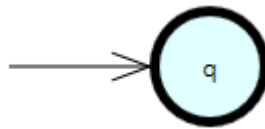
```
State s1 = new State(grid1, 30, "q", Brushes.LightCyan, 70, 100, true, false);
```



14. ábra: Végállapot grafikus megjelenése

Mint már szó volt róla, nem kizárt hogy egy állapot egyben kezdő- és végállapot is legyen. Erre is lehetőséget adtam az osztályon belül.

```
State s1 = new State(grid1, 30, "q", Brushes.LightCyan, 70, 100, true, true);
```



15. ábra: Kezdő- és végállapot grafikus megjelenése

4.2.2. Arrow osztály

Az Arrow osztály felel az automata egyes állapotai közti kapcsolatok grafikus megjelenítéséért. Ha egy automatóban két állapot kapcsolatban van egymással, tehát értelmezett köztük valamilyen irányú átmenet, akkor ezt egy az állapotokat összekötő nyíl segítségével szemléltetjük. Megjegyezném, hogy az állapotok összeköttetésére több jelölésmód is kialakult. Én azt a módszert választottam, melyben egy irányba csak egy nyilat veszünk fel, és ezen jelöljük az összes bemenő jelet, melyek hatására az átmenet létrejön. Ez egyrészt technikailag könnyebben kivitelezhető, mint az a megközelítés mely szerint ahány átmenet annyi nyíl legyen két állapot között, másrészt olyan automata esetén, ahol két állapot között igen nagyszámú átmenet is értelmezett, nagyságrendekkel megkönnyíti annak megértését és átlátását.

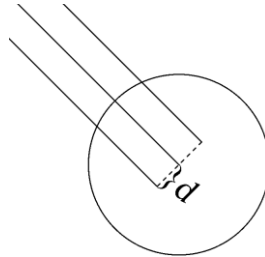
Fontos megjegyezni, hogy az osztály bár az automata állapotainak összeköttetésére jött létre, de ugyanakkor ettől függetlenül használható.

Alapvetően kétfajta nyíl megjelenítését készítettem el az osztállyal. Az első eset amikor egy q és w állapotot szeretnénk összekötni. Ekkor egy egyenes, egyik oldalán nyíllal ellátott szakaszt generál az osztály. A másik esetben egy q állapotot szeretnénk összekötni önmagával. Ilyenkor egy hurok reprezentálja a kapcsolatot. Ezért a két variáció megjelenítése nagymértékben eltér egymástól.

Az első esetben a C# beépített Line osztályát használtam fel, mellyel tetszőleges színű és vastagságú, két végpontjának koordinátaival adott szakaszt rajzolhatunk. Amikor két különböző állapot összeköttetéséről van szó, első lépésként az osztály generál egy szakaszt melynek egyes végpontjai a két állapot egyes középpontjai, így azokat a középpontjuk mentén összeköti.

Az osztály elkészítése során gondoskodtam arról az esetről, amikor két állapot között mindkét irányba van átmenet. Ilyenkor két párhuzamos, ellentétes irányú nyíl jelenik meg a két

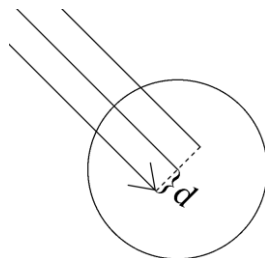
objektum között. Emiatt nem jó az a megközelítés, hogy a két *State* objektum a középpontjuknál fogva legyenek összekötve, hisz így a két *Arrow* típus fedné egymást, mellyel olyan hatást keltene, mintha egy darab két végén nyilazott szakasz lenne a két állapot között. Ezért az összekötést úgy valósítottam meg, hogy az ne a középpontokba történjen, hanem egy a középponttól vett olyan d pont távolságban, hogy a középpont és d pont által meghatározott szakasz merőleges legyen az eredeti középpontos összeköttetésre. Ily módon, ha két állapot között mindkét irányba van kapcsolat, akkor egy d eltolású és egy $-d$ eltolású nyíl objektummal grafikailag is látványos és érthető megoldást produkál a program.



16. ábra: *State objektum a középpontjába rajzolt szakasszal és annak d és $-d$ irányú merőleges eltolásaival*

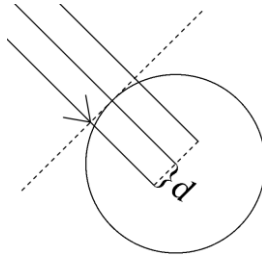
A megjelenítés másik fontos eleme az irány jelölése. Egy-egy logikai változó segítségével valósítottam meg a nyíl két végének ki-be kapcsolását. Ugyan a feladat szempontjából nem szükséges és hibás jelölés is lenne, de mindazonáltal az osztály képes mindkét végén nyilazott szakasz megjelenítésére. A nyíl hozzáadását úgy valósítottam meg, hogy vettem a már felrajzolt összekötő *Line* objektum két végpontját, ezeket jelöljük i és j pontokkal. A nyíl rajzoló metódus elforgatja az $(i; j)$ szakaszt először i pont körül x fokkal. Majd veszi az így kapott új szakasz egységvektorának l -szeresét. Ez után elvégzi ugyanezt a műveletet, de most $-x$ fokkal elforgatva, így megkapja a nyilat felépítő két *Line* objektum koordinátáit. Végül az egész műveletet megismétli az összekötő szakasz másik felén is. Ekkor nyilván az $(i; j)$ szakaszt most j pont körül forgatja. A transzformációt szintén a megadott x és l értékekkel végzi. Természetesen x és l értékei, tehát a nyilak által bezárt szög és a nyilak hossza szabadon változtatható.

Ekkor még fent áll az a probléma, hogy a nyilak ugyan rajta lesznek az $(i; j)$ szakasz megfelelő végein, de ezek az adott állapotok belsejébe mutatnak, a cél pedig az lenne, hogy a nyilak az állapotokat reprezentáló ellipszis objektumok szélén legyenek.



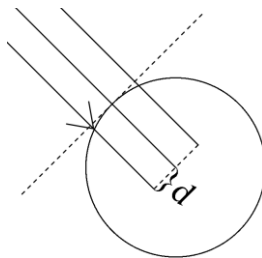
17. ábra: *Az eddig elgondolással a megjelenített nyíl az állapot belsejében található*

Ehhez először is el kell tolni az adott $\pm d$ értékkel eltoló $(i; j)$ szakasz adott (i vagy j) végpontján lévő nyilat r értékkel az $(i; j)$ szakasz mentén, ahol r az adott állapotot reprezentáló ellipszis sugara. Ekkor a nyíl kikerül a kör belsejéből, láthatóvá válik.



18. ábra: A kör sugarával eltolt nyíl rossz pozícióban

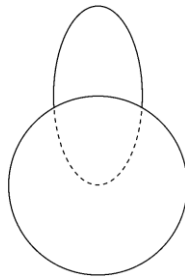
Viszont könnyen belátható, hogy $d > 0$ esetben nem az állapot szélén lesz a nyíl alakzat, hanem attól kicsit távolabb. Ennek érdekében úgy írtam meg az algoritmust, hogy az nem r értékkel tolja el a nyilat, hanem a félkör egyenletébe behelyettesítettem a d eltolási tényezőt, és az így kapott értékkel toltam el a nyíl pozícióját. Így a nyíl minden esetben az állapot objektum szélét fogja érinteni.



19. ábra: A pozíció korrigálásával a nyíl a kör szélére kerül

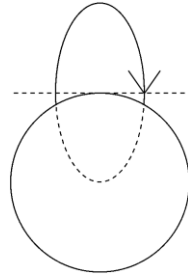
A kapcsolat megjelenítésének másik módja, amikor a két állapot ugyanaz, tehát hurok megjelenítésére van szükség. Itt a kapcsolat megjelenéséhez nem a *Line*, hanem a már szintén említett *Ellipse* osztályt használtam fel.

Ezt úgy valósítottam meg, hogy az osztály generál egy olyan ellipszis objektumot, melynek szélessége legyen r , magassága pedig legyen $2r$, ahol r az adott állapot sugara. A hurokot megvalósító ellipszis középpontja kezdetben legyen $(x; y - r)$, ahol $(x; y)$ az állapot középpontja.



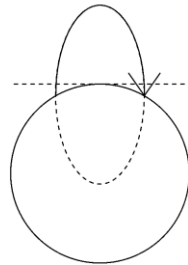
20. ábra: A kezdeti hurok megjelenése

Az algoritmusom ez után felhelyezi a nyilat a hurok egyik végére. Kezdetben ennek pozíciója legyen $(x + \frac{r}{2}; y - r)$.



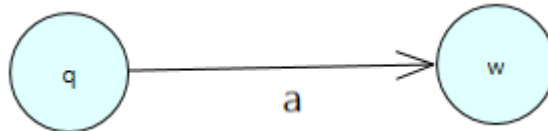
21. ábra: Nyíl megjelenítése korrekció nélkül

Jól látható, hogy a nyíl a hurokhoz képest megfelelő helyen van, viszont az állapothoz képest már nem. Ezért az volt az elgondolásom, hogy a mind a hurok mind a nyíl kerüljön lejjebb az y tengelyen annyival, hogy a nyíl pont az állapot ellipszisének szélét érintse. Itt ugyanazt a módszert alkalmaztam, mint a $q - w$ összekötés esetén. Vettem a félkör függvényét, és behelyettesítettem jelen esetben az $\frac{r}{2}$ értéket, hiszen az x tengelyen ennyivel toltam el a nyíl ábráját. Majd az így kapott számot adtam hozzá a nyíl és a hurok pozíciójának y értékéhez.



22. ábra: Hurok és nyíl a megfelelő pozíción

Az *Arrow* példányhoz tartozik még egy *Label* típus is, mely az adott átmenethez tartozó bemenő jeleket tartalmazza és megjeleníti mindig a nyíl közepén, vagy hurok esetén annak „tetején”.



23. ábra: A q állapotból a w állapotba a bemenő jel hatására jutunk



24. ábra: A q állapotból önmagába jutunk b vagy c bemenő jel hatására

4.2.3. MachineElement osztály

Az osztályt azért hoztam létre, hogy segítségével egyszerűen lehessen tárolni az egyes automata állapotok közti kapcsolatokat. Így elsősorban menedzselési feladatokat szántam a típusnak, de ugyanakkor szerves részét képezi a megjelenítésnek.

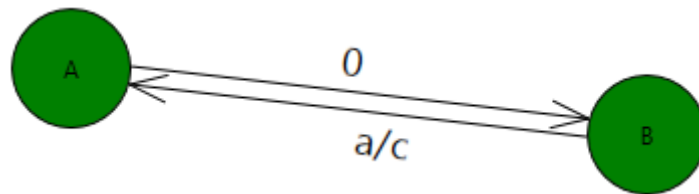
Az egyik nagyon fontos funkciója az osztálynak, amit megvalósítottam, az a dinamikus kapcsolat megteremtése az állapotok és a hozzá tartozó kapcsolatok (*Arrow*) között. Mint már

korábban szó volt róla, az automátákat úgy valósítottam meg, hogy azok grafikai reprezentációja dinamikus, tehát a síkban mozgathatóak az egyes állapotok. Adott a probléma, hogy ha elmozdítok egy állapotot, amibe mutat egy *Arrow* típus, akkor célszerű lenne, ha az *Arrow* példány megfelelő végpontja is követné a *State* objektumot. Ezt a feladatot látja el a *MachineElement* osztály, melyhez ismét az eseménykezelés módszerét használtam fel. Az osztály egy olyan eseményt generál az adott kapcsolathoz tartozó két *State* objektumra, mely megvalósítja azt, hogy a mozgásba jön valamelyik állapot, akkor a tartalmazott *Arrow* típusok megfelelő végpontjai követik a *State* objektum mozgását. Így a nyilak mindig a megfelelő állapotokba fognak mutatni.

```
State s1 = new State(grid1, 30, "A", Brushes.Green, 70, 100, false, false);
State s2 = new State(grid1, 30, "B", Brushes.Green, 350, 130, false, false);

MachineElement me1 = new MachineElement(s1, s2, grid1); //Paraméterként kapja a
kapcsolatban szereplő két állapotot és a felületet, ahova rajzoljon.

me1.AddArrow("B", "A", "a"); //"A"-ból "B"-be legyen "a" átmenet.
me1.AddArrow("B", "A", "c"); //"A"-ból "B"-be legyen "c" átmenet.
me1.AddArrow("A", "B", "0"); //"B"-ből "A"-be legyen "0" átmenet.
```



25. ábra: Az *me1* nevű *MachineElement* megjelenése

4.2.4. *MachineDrawer* osztály

Feladata a teljes automata megjelenítés koordinálása. Tartalmazza a szükséges *State* és *MachineElement* típusokat. Fő feladata a grafikus reprezentáció megszerkesztése egy bemenetként adott *Machine* típus példánya alapján. Az osztály a megjelenítés megtervezése során az egyes automata állapotok síkbeli koordinátáinak meghatározásakor alapvetően véletlenszerűen választ pozíciókat, de figyelembe veszi a már lehelyezett állapotokat. Így egymást fedve vagy túl közel egymáshoz nem helyez állapotokat. Tehát véletlenszerű, de nem egyenletes eloszlású a pozíciók sorsolása. Ekkor még fent áll az a probléma, hogy az állapotok közti kapcsolatok függvényében átláthatóság szempontjából jobb vagy rosszabb eredményt kaphatunk a sorsolt pozícióktól függően. Sajnos ennek megoldása jelenleg is egy elég komplex kutatási terület. Ennek megoldása meglehetősen összetett probléma, erre a munkám folytatásában látok lehetőséget.

```
Set<String> Q = new Set<string>();
Q.Add("A");
Q.Add("B");
Q.Add("C");

Set<String> I = new Set<string>();
I.Add("0");
I.Add("1");

SetOfTransitionFunction d = new SetOfTransitionFunction();
d.Add("A", "0", "B");
```

```

d.Add("A", "1", "C");
d.Add("C", "1", "C");
d.Add("C", "1", "B");

String q = "A";

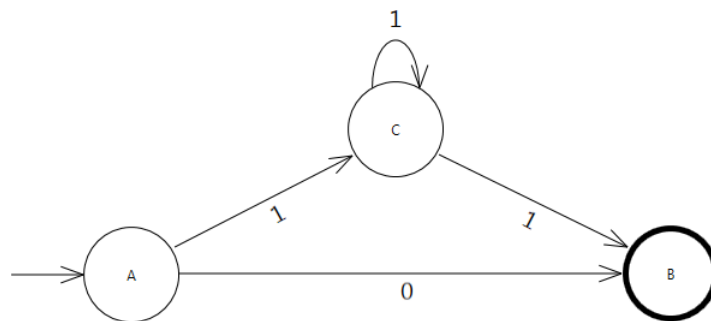
Set<String> F = new Set<string>();
F.Add("B");

Machine m = new Machine(Q, I, d, q, F);

MachineDrawer md1 = new MachineDrawer(grid1); //Paraméterként kapja a felületet,
ahova rajzolni fog.

md1.BuildMachine(m); //A grid1 felületen megjeleníti a paraméterként kapott
Machine típust.

```



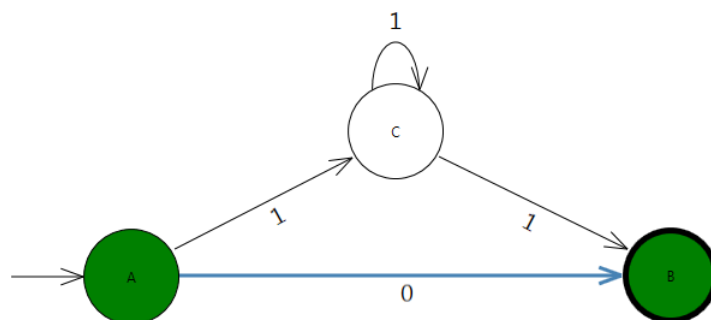
26. ábra: Az automata grafikus megjelenése

Az automata felismerésének végig követésére lehetőséget biztosítottam az osztályon belül arra, hogy az egyes kapcsolatok kiemelhetőek legyenek. Ezek egymás utáni kombinációjával jól bemutatatható egy adott bemeneti jelnek a lefutása.

```

md1.ShowConfig("A", "B"); //Az "A"-ból a "B"-be vezető kapcsolat kiemelése.

```



27. ábra: Automata megjelenése átmenet kiemeléssel

4.2.5. GrammarDrawer osztály

A *GrammarDrawer* típus felel a nyelvtanok látványos megjelenítéséért. Ehhez a C# beépített *RichTextBox* típusát használtam fel, mely segítségével formázott szöveget lehet a képernyőn megjeleníteni.

Az osztálynak a rajzfelületet és a megfelelő *Grammar* típust kell a rendelkezésére bocsátani, ez alapján elkészíti a *RichTextBox* objektumot. Lehetőséget adtam a pozíciójának változtatására is.

```

Language N = new Language();
N.Add("A");
N.Add("B");
N.Add("C");

Language T = new Language();
T.Add("0");
T.Add("1");

Set<Rule> P = new Set<Rule>(); ;
P.Add(new Rule("A", "0B"));
P.Add(new Rule("A", "1C"));
P.Add(new Rule("C", "1C"));
P.Add(new Rule("C", "1B"));
P.Add(new Rule("B", ""));

String S = "A";

Grammar G = new Grammar(N, T, P, S);

GrammarDrawer gd = new GrammarDrawer(grid1, G); //A rajzfelület és a nyelvtan
átadása.
gd.SetPos(100, 100); //pozíció megadása a rajzfelületen.
gd.Show(); //Megjelenítés.

```

{A, B, C}, {0, 1}, {A→0B, A→1C, C→1C, C→1B, B→λ}, A)

28. ábra: A megadott nyelvtan grafikus megjelenése

A *MachineDrawer* osztályhoz hasonlóan itt is megvalósítottam egy függvényt, amivel tetszőleges szabályt ki lehet emelni a nyelvtanból. Ezzel egy szó felismerése során alkalmazott szabályok végig követését lehet szemléletessé tenni.

```
gd.LightOnRule(new Rule("C", "1B")); //Kiemeli a "C->1B" szabályt, ha létezik.
```

{A, B, C}, {0, 1}, {A→0B, A→1C, C→1C, C→1B, B→λ}, A)

29. ábra: Nyelvtan megjelenése szabálykiemeléssel

4.2.6. RulesDrawer osztály

Az osztállyal azt a célt értem el, hogy tetszőleges szó sikeres levezetése esetén, megjelenítse az levezetés során használt egyes átírási szabályok következtében kapott szavakat. Ehhez szüksége van az adott *GrammarDrawer* típusra, a nyelvtanra, mellyel a vizsgálatot végezzük, illetve a rajzfelületre. Valamint meg kell adni a pozícióját, ahol a megjelenítést fogja végezni.

Miután elvégeztük a levezetést a *Grammar* osztály segítségével, a kapott levezetési listát kell átadni a *RulesDrawer* objektumnak, amely megjeleníti azt a grafikus felületen.

```

RulesDrawer rd = new RulesDrawer(grid1, gd, G, 200, 200); //Szükséges paraméterek
és a pozíció átadása.
ListOfRules lor = new ListOfRules();
G.Matching4("111", out lor);
rd.SetRules(lor); //Az "111" szóra vonatkozó levezetési lista átadása.
rd.Show(); //Megjelenítés.

```

A
1C
11C
111B
111

30. ábra: A megadott szó levezetésének megjelenése

A levezetés nyomon követéséhez megvalósítottam egy olyan eseménykezelésre épülő funkciót az osztályon belül, amivel az egyes átírások között megvizsgálhatjuk, hogy az adott nyelvtan mely szabályát alkalmazta az osztály. Ezt úgy készítettem el, hogy ha a felhasználó a kurzort valamely közbülső szó fölé viszi, akkor a megadott *GrammarDrawer* típus kiemeli az adott helyen használt átírási szabályt, a *RulesDrawer* típus pedig kiemeli az átírt szuffixeket.

$(\{A, B, C\}, \{0, 1\}, \{A \rightarrow 0B, A \rightarrow 1C, C \rightarrow 1C, C \rightarrow 1B, B \rightarrow \lambda\}, A)$

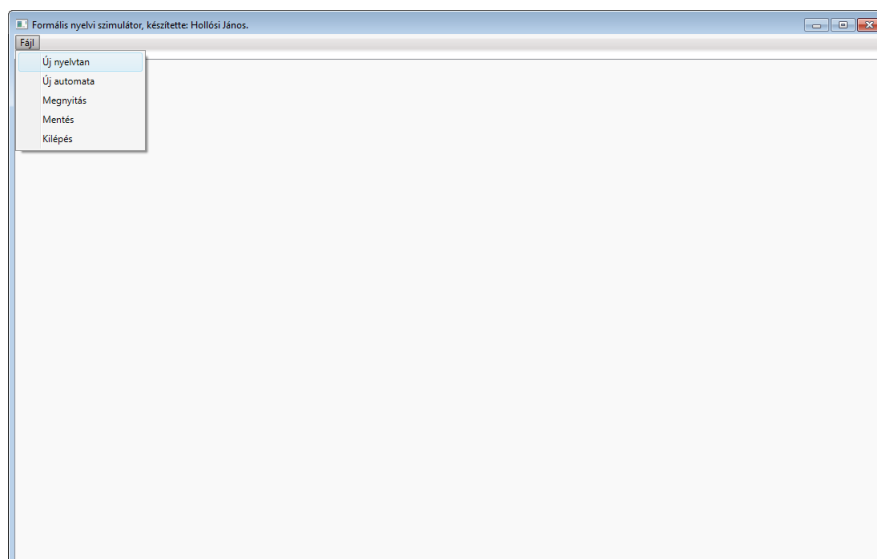
A
1C
11C
111B
111

31. ábra: Egyes átírások kiemelése a kurzor segítségével

4.3. A szoftver használata

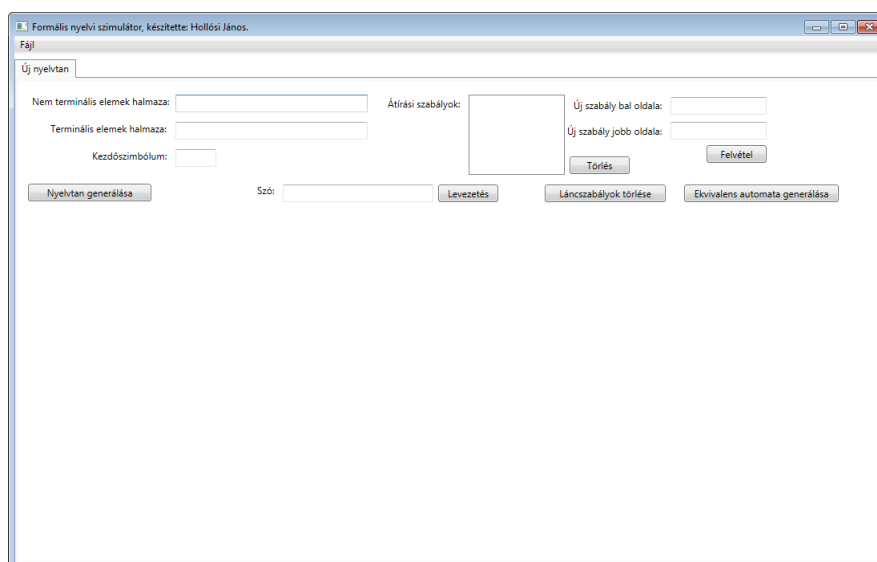
A következőkben bemutatom a már elkészült szoftver használatát. Rövid példákon keresztül szemléltetem az egyes fő funkciók működését.

A szoftver futtatása után annak üres kezdőképernyője jelenik meg, ahol a Fájl menüpontra kattintva választhatjuk ki, hogy mit szeretnénk végezni. Itt tudjuk kiválasztani, hogy új automatát vagy nyelvtant szeretnénk modellezni.



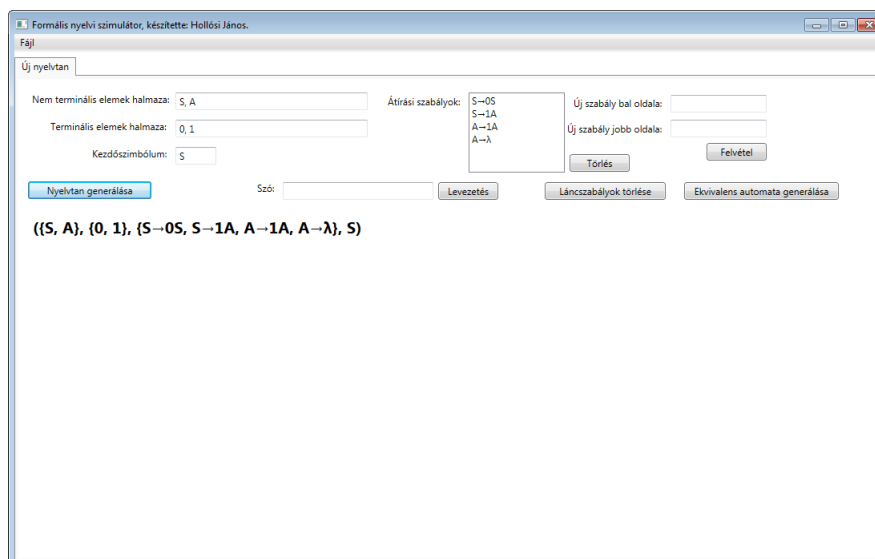
32. ábra: A szoftver kezdőképernyője és a Fájl menü tartalma

Miután kiválasztottuk az új nyelvten lehetőséget, a munkaterületen megjelenik egy új fül a nyelvten számára, ahol a nyelvten szerkesztéséhez szükséges felületet találjuk.



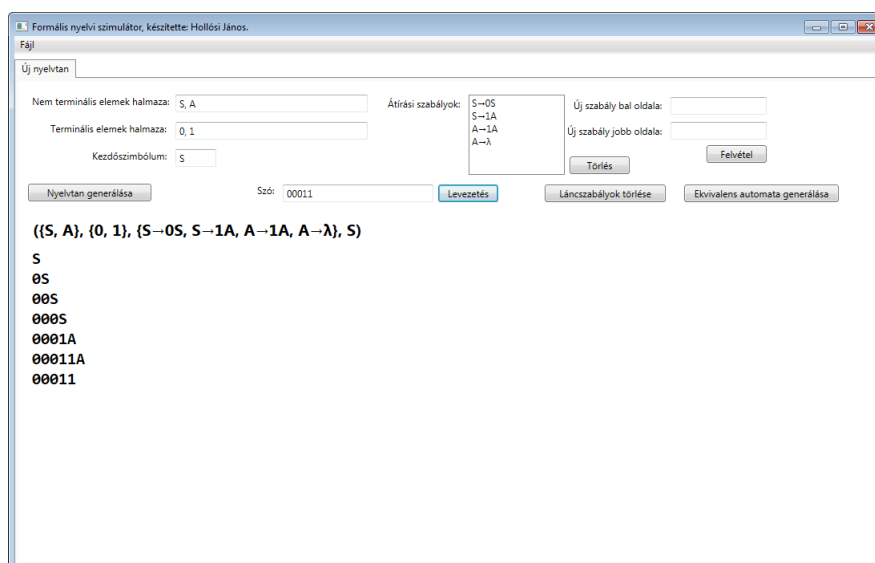
33. ábra: Nyelvten szerkesztőfelület

Egyszerű beviteli mezők segítségével megadhatjuk a nyelvten egyes alkotóelemeit. Ha felvittük az adatokat, akkor generálhatunk belőle nyelvten, melyet a megszokott formában jelenít meg a szoftver.



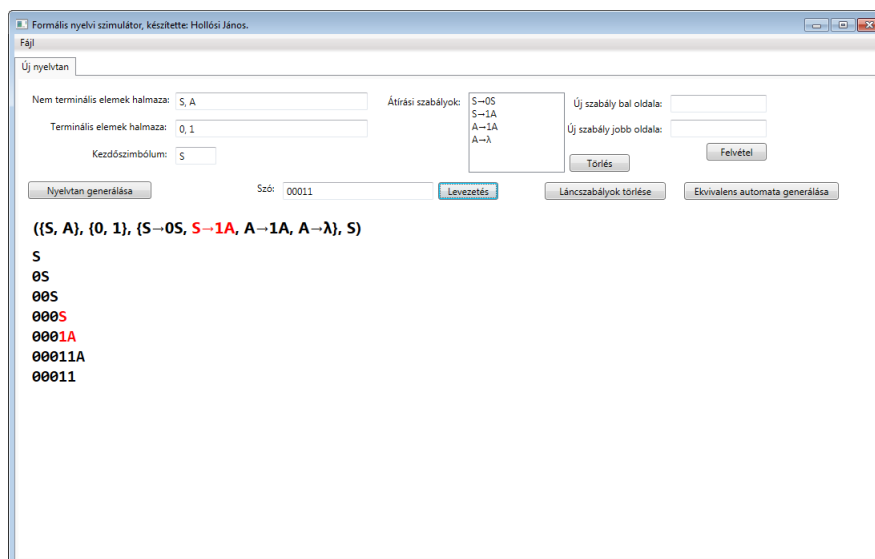
34. ábra: Nyelvten generálása

A nyelvten megalkotása után lehetőség van tetszőleges szó levezethetőségének vizsgálatára. Ehhez a kívánt szót a megfelelő beviteli mezőbe kell írni, majd a Levezetés nyomógomb segítségével kiértékeli azt a szoftver. Amennyiben a levezetés lehetséges, láthatóvá válnak annak lépései. Ha a szó nem eleme a nyelvten által generált nyelvnek, akkor erről tájékoztat a szoftver.



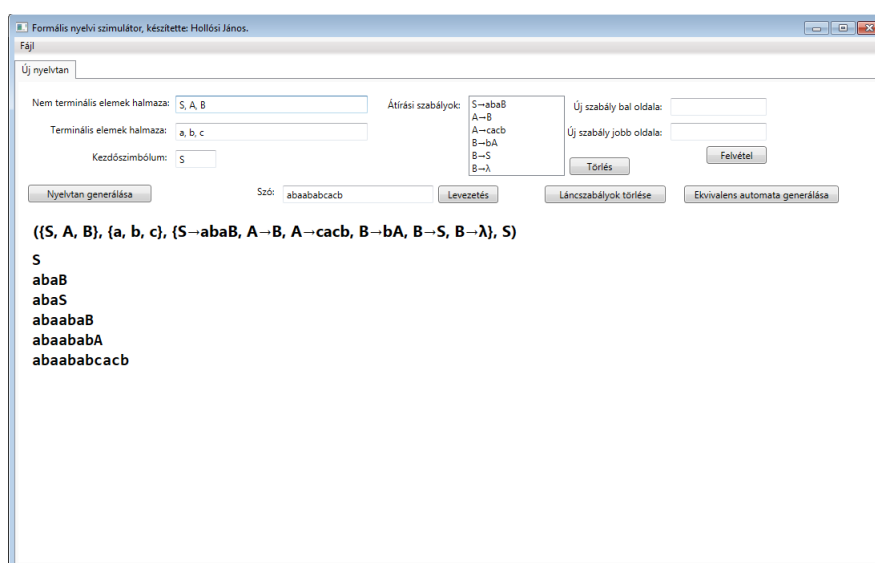
35. ábra: Szó levezetés lépései

A szoftver ekkor mutatja, hogy a kezdőszimbólumból kiindulva milyen közbülső szavakon keresztül jutunk el a kívánt szóig. Ha meg akarjuk tekinteni, hogy az egyes szavak között mely átírási szabályt alkalmazta a szoftver és azok hatására a szavak mely részeit írta át, a kurzort vigyük a megtekinteni kívánt szó fölé. Ekkor piros színnel jelzi a használt szabályt és az átírt részsavakat.



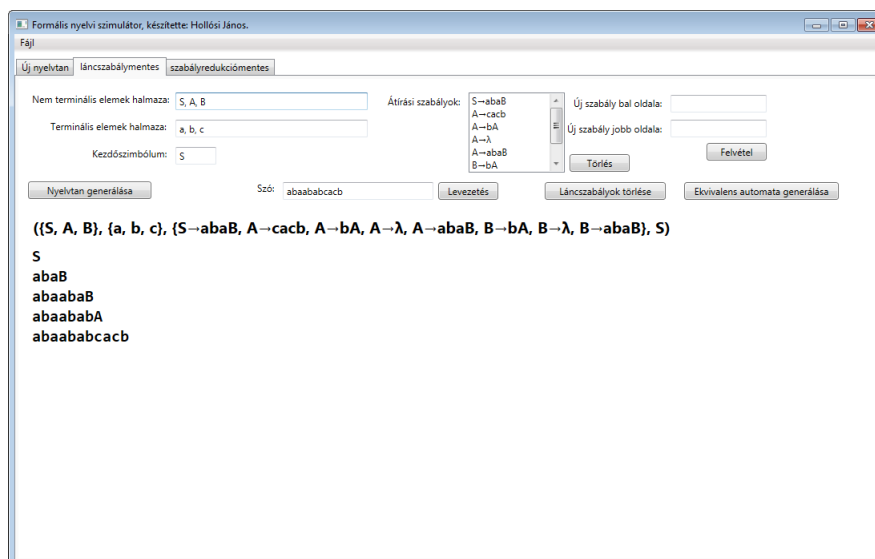
36. ábra: Átírások nyomon követése

Lehetőség van a láncszabályok és szabályredukciók vizsgálatára. Ha ezt az opciót választjuk, akkor a szoftver létrehoz két új nyelvten modellt. Az elsőből törli a láncszabályokat, a másodikban pedig a szabályredukciókat is elvégzi. Ehhez nézzünk az alábbi összetettebb nyelvtant.

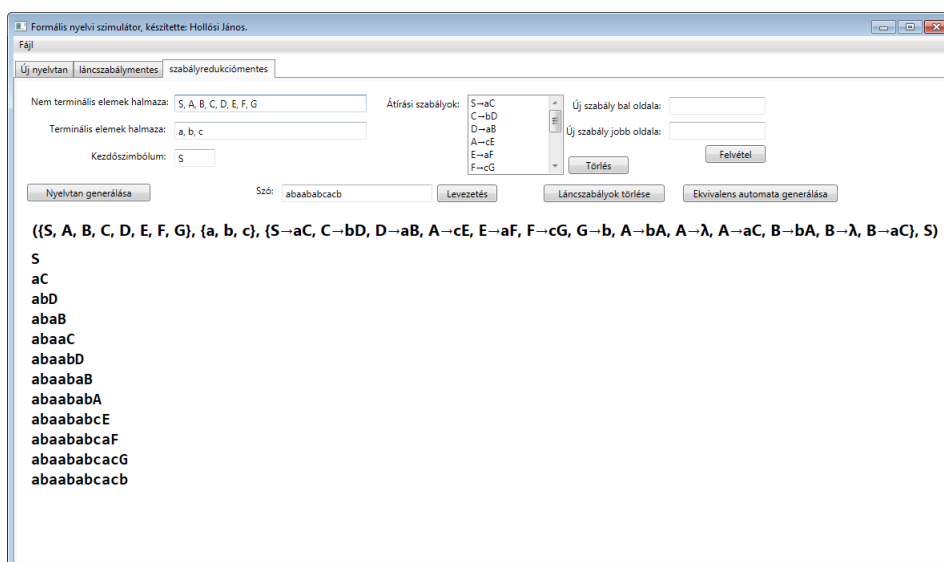


37. ábra: Komplex nyelvten modell

Jól látható, hogy a nyelvtenban több láncszabály és $A \rightarrow xB$ alakú szabály is szerepel. A láncszabályok törlése nyomógomb segítségével a szoftver automatikusan előállítja a láncszabály mentes nyelvten és elvégzi a szabályredukciót.

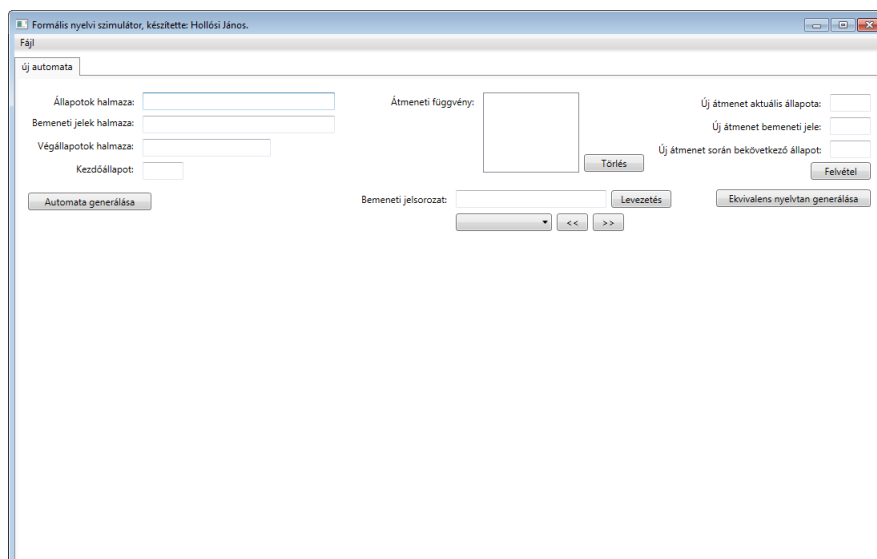


38. ábra: Láncszabály mentes nyelvten



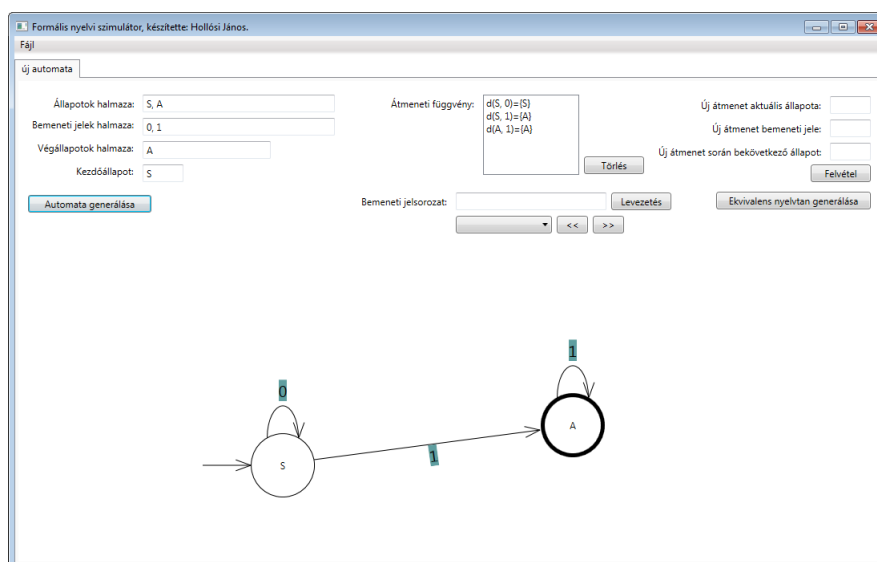
39. ábra: Nyelvten szabályredukció után

Lehetőség van automata modellezésére is. Ehhez a Fájll menüből most az Új automata menüpontot kell kiválasztani. Ekkor megjelenik egy új fül az automata számára, ahol a szerkesztéshez szükséges felületet találjuk.



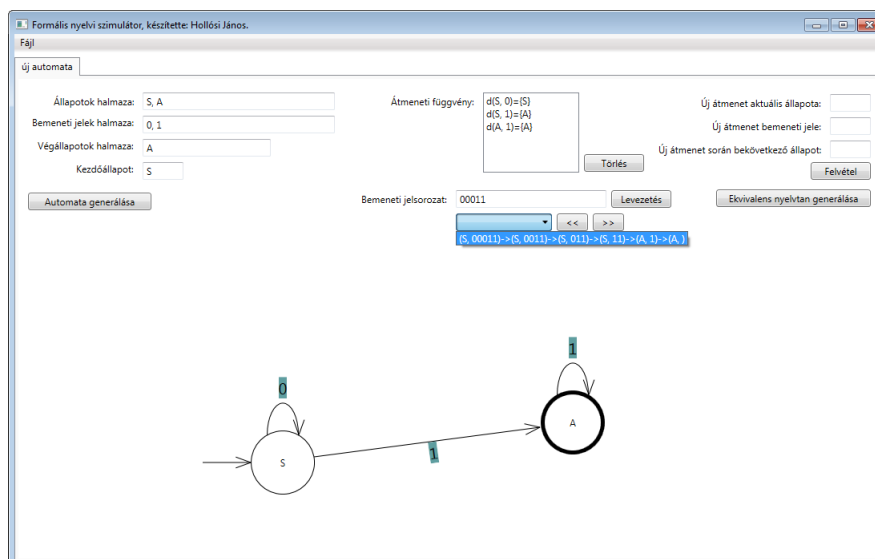
40. ábra: Automata szerkesztő felület

Ez után akár csak a nyelvtannál, itt is felvihetjük az automatahoz tartozó adatokat. Az adatok megadása után legenerálhatjuk a kívánt automátát, amit a szoftver grafikusán is megjelenít.



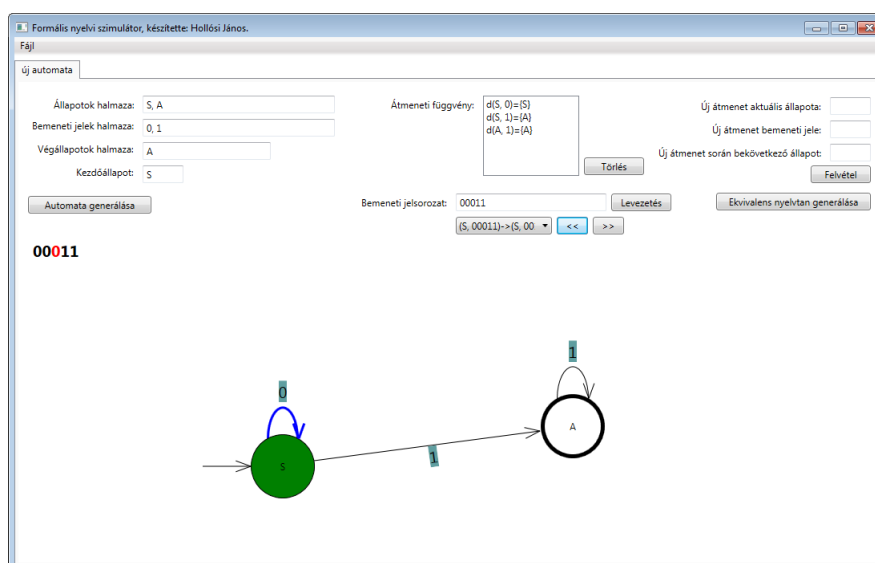
41. ábra: Automata generálás

Végül megadhatunk egy bemeneti jelsorozatot, melynek felismerhetőségét vizsgálhatjuk. Amennyiben az automata felismeri a jelsorozatot, megadja az összes lehetséges felismerési módot, amik közül egy legördülő listából választhatunk. Jelen esetben csak egy ilyen mód van.



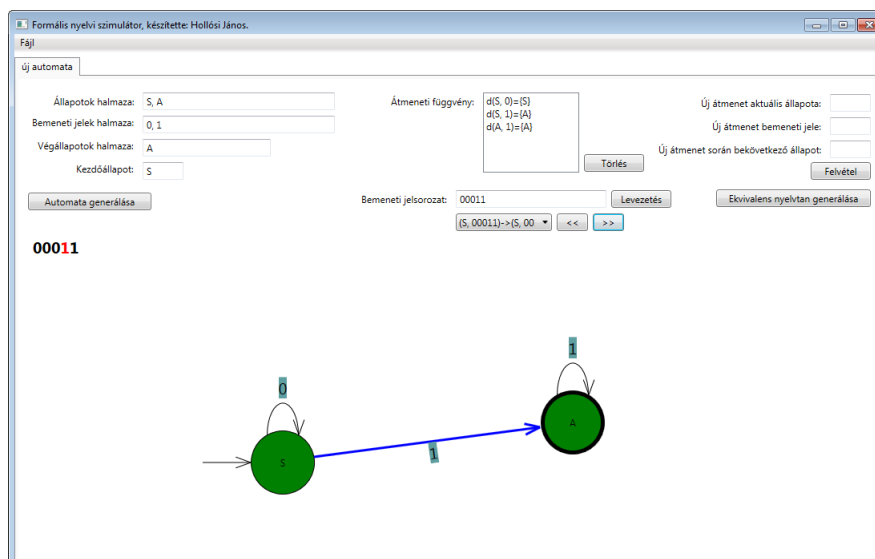
42. ábra: Lehetséges felismerési útvonalak megjelenítése

Ha ezek közül kiválasztunk egyet, akkor a legördülő lista melletti két nyomógomb segítségével lépkedhetünk előre is vissza a felismerés egyes lépései között. Minden lépésben kiemeli a szoftver az éppen használt átmenethez tartozó állapotot vagy állapotokat, illetve a bemeneti szóban éppen felismert karaktert is. Az alábbi képen a bemeneti „00011” jelsorozat harmadik karakterének felismerése zajlik. Látható, hogy a felismerésben csak az „S” állapot vesz részt, ahol a „0” jel hatására ismét „S” állapotba kerül az automata.



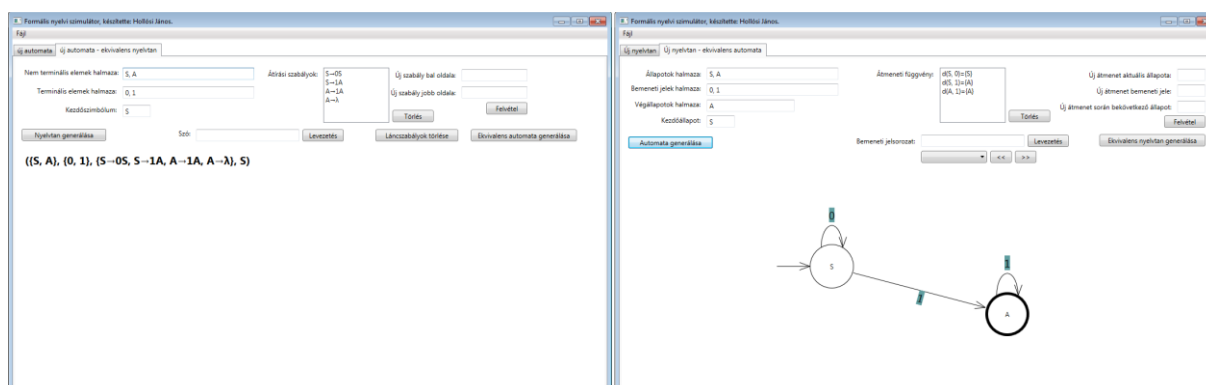
43. ábra: A bemeneti szó harmadik karakteréhez tartozó átmenet

Ha a következő lépésre ugrunk, akkor a negyedik karakter felismerése látható, mely során az „S” állapotból az „A” állapotba kerül az automata.



44. ábra: A bemeneti szó negyedik karakteréhez tartozó átmenet

Mind a nyelvtanok, mind az automaták esetében lehetséges az ekvivalens automata és nyelvtan megkonstruálása. Ehhez a panel megfelelő gombjára kattintva a szoftver legenerálja az ekvivalens modellt és egy új fülön megjeleníti azt egy önálló modellként, amivel akár tovább dolgozhatunk.



45. ábra: Ekvivalens nyelvtan és automata generálása

5. Összehasonlítás

Az alábbi fejezetben összehasonlítom az általam elkészített szoftvert és a már tárgyalt JFLAP alkalmazást. A tárgyalás során kitérek a JFLAP általam érzékelt hiányosságaira és bemutatom, hogy véleményem szerint az én szoftverem miért alkalmasabb oktatási felhasználásra.

Nyelvtan konstruálása esetén a JFLAP nem az elméletben és a legtöbb szakirodalom alapján használatos jelölésmódot alkalmazza. Ennek következményeként a létrehozott nyelvtan objektum nem egyértelmű. Nem lehet egyértelműen megadni a terminális és nemterminális elemek halmazát, nem lehet kijelölni kezdőszimbólumot. A nyelvtan megadása kimerül az átírási szabályok felvitelében. A szoftver kiköti, hogy az első szabály bal oldala legyen a kezdőszimbólum, továbbá az angol ábécé nagybetűi a nemterminális elemek, míg a számok és az angol ábécé kisbetűi a terminális elemek. Ugyan a legtöbb szakirodalom az elmélet bemutatása során alkalmazza ezt a megkülönböztetési módot, de mindazonáltal ez nem triviális és jelentősen szűkíti a megvalósítható nyelvtanok halmazát. Ezzel szemben az általam megvalósított alkalmazásban egyértelműen és tetszőlegesen felvehetőek a nyelvtan létrehozásához szükséges terminális és nemterminális elemek, a kezdőszimbólum és az átírási szabályok.

LHS		RHS
S	→	aaA
S	→	bbB
A	→	b
B	→	λ
B	→	c

Nem terminális elemek halmaza: S, A, B

Terminális elemek halmaza: a, b, c

Kezdőszimbólum: S

Átírási szabályok: S→aaA, S→bbB, A→b, B→λ, B→c

Nyelvtan generálása

Szó:

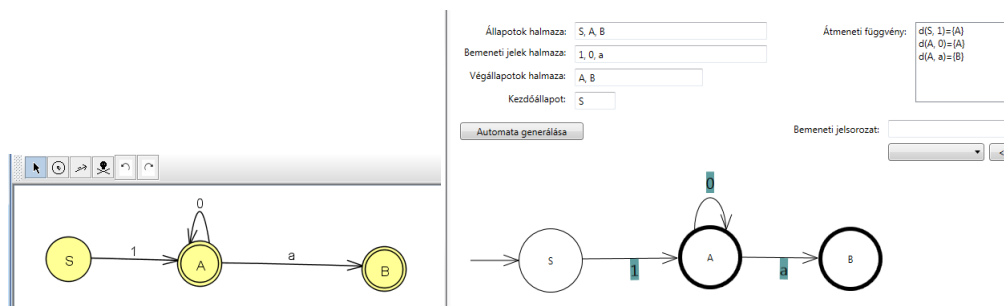
Levezetés

Li

((S, A, B), {a, b, c}, {S→aaA, S→bbB, A→b, B→λ, B→c}, S)

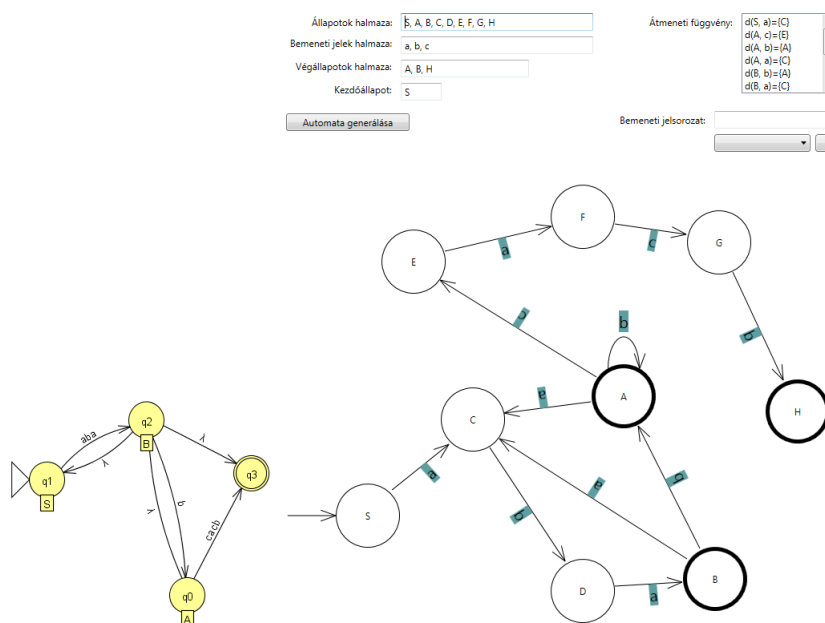
46. ábra: Nyelvtan szerkesztés

Automaták generálása esetén a JFLAP csak grafikus módot ajánl fel. Nincs lehetőség a formális nyelvi elméletben megszokott jelölésmódok használatára. Nem adható meg az automata annak tulajdonságaival, úgy, mint állapotok halmaza, bemenő jelek halmaza, átírási függvény, végállapotok halmaza és kezdőállapot. Komplex automata szerkesztése esetén ez problémát jelenthet, hiszen ha az automata elég sok állapotot tartalmaz vagy az állapotok közti kapcsolat elég összetett, az automata nem feltétlen szerkeszthető meg a síkban úgy, hogy az kellőképp átlátható legyen. Ez esetben komoly probléma a formális megadás hiánya. Ezen felül nem ad lehetőséget a hallgatóknak arra, hogy formálisan megadott automata grafikus reprezentációjának elkészítését gyakorolják. Az én szoftverem segítségével a szakirodalmakból elsajátított jelölésmód segítségével adható meg az automata annak minden tulajdonságával együtt. A szoftver ezek alapján maga generálja le az automatahoz tartozó grafikus reprezentációt, gondosan ügyelve arra, hogy az egyes állapotok ne fedjék egymást a képernyőn.



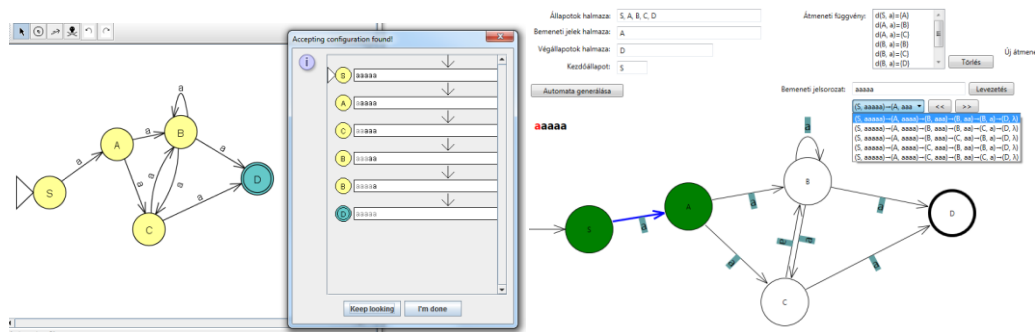
47. ábra: Automata szerkesztés

A szabályredukciók vizsgálatával nem foglalkoztak a JFLAP fejlesztői. Ez több szempontból is problémás. Az elméletet tanuló hallgatók a JFLAP segítségével nem tudják a láncszabályok törlését és szabályredukciókat vizsgálni, így a szoftver az ilyen jellegű gyakorlati feladatok megoldására és ellenőrzésére nem hasznosítható. A hiányosság másik következménye, hogy így a szoftver nem tudja előállítani a megfelelő ekvivalens automatát $A \rightarrow xB$ alakú szabályokat tartalmazó nyelvtan esetén. Oktatási szempontból ez jelentős probléma, hiszen a nyelvtanok és automatakapcsolatának vizsgálata az elmélet egy fontos eleme. Jómagam által elkészített szoftver felismeri a szabályredukciókat, lehetőség van erős normál formára hozott ekvivalens nyelvtan generálására, és az egyenértékű automata létrehozása során is figyelembe veszi a láncszabályokat és a szabályredukciókat a szoftver.



48. ábra: Ekvivalens automata

A JFLAP alkalmazásban is lehetőség van tetszőleges bemeneti jelsorozat felismerhetőségének vizsgálatára. A szoftver helyes eredményt ad, de ha az automata felépítéséből kifolyólag több lehetséges levezetési mód is fent áll, a JFLAP ezeket nem veszi észre, mindig csak egy levezetéssel szolgál. Én a szoftveremet úgy terveztem meg, hogy az minden esetben megkeresse az összes levezetési módot. A felhasználó szabadon választhat a levezetések között és bármelyik esetén megvizsgálhatja az elfogadás egyes lépéseit.



49. ábra: Jelsorozat elfogadás több lehetőség esetén

Mind az automaták, mind a nyelvtanok esetén a levezetés egyes lépéseinek prezentálása kevésbé szemléletes a JFLAP-ben, mint az én szoftveremben. Már korábban említést tettem arról, hogy a JFLAP egy igen hosszú múltra visszatekintő alkalmazás. Sajnos ez idő alatt a grafikus megjelenítés korszerűsítésére csekély hangsúlyt fektettek, pedig a levezetés lépéseinek megértéséhez ez nagymértékben hozzájárulna. Én próbáltam szemléletes és dinamikus grafikus reprezentációt alkalmazni, ami jól igazodik a modern elvárásokhoz, és reményeim szerint segítségül szolgálhat az elmélet elsajátítása során.

6. Összefoglalás

A dolgozatban bemutattam a téma megértéséhez szükséges elméleti alapokat, demonstráltam a JFLAP alkalmazás témába vágó funkcióit és azok működését. Megvalósítottam egy olyan alkalmazást, aminek segítségével tetszőleges nyelvtanokat és véges elfogadó automatákat hozhatunk létre. A megvalósított osztályok segítségével olyan komplex formális nyelvi reprezentációt valósítottam meg, amely hűen tükrözi az eredeti értelemben vett nyelvtan és automata modelleket, azok minden szükséges tulajdonságával és funkciójával együtt. Mind a nyelvtanok, mind az automaták esetében megalkottam azok grafikus és dinamikus megjelenítését. Ezzel elértem, hogy azok megjelenése a szakirodalmakban és számos releváns alkalmazásban fellelhető ábráknál szemléletesebb legyen. Sikerült kifejlesztenem egy olyan algoritmust, mely segítségével bármilyen 3-típusú nyelvtanra megvizsgálható, hogy adott szó része-e a nyelvtan által generált nyelvnek és amennyiben igen, a program helyes levezetési megoldással szolgál. Így nem csak a szóprobléma megoldását valósítottam meg, de elvégeztem annak szintaktikai elemzését is, ami a mai napig egy aktuális kutatási területnek tekinthető. Az algoritmust úgy implementáltam, hogy a nyelvtan szabályaira vonatkozóan nem kötöttem ki, hogy azok erős normálformában legyenek. Ilyen jellegű megoldást a fellelt szakirodalmakban nem találtam. Adott szó levezetésének lehetőségét megvalósítottam az automaták esetén is. Az algoritmust úgy terveztem meg, hogy az összes lehetséges levezetési módot megadja. A program képes bármilyen 3-típusú nyelvtanra megkonstruálni a vele ekvivalens véges elfogadó automatát, gondosan ügyelve az esetleges láncszabályok és szabályredukciók kiküszöbölésére, valamint tetszőleges automata esetén is létre tudja hozni a vele egyenértékű 3-típusú nyelvtant.

Munkámban kimondottan értékesnek tartom a szóprobléma szintaktikai elemzésének egyedi kivitelezését, a szemléletes és beszédes grafikai kimenet megvalósítását.

Az adott részterületen sikerült elérnem, hogy az általam létrehozott szoftver szemléletesebb és az elméletben megszokott jelölésmódokhoz, megoldásokhoz hűbb legyen, mint az Interneten fellelt JFLAP szoftver, mely hosszú múltra visszatekintő, igen nagy fejlesztői gárda által létrehozott komplex szimulációs alkalmazás.

Összességében sikerült egy olyan összetett formális nyelvi és automata szimulációs alkalmazást létrehoznom, mely nem csupán segítségül szolgálhat a téma iránt érdeklődő hallgatóknak, de összetettebb szimulációs feladatok elvégzésére is hasznosítható.

7. Irodalomjegyzék

- [1] ALAN P. PARKES (2008) *A Concise Introduction to Languages and Machines*. London: Springer.
- [2] ANDREW TROELSEN (2012) *Pro C# 2008 and the .NET 3.5 Platform*. New York: Springer-Apress.
- [3] ANNA O. BILSKA, KENNETH H. LEIDER, MAGDALENA PROCOPIUC, OCTAVIAN PROCOPIUC, SUSAN H. RODGER, JASON R. SALEMME, EDWIN TSANG (1997) *A Collection of Tools for Making Automata Theory and Formal Languages Come Alive*. Twenty-eighth SIGCSE Technical Symposium on Computer Science Education, 15-19.
- [4] BACH IVÁN (2002) *Formális nyelvek*. Budapest: Typotex Kiadó.
- [5] DÖMÖSI PÁL, FALUCSKAI JÁNOS, HORVÁTH GÉZA, MECSEI ZOLTÁN, NAGY BENEDEK (2011) *Formális nyelvek és Automaták*. (e-könyv)
- [6] ERIC GRAMOND, SUSAN H. RODGER (1999) *Using JFLAP to Interact with Theorems in Automata Theory*. Thirtieth SIGCSE Technical Symposium on Computer Science Education, 336-340.
- [7] FÜLÖP ZOLTÁN (2001) *Formális nyelvek és szintaktikus elemzésük*. Szeged: Polygon Jegyzettár.
- [8] FÜLÖP ZOLTÁN (2013) *Automaták és formális nyelvek*. (e-könyv)
- [9] HUNYADVÁRI LÁSZLÓ, MANHERTZ TAMÁS (2006) *Automaták és formális nyelvek*. (e-könyv)
- [10] ILLÉS ZOLTÁN (2008) *Programozás C# nyelven*. Budapest: Jedlik Oktatási Stúdió.
- [11] JOHN E. HOPCROFT, RAJEEV MOTWANI, JEFFREY D. ULLMAN (2001) *Introduction to Automata Theory, Languages, and Computation*. New York: Addison-Wesley.
- [12] MIKEL ALECHA, MONTSERRAT HERMO (2009) *A Learning Algorithm for Deterministic Finite Automata using JFLAP*. Electronic Notes in Theoretical Computer Science, 248, 47-56.
- [13] SUSAN H. RODGER, ERIC WIEBE, KYUNG MIN LEE, CHRIS MORGAN, KAREEM OMAR, JONATHAN SU (2009) *Increasing Engagement in Automata Theory with JFLAP*. Fourtieth SIGCSE Technical Symposium on Computer Science Education, 403-407.