Formális nyelvek és automaták

Király Roland

2012. november 16.

Tartalomjegyzék

1. fejezet

Előszó

Tisztelt Olvasó. Ez a formális nyelvek és automaták jegyzet rendhagyó abból a szempontból, hogy nem kizárólag a matematikai formalizmusok szemszögéből közelíti meg az említett témaköröket, hanem a gyakorlat, és a gyakorlati alkalmazások felől is.

Ez nem jelenti azt, hogy a matematikai formalizmusokat mellőznénk, mivel ez a fajta hozzáállás nem vezetne semmilyen célhoz. A formalizmusok, és a segítségükkel leírt definíciók nélkülözhetetlen részei mind a matematikának, mind az informatikának, de minden bizonytalanság nélkül kijelenthetjük, hogy minden egyes tudományágnak.

Mindezért a jegyzet lapjain a matematikai formalizmusokkal felírt definíciókhoz a lehető legtöbb esetben találunk gyakorlati, általában az informatikai alkalmazások területéről vett példákat, és megvalósításokat.

A jegyzet főként programtervező informatikus hallgatók számára készült, de nyugodtan forgathatják nyelvtanár, vagy nyelvész szakterületek művelői is.

Megpróbáltuk a fejezeteket logikusan felosztani, és úgy elrendezni a definíciókat és magyarázatokat, hogy azokon sorban haladva még azok is minden nehézség nélkül megértsék az fejezetek tartalmát, akik korábban nem foglalkoztak algoritmusokkal, nyelvészettel, vagy a fordító és elemző programok elméleti kérdéseivel.

A jegyzet egyes szakaszaihoz feladatok is tartoznak. Ezek számos esetben a fejezetek végén találhatóak, és egy részükhöz a megoldásokat is megtalálható a feladathoz. Néhány fejezetben ettől a sémától eltérünk, mivel a definíciók megértéséhez szükséges példák a definíciók közvetlen környezetében lettek elhelyezve, és a megoldásaik ekkor szintén a feladat mellett kaptak helyet.

A jegyzet megírásában sokan voltak a segítségemre. Közülük kiemelném Dr. Hernyék Zoltánt, akinek a főiskolai tanárszakos hallgatók számára írt

jegyzeteit vettem alapul, és Dr. Csörnyei Zoltán Professzort, akinek Fordítóprogramok c. könyvében leírtak szintén alapul szolgáltak az elemzési módszerekről szóló részek algoritmusainak elkészítéséhez.

Kettőjük mellet külön köszönet illeti még Dr. Egri-Nagy Attilát is, aki a diszkrét matematika tudományterületről hozott példái és ismeretei szintén segítettek a jegyzet elkészítésében.

2. fejezet

Bevezetés

2.1. Út a matematikai formulától az implementációig

A jegyzetben, és általában a matematikához közel álló területeken elkerülhetetlen, hogy alkalmazzuk a diszkrét matematikában, és a matematikai többi ágában használatos jelöléseket és formákat. Különösen igaz ez a formális nyelvek, valamint az automaták világában.

A nyelvek megadásánál, és az elemzésükhöz használt programok definiálásánál gyakran alkalmazzuk a halmazelméletnél látott jelölésrendszert, és a definíciókat is halmazokkal, és a rajtuk értelmezett műveletek definiálásával adjuk meg.

Mielőtt azonban ebbe belekezdenénk, foglalkozzunk kicsit a halmazokkal, valamint néhány egyszerű példán keresztül jussunk el az implementálás, de legalább a tervezés fázisáig.

A halmazokat mi az angol ABC nagy betűivel fogjuk jelölni:

$$A, B, C, \ldots, Z.$$

A halmazok elemeit ugyanezen ABC kis betűivel:

$$a, b, \ldots, z,$$

de számos helyen, informális és formális definíciókban a halmazok konkrét értékeit is megadjuk:

$$A := \{a, b, c\},$$

$$B := \{1, 2, 3, \ldots, n\}.$$

A halmazokat, kiemelten a nagy elemszámú, vagy a nem véges halmazokat praktikus okokból nem az elemeik felsorolásával, hanem a halmazba tartozás feltételének a definiálásával, másképpen a halmaz elemeinek létrehozását definiáló algoritmussal adjuk meg.

$$B = \{1,\ 2,\ 56,\ 34,\ 123,\ \ldots\},$$

$$A = \{a|\ a \in B \wedge a\ >\ 2\}$$

Ez a fajta jelölésrendszer egyrészt sokkal kifejezőbb és rövidebb, mint a felsorolások, másrészt hasznos is, mert segítségével sokkal közelebb jutunk a konkrét implementációhoz.

Ha megvizsgáljuk a fenti jelölést, láthatjuk, hogy valójában azt az algoritmust, vagy másképp programot is megadtuk, amely a halmaz elemeit előállítja.

Ez a módszer ismert a funkcionális nyelvi paradigmát képviselő programozási nyelveknél, és úgy hívják, *halmazkifejezés*, vagy *listagenerátor*, és az implementáció alakjában nem sokban különbözik a matematikai formulától:

```
halmaz() ->
   A = [1, 2, 56, 34, 123],
   [a || a <- A, a > 2].

...

B = halmaz(),
```

Az imperatív nyelvi implementáció már sokkal bonyolultabb. Ennek az egyik oka, hogy az imperatív nyelvek kifejező ereje sokkal kisebb az ilyen feladatok esetében, mint a funkcionális nyelveké, a másik ok pedig az, hogy nagyon kevés nyelv esetében találjuk meg a halmazok könyvtári modulokban rendelkezésünkre bocsájtott változatait.

Mindezek miatt az algoritmust nekünk kell elkészítenünk, de ha nem ez a helyzet, akkor sem árt megtanulnunk, hogyan is tehetjük ezt meg.

Az rögtön látszik, hogy valamilyen ismétléses vezérlő szerkezetet kell választanunk, mivel egymást követően több adatot tudunk előállítani. Ahhoz viszont, hogy a több adatot tárolni tudjuk, valamilyen homogén, de indexelhető összetett adatszerkezetet kell keresnünk. Ilyen adatszerkezet a tömb, vagy a lista.

Próbáljuk meg megtervezni azt a programot, amely az

$$A = \{1,\ 2,\ 56,\ 34,\ 123\}$$

halmaz elemeiből előállítja a

$$B = \{a \mid a \in A, a > 2\}$$

halmazt. A szemléletesség miatt az A elemeit egy konstans tömbben, vagy listában (az implementációra használt nyelv lehetőségeihez mérten) adjuk meg, majd ezt a listát bejárva egy ciklussal minden, az $a > 2$ feltételnek megfelelő elemet átemelünk egy kezdetben üres, majd az A megfelelő elemeit tartalmazó, egyre bővülő listába.

```
EGÉSZ[] A = [1, 2, 56, 34, 123];
EGÉSZ Hossz = HOSSZ(A);

EGÉSZ i, j = 0;

AMÍG i < Hossz ISMÉTEL
  HA A[i] > 2
     B[j] = A[i];
     j = j + 1;
  HA VÉGE
  j = j + 1;
ISMÉTLÉS VÉGE
```

A leírónyelvi változatot most már könnyedén átírhatjuk egy konkrét programozási nyelvi változatra. Természetesen kisebb változtatásokat eszközölnünk kell a programot, és ki kell használnunk az implementációhoz használt programozási nyelv kínálta lehetőségeket.

```
...
int i = 0;
int[] A = new int[10] {1, 2, 56, 34, 123};

while (i < A.length)
{
  if (A[i] > 2)
  {
```

```
      B.add(A[i]);
    }
  }
  ...
```

A programkódban a halmazok helyett *int* tömböket használtunk, és a *HOSSZ* függvény helyett az int tömb *length* tulajdonságát (Objektum Orientált nyelveknél ez a módszer használatos az elemszám megadására) az ismétlés megvalósításához.

Ahogy láthatjuk a matematikai formula nemhogy bonyolítaná a programozás folyamatát, hanem inkább segíti azzal, hogy általa ellenőrizni tudjuk a program helyességét már a kezdeti, tervezési fázisban.

Mindezek mellett össze tudjuk hasonlítani az absztrakt matematikai modellt a konkrét implementációval, és ki tudjuk szűrni annak hibáit. Ezt a műveletet verifikációnak nevezi a szakirodalom, és ez a programtervezési lépés szerves részét képezi a szoftverek teljes életciklusát magába foglaló tervezési folyamatoknak.

2.2. Feladatok

- Definiáljuk formálisan azt az A halmazt, amelynek elemei az egész számok halmazából származnak, de az A halmaz csak azokat az elemeket tartalmazza, amelyek nem oszthatóak hárommal, és kisebbek száznál.
- Készítsük el az előző feladat leíró nyelvi implementációját, majd ebből egy konkrét nyelvi implementációt.
- Adjuk meg annak a halmaznak a matematikai definícióját, amely halmaz a magyar ábécé páros betűit tartalmazza.
- Készítsünk programot, amely egy tetszőlegesen választott típusú elemeket tartalmazó halmazról eldönti, hogy az üres-e.
- Készítsünk programot, amely egy tetszőlegesen választott típusú elemeket tartalmazó halmazról eldönti, hogy az tartalmazza-e a paraméterként megadott elemet.
- Készítsünk programot, amely két tetszőlegesen választott típusú elemeket tartalmazó halmaznak megadja a közös elemeit.
- Készítsünk programot, amely képzi két tetszőlegesen választott típusú elemeket tartalmazó halmaz unióját.

- Készítsünk programot, amely képzi két tetszőlegesen választott típusú elemeket tartalmazó halmaz metszetét.

- Készítsünk programot, amely képzi két tetszőlegesen választott típusú elemeket tartalmazó halmaz különbségét.

2.3. Típus, művelet, állapot és állapottér

Ahhoz, hogy a jegyzet fejezeteiben található definíciókat, és magyarázatokat jobban megérthessük, a matematikai formalizmusok vizsgálata mellett szükségünk lesz még néhány fontos fogalom tisztázására.

Az első ilyen a fogalom a típus. A matematikában, és az informatikában mikor adatokról beszélünk, a legtöbb esetben megadjuk az adatok tárolására alkalmas típust is, más néven adattípust. Definiáljuk az adattípust, amely informális megadás esetén a következő párral jellemezhető:

$$(\mathcal{A}, \mathcal{M}),$$

ahol a pár első eleme az adatok halmaza, és a második elem, az $\mathcal{M}$ a műveletek véges halmaza. Most nézzük meg azt a néhány tulajdonságot, amely a számunkra fontos lehet:

$$\forall\, m \in M : m \to A,$$

A műveletek az adatokon vannak értelmezve, és kell legalább egy művelet, amely az összes adat előállítására alkalmas.

$$\mathcal{M}_k \subset \mathcal{M}$$

A műveletek ezen részhalmazát konstruktív műveletnek, vagy másképpen *konstruktornak* nevezzük (a konstruktor kifejezést inkább az összetett adattípus esetén alkalmazzuk).

Az adattípussal definiálhatjuk a programjainkban használt változók típusát. Ezt a lépést deklarációnak nevezzük. Az adattípus megadásával ahhoz állapotinvariánst is rendelünk. Az adattípussal megadott változó csak az állapot invariánsnak megfelelő értékeket veheti fel.

Az állapot időintervallumhoz kötött, és valamely $m \in \mathcal{M}$ művelet hatására jön létre. Ugyanígy az egyik állapotból a másikba is minden esetben egy művelet hatására kerül. A műveleteknek lehet paramétere, elő, és utófeltétele, valamint még számos olyan tulajdonsága, amelyek a számunkra nem lényegesek, ezért ezekkel nem is foglalkozunk.

Az állapot, valamint az állapot átmenet azért fontos, mert az automaták tárgyalásánál, valamint implementációjuk bemutatásánál sok esetben hivatkozni fogunk ezekre a fogalmakra.

Az automatákat is a belső állapotaikkal jellemezzük, és a szavak, valamint a mondatok felismerését is az állapotokra alapozzuk majd. Minden automatának, ha programként tekintünk rájuk definiálnunk kell az állapotait, és a teljes állapotterét, amelyet az attribútum értékeik aktuális állapotainak rendezett n-eseivel jellemezünk majd.

$$(A_{akt}, I, [V_{akt}])$$

Ahol a hármasból az első elem az aktuális állapotot jelöli, a második pedig az bemenetre érkező, ellenőrzendő szöveg még hátralévő részét (lásd.: később). (A harmadik elem csak verem automaták esetén van jelen, és a verem aktuális állapotát tartalmazza.)

Ez azt jelenti, hogy egy automataosztály megadásánál leírjuk az automata összes lehetséges állapotait, valamint a kezdő, és a befejező állapotát. Verem automata esetén a felismeréshez használt verem, és az input szalag állapotát is. Ezeket az állapotokat változókban, vagy változók rendezett n-eseiben tároljuk majd.

Az elemzés műveletét szintén állapotok rendezett n-eseivel (konfiguráció), és az azokat megváltoztató állapotátmenetek sorozataival adjuk meg, minden automata osztálynál.

2.4. Feladatok

- Adjuk meg az ismert halmaz összetett adattípus formális definícióját (a műveletekhez tartozó axiómák megadása nem szükséges).
- Adjuk meg az ismert verem adattípushoz tartozó állapot invariánst általános formában. (A maximálisan a veremben elhelyezhető elemek száma általánosan n).
- Készítsük el a verem adattípus modelljét az általunk választott eszközzel. Ez lehet egy konkrét programozási nyelv, vagy egy absztrakciót támogató tervező eszköz, mint az UML.

  A feladat megoldásához adjuk meg az alaphalmazt, a halmazon értelmezhető műveleteket, valamint az invariáns tulajdonságot leíró feltételeket is.

3. fejezet

ABC-k, szavak, nyelvek

3.1. Szavak és ábécék műveletei

Mielőtt elmélyednénk a formális definíciókban, vizsgáljunk meg néhány, az ábécékre, és a szavakra kimondott állítást. A jobb megértés érdekében ezeket később formálisan is definiáljuk majd.

- Egy (általában véges), nem üres halmazt **ABC**-nek tekinthetünk.
- Egy ABC elemeit (halmazt alkotó elemeket) **jeleknek** (karaktereknek, betűknek, szimbólumoknak) nevezzük.
- Egy ABC jeleiből alkotott kifejezéseket az ABC fölötti **szónak** nevezzük. Jele kis görög betű. Pl: az α « egy A ABC fölötti szó.
- Egy ABC fölötti α szó hosszán az őt alkotó jelek számát értjük.
- Egy ABC fölötti ε szót üres szónak nevezzük. Az üres szó jele általában ε, ϵ görög betű (epszilon).

A következőkben a fenti állításokat fogjuk végigjárni, és ahol lehet formálisan definiáljuk a felmerülő fogalmakat.

3.2. Véges szavak

Amennyiben A egy véges nem üres halmaz, tekinthetjük ábécének. Ahogy korábban már említettük, az ábécék elemei betűk, vagy másképpen szimbólumok. Az A halmaz $a_0, a_1, \ldots, a_n$ elemeiből képzett sorozatokat az A ábécé fölötti szavaknak nevezzük. És ahogy azt korábban már szintén láthattuk, az így képzett szavak hossza az adott szót alkotó jelek számával egyezik meg.

Ezt a $|a_1 \dots a_n|$, vagy az $L(a_1 \dots a_n)$ formában adhatjuk meg, de sokkal egyszerűbb, ha a szavakat az $\alpha, \beta, \dots$ betűkkel jelöljük. Ekkor a szó hossza az $L(\alpha)$, vagy a $|\alpha|$ formában adott.

Az összes szó az adott ábécé szimbólumaiból áll elő az ábécé hatványozása során:

$$A^+ = A^* \ \{\varepsilon\},$$

és

$$A^n = \{\alpha \in A^* |\ |\alpha| = n|\},$$

vagyis

$$\{a_1, a_2, \dots, a_n | a_i \in A\}.$$

Mindez azt jelenti, hogy az A^+ az A fölötti szavak halmaza kivéve az üres szót, és az A^* az A ábécé fölötti összes szó az üres szót is beleértve. Az A^n az n hosszúságú szavak halmazát jelenti, és $A^0 = \{\varepsilon\}$, ahol az $|\varepsilon|$, vagyis az $L(\varepsilon) = 0$.

3.3. Műveletek véges szavakkal

3.4. Konkatenáció

A szavakkal műveleteket végezhetünk, és a műveleteknek ugyanúgy vannak tulajdonságai, mint például a számokon értelmezett műveleteknek.

Az első művelet, amelyet értelmezünk szavakra a konkatenáció (szavak szorzása), ami egyszerűen azt jelenti, hogy kettő, vagy több szóból (tekinthetünk ezekre résszavakkén) egy új szót képezünk, vagyis egymás mellé illesztjük ezeket.

Egy A ABC fölötti α és β szavak konkatenációján azt a γ A ABC fölötti szót értjük, melyet úgy kapunk, hogy az α szót alkotó jelek mögé írjuk a β szót alkotó jeleket. A konkatenáció jele a $+$.

1. Megjegyzés. *Vagyis ha pl:* $\alpha =$ *"alma" és* $\beta =$*"fa" akkor* $\alpha + \beta =$ *"almafa".*

A konkatenáció jelölése során a $+$ jelet nem mindig írjuk ki, $\alpha + \beta = \alpha\beta$.

Amennyiben informálisan szeretnénk definiálni a műveletet, akkor a következő definíció megfelelő lesz:

1. Definíció (Összefűzés). *Legyenek α, és β az A ábécé feletti szavak, vagyis az ábécé szimbólumaiból képzett szavak. Ekkor az $\alpha\beta$ eredménye a két szó konkatenáltja, vagyis $\gamma = \alpha\beta$, ahol a $|\gamma| = |\alpha| + |\beta|$, vagyis az új szó hossza megegyezik a két résszó hosszával.*

Nézzük meg most a teljesen formális definíciót is, amely a következőképpen írható le:

2. Definíció (Konkatenált). *Ha az $\alpha = a_1, a_2, \ldots, a_n$, és a $\beta = b_1, b_2, \ldots, b_m$ az A ábécé feletti szavak, akkor:*

$$\gamma = \alpha\beta = a_1 a_2 \ldots a_n b_1 b_2 \ldots b_m.$$

A fenti definíciónak van néhány következménye, amelyek szintén fontosak a számunkra:

3.5. Konkatenáció tulajdonságai

Asszociatív, nem kommutatív, és van neutrális elem.

Ha megvizsgáljuk ezeket a tulajdonságokat, ezek alapján újabb megállapításokat tehetünk:

Amennyiben adott egy α « A (A ABC fölötti α szó):

- $\alpha^0 = \epsilon$ (Bármely szó nulladik hatványa az üres szó).
- $\alpha^n = \alpha + \alpha^{n-1}$ $(n \geq 1)$ (Bármely szó n. hatványa nem más, mint a szó n-szeres konkatenációja)
- az α szó a γ szó prefixuma (szókezdő résszó), és, mivel az α hossza nem nulla ($|\alpha| \neq 0$), ezért valódi prefixum.
- a β szó a γ szó szuffixuma (szózáró résszó), és, mivel a β hossza nem nulla ($|\beta| \neq 0$), ezért valódi szuffixum.
- a művelet asszociatív, vagyis az $\alpha(\beta\gamma)$ ekvivalens az $(\alpha\beta)\gamma$ művelettel.
- a művelet nem kommutatív, vagyis az $\alpha\beta \neq \beta\alpha$.
- a műveletnek van semleges eleme (neutrális elem), vagyis $\varepsilon\alpha = \alpha\varepsilon$, és az A^* ábécé, vagy pontosabban halmaz a művelettel *monoid*.

3.6. Szavak hatványozása

A következő műveletünk a szavak hatványozása, amely műveletet úgy képzelhetünk el, mint az n szeres konkatenációját egy adott szónak. Tehát felhasználva a konkatenáció műveletét, a hatványozást már könnyedén értelmezhetjük és definiálhatjuk formálisan.

3. Definíció (Szavak hatványa).

$$\alpha^0 = \varepsilon$$

$$\alpha^n = \alpha^{n-1}\alpha$$

akkor, ha $n \geq 1$., vagyis egy α szó n-edik hatványa megegyezik a szó n-szeres konkatenáltjával.

Ebből a műveletből is következik néhány állítás, amelyeket a következő felsorolásban láthatunk:

- az α szó primitív, ha egyetlen másik szónak sem a hatványa, vagyis α primitív szó, ha $\alpha = \beta^n, \beta \neq \varepsilon \Rightarrow n = 1$. Például $\alpha = abcdefgh$ primitív szó, mivel nem bontható fel, de az 123123123 szó nem az, mivel $\alpha = (123)^3$.
- Az α, és a β szavak egymásnak *konjugáltjai*, ha létezik $\alpha = \gamma\delta$, és $\beta = \delta\gamma$.
- $\alpha = a_1, a_2, \ldots, a_n$ szó *periodikus*, ha létezik olyan $k > 1$ szám, hogy $a_i = a_{i+k}$, $i = 1, 2, \ldots, n-k$ értékre, ekkor a k, az α szó egy periódusa. Az $\alpha = 1231231$ szó legkisebb periódusa 3 (123).

3.7. Szavak tükrözése

4. Definíció (Szavak tükrözése). *Az $\alpha = a_1, a_2, \ldots, a_m$ szó esetén az $\alpha^T = a_m, a_{m-1}, \ldots, a_1$ szó az α tükörképe. Amennyiben $\alpha^T = \alpha$, akkor ez a szó palindrom szó.*

A fentiekből következik az is, hogy az $(\alpha^T)^T = \alpha$, vagyis, ha kétszer egymás után tükrözzük az α szót, akkor az eredeti szót kapjuk vissza.

Példaként az *abccba* egy palindrom szó, és az "*indulagörögaludni*", vagy a "*gézakékazég*" szintén palindrom szövegek, amennyiben a szóközöket eltávolítjuk, valamint a kis és nagybetűket ekvivalensnek tekintjük.

3.8. Résszavak

5. Definíció (Résszó). *A β szó résszava az α szónak, ha léteznek γ, és δ szavak úgy, hogy $\alpha = \gamma\beta\delta$, és $\gamma\delta \neq \varepsilon$, vagyis β valódi résszava az α szónak.*

6. Definíció (Különböző hosszúságú résszavak). *Az α szó k hosszúságú résszavainak halmazát jelöljük $R_k(\alpha)$-val. Az $R(\alpha)$ az összes ilyen résszó halmaza, vagyis*

$$R(\alpha) = \bigcup_{k=1}^{|\alpha|} R_k(\alpha).$$

Például, ha vesszük az $\alpha = abcd$ szót, akkor a szó egy hosszúságú résszavai, vagyis

$$R_1(\alpha) = \{a, b, c, d\},$$

a kettő hosszúságú résszavai

$$R_2(\alpha) = \{ab, bc, cd\},$$

a három hosszúságúak

$$R_3(\alpha) = \{abc, bcd\},$$

és az egyetlen négy hosszúságú maga a szó, vagyis

$$R_4(\alpha) = \{abcd\}.$$

3.9. Szavak bonyolultsága

Ahogy mindennek a matematikában, és az informatikában van valamilyen szintű bonyolultsága, így a szavaknak is. Minden formáját a bonyolultságnak valamilyen mértékrendszer bevezetése mellett vizsgálunk. A szavak bonyolultságát a résszavaik vizsgálatán alapuló mértékre alapozzuk. Mindezek alapján, vagyis a szavak alakjának, valamint résszavainak ismeretében definiálhatjuk a szó bonyolultságát.

Pontosabban, a szavakra értelmezett bonyolultság alatt az adott szót alkotó résszavak sokféleségét, vagy változatosságát értjük. Ez azt jelenti, hogy a bonyolultság megállapításához meg kell keresnünk a szó különböző hosszúságú részszavait, és ezek összes előfordulásait.

7. Definíció (Szavak bonyolultsága). *A szó bonyolultsága a különböző hosszúságú résszavainak a számával egyezik meg. Az α szó k hosszúságú résszavainak száma $r_\alpha(k)$.*

A bonyolultság ismeretében értelmezhetjük a maximális bonyolultságot is, amelyet a következőképpen definiálhatunk:

8. Definíció (Maximális bonyolultság). *A maximális bonyolultságot csak véges szavakra értelmezhetjük, így a*

$$Max(\alpha) = max\{r_\alpha(k)|k \neq 1\}, \alpha \in \mathcal{A}*,$$

ahol az $\mathcal{A}*$ az adott ábécéből képzett lezárt. (Végtelen szavakra értelmezhetjük az alsó, vagy a felső maximális bonyolultságot.)

Ahogy egy szónak létezik maximális bonyolultsága, úgy létezik teljes bonyolultság is, amely az alábbi definícióban található:

9. Definíció (Teljes bonyolultság). *A teljes bonyolultság egy adott szó összes, nem üres különböző résszavainak a számát jelenti, vagyis*

$$Tb(\alpha) = \sum_{i=1}^{|\alpha|} r_\alpha(i), \alpha \in \mathcal{A}*.$$

3.10. Mondatok bonyolultsága

Ebben a fejezetben nem kimondottan a beszélt nyelvi mondatok bonyolultságával, hanem inkább gyakorlati megfontolásból a programok, pontosabban a különböző programozási nyelveken írt mondatok bonyolultságával foglalkozunk.

Hogy ennél is pontosabbak legyünk, a programozási nyelvek különböző, az adott paradigmára és nyelvre jellemző nyelvi konstrukcióival.

Minden programozási nyelvben számos olyan nyelvi elem van, amelyeket egymásba ágyazhatunk, vagy sorban egymást követően használhatjuk az egymásba ágyazott elemeket. Készíthetünk bonyolultabb konstrukciókat is, mint a függvények, vagy eljárások, amelyek szintén különböző nyelvi elemekből épülhetnek fel.

Annak, hogy a nyelvi elemeket az adott programozási cél elérése érdekében hogyan, és milyen kombinációkban használjuk, nincs meghatározott szabály, vagyis nincs olyan jól bevált séma, vagy módszer, amely minden probléma megoldása során alkalmazható lenne.

Így a programok bonyolultsága még egyazon feladat különböző megoldásainál is nagyon eltérő bonyolultságot eredményezhet. Ez hamar a tesztelhetőség, valamint a javíthatóság lehetőségét veszi el a program fejlesztőjétől, mivel a programszöveg átláthatatlanná, és kezelhetetlen bonyolultságúvá válik.

Mindezek miatt, és azért, mert minden egyes fejezetnél, és új fogalom bevezetése esetén feltett célunk, hogy annak a gyakorlati hasznát, vagy egy lehetséges gyakorlati alkalmazását is szemügyre vegyük, vizsgáljunk meg néhány, a programszövegek bonyolultságát érintő fogalmat.

A bonyolultság vizsgálatakor a forrásszöveg azon tulajdonságait mérjük, és számszerűsítjük, amely tulajdonságok segítségével képet kaphatunk a forrásszöveg felépítéséről, karakterisztikájáról, és a programelemek együttes alkalmazásának bonyolultságáról. A bonyolultság alapján becsléseket adhatunk a programszöveg tesztelési, fejlesztési, valamint átalakítási költségeire.

A szoftverek bonyolultságát mérhetjük a forráskód összetettsége (struktúrája) valamint mérete alapján. Vizsgálhatjuk a forrásszöveget a fejlesztési fázisban (*process metrics*), vagy a kész programokat a felhasználhatóságuk alapján. Ez a fajta vizsgálat a végterméket jellemzi (*product metrics*), de szorosan kapcsolódik a forrásszöveghez, és a modellhez, ami alapján a forrásszöveget felépítették.

A strukturális bonyolultságot mérhetjük továbbá a fejlesztési költségek (*cost metrics*), a szükséges erőfeszítés (*effort metrics*) alapján is, de a fejlesztés előre haladottsága (*advancement*), vagy a megbízhatóság alapján (*nonrelialibity* (hibaszám)). Lehet mérni a forrásszöveget az újrahasznosítás mértékének (*reusable*) számszerűsítésével, és mérhetjük a funkcionalitást *functionality*, vagy a használhatóságot is, de a bonyolultsági mértékek mindegyike az alábbi három fogalom köré csoportosul:

- méret,
- komplexitás,
- stílus.

A szoftver bonyolultsági mértékek minősíthetik a programozói stílust, a programozás folyamatát, a felhasználhatóságot, a várható költségeket, és a programok belső jellemzőit. Természetesen a programozás során mindig arra törekszünk, hogy a mérhető felhasználhatósági mérték, az erőforrások használata, és program belső jellemzői összhangban legyenek.

Mindezek alapján megállapíthatjuk azt, hogy a programok jellemzéséhez nem elegendő egyetlen tulajdonság, vagy attribútum vizsgálata, sőt, ha tovább megyünk, nem elég az összes lehetséges mérték összegyűjtése és mérése.

A programszöveg elemei közötti kapcsolatokhoz hasonlóan a mértékek közti kapcsolatok felderítése, és azok egymásra hatásának a vizsgálata adhat csak pontos képet a mérés tárgyát képző szoftverről.

3.11. A bonyolultságból adódó problémák

Thomas J. McCabe [?] már 1976-ban rámutatott arra, hogy milyen fontos a forráskód struktúrájának az elemzése. McCabe cikkében leírja, hogy az IBM által javasolt, a programkód kezelhetősége szempontjából ideális 50 soros modulok esetén 25 egymást követő `IF THEN ELSE` konstrukció is 33.5 millió különböző végrehajtási ágat eredményez. Ilyen nagyszámú végrehajtási ágat bármely ma ismert rendszer használata mellett sem lehet emberi időn belül tesztelni, ezáltal a program helyességét tesztelési módszerek használatával igazolni.

A probléma vizsgálata rámutat arra, hogy a programok bonyolultsága a forráskódban található vezérlő szerkezetek száma, azok egymásba ágyazottságának mértéke, és a forráskód minden egyéb mérhető attribútuma fontos szerepet játszik abban, hogy milyen költségekkel, és milyen mértékű költség ráfordítással lehet azokat tesztelni, javítani, átalakítani.

Minthogy a jegyzet terjedelme, és témája miatt nem ad lehetőséget az összes lehetséges bonyolultságot érintő probléma, és azok mérésének a bemutatására, a különböző mértékek közül kizárólag egyet fogunk definiálni, és az a korábban bemutatott példa miatt legyen a McCabe féle ciklomatikus szám:

McCabe féle ciklomatikus szám. A *mc_ cabe* bonyolultság mérték értéke a *Thomas McCabe* által konstruált vezérlési gráfban [?] definiált alapvető útvonalak számával azonos, vagyis azzal, hogy hányféle kimenete lehet egy függvénynek nem számítva a benne alkalmazott további függvények bejárási útvonalainak a számát. A *Mc Cabe* ciklomatikus számot eredetileg a procedurális nyelvek alprogramjainak a mérésére fejlesztette ki *Thomas J. Mc Cabe* [?]. *Mc Cabe* a programok ciklomatikus számát a következőképpen definiálja:

10. Definíció. [*Mc Cabe-féle ciklomatikus szám*] *A $G = (v, e)$ vezérlési gráf $V(G)$ ciklomatikus száma $V(G) = e - v + 2p$, ahol p a gráf komponenseinek a számát jelöli, ami megegyezik az erősen összefüggő gráfban található lineárisan összefüggő körök számával.*

Nézzünk meg egy konkrét példát a ciklomatikus szám alkalmazására. Vegyük azt az esetet, mikor a programunk 4 feltételes elágazásból, és egy olyan

feltételes ciklusból áll, amelynek a feltétele összetett, és pontosan két részfeltételből épül fel.

Ekkor a ciklomatikus számot a programban található feltételes döntések száma alapján kapjuk úgy, hogy a döntések számához hozzá kell adnunk egyet, valamint az összetett feltételt kétszer kell számolnunk. Ez azért van így mert a programban található összes döntések számát kell vizsgálnunk, így a számításunk eredménye erre a programra hét. Valójában az eredményhez hozzávehetjük még a programban szereplő kivételkezelők, valamint a több ágból (OOP, vagy Funkcionális paradigmában található "overload" típusú függvények esetén) álló függvények által behozott döntések számát is hasonló feltételekkel, mint ahogyan azt az elágazásoknál, vagy a ciklusoknál tettük.

3.12. Végtelen szavak

A véges szavak mellet a végtelen szavakat is értelmezhetjük, amely szavak hasonlóan a végesekhez, egy adott ábécé elemiből képezhetőek. A $\forall a \in A$ szimbólumokból képzett $\alpha_w = a_1 a_2 \dots a_n \dots$ végtelen szavak jobbról végtelenek, vagyis az α_w ekkor jobbról végtelen szó.

11. Definíció (Végtelen szavak). *Mindezek alapján a jobbról végtelen szavak halmazát jelölje A_w, a véges és végtelen A abécé feletti szavak halmazát pedig jelöljük a következőképpen:*

$$A_{osszes} = A^* \cup A_w.$$

Ebben az esetben, vagyis a végtelen szavak esetén is értelmezhetjük a résszó, prefixum, és szuffixum fogalmakat.

3.13. Műveletek ABC-kel

A szavak mellett az ábécékkel is végezhetünk műveleteket. Ezek a műveletek azért fontosak, mert megértésükön keresztül eljuthatunk a formális nyelvek definíciójáig.

12. Definíció. *Ha A és B két ABC, akkor $A * B := \{ab | a \in A, b \in B\}$. Ezt a műveletet komplexus szorzásnak hívják.*

2. Megjegyzés. *Vagyis két ABC komplexus szorzatán egy olyan ABC-t értünk, melynek a karakterei kettős jelek, az egyik jel az első ABC-ből származik, a másik jel pedig a második ABC-ből.*

Pl.: $A := a, b$, és $B := 0, 1$. $C := A * B := a0, a1, b0, b1$. Ezek alapján a C ABC fölötti szó pl. az $\alpha =$"a0b0a1", és $L(\alpha) = 3$, ugyanis a fenti szót három C-beli szimbólum alkotja, az "a0", a "b0", és az "a1".

Ugyanakkor, pl. az "a0aba1" szó nem lehet "C" ABC fölötti szó, ugyanis azt nem lehet csak "C" ABC jeleiből összerakni.

13. Definíció. *Legyen adva egy A ABC.* $A^0 := \varepsilon$ *, vagyis minden ABC nulladik hatványa az a halmaz, amelynek egyetlen eleme van, az* ε *(üres szó).*

14. Definíció. $A^n := *A^{n-1}$ *ahol* $n \geq 1$*. Vagyis egy ABC n-edik hatványán az n-szeres komplexus szorzatát értjük.* $A^0 = \varepsilon$ *szükséges, mivel* $A^1 = *A^0$*, és vissza kell kapni A-t!*

3. Megjegyzés. *Mindezek alapján az A ABC harmadik hatványa egy olyan ABC, amelynek minden eleme igazából három karakterből áll. Általánosan: az n-edik hatványa egy olyan ABC, melynek minden eleme n karakter hosszú.*

3.14. Lezárt, pozitív lezárt

Pl.: ha A=a,b, akkor pl. A^2 =aa,ab,ba,bb. Ezt a halmazt mint ABC is fel lehet fogni, úgy is kezelhetjük, mint az A ABC fölötti kettő hosszú szavak halmazát.

15. Definíció. *V:=* $\alpha|\alpha$ *« A és L(*α*)=1 . Vagyis legyen a V halmaz az A ABC fölötti egy* ***hosszú szavak halmaza****. Jele* V^{*1} *« A, vagy röviden V.*

16. Definíció. *A V halmazon értelmezett kontextusos szorzás* $V \otimes V := \{\alpha\beta|\alpha \in V\}$ *és* $\beta \in V$ *műveletén azt a szavakból álló halmazt értjük, amelyeket a meglévő szavakból kapunk úgy, hogy mindegyiket mindegyikkel összefűzzük.*

A V halmazt igazság szerint az 1 hosszú szavak alkotják. A $V \otimes V$ halmazt pedig a kettő hosszú szavak alkotják.

17. Definíció. $V^0 := \{\varepsilon\}$*, és* $V^n := V \otimes V^{n-1}$ *,* $n \geq 1$*, és A* $V* := \bigcup_{i=0}^{\infty} V^0 = V^0 \cup V^1 \cup V^2 \cup \ldots$ *halmazt a "V"* ***lezártjának*** *nevezzük.*

Vagyis elemei a nulla hosszú szó, az egy hosszú szavak, a kettő hosszú szavak, stb...

18. Definíció. *$V+ := \bigcup_{i=0}^{\infty} V^0 = V^1 \cup V^2 \cup V^3 \cup \ldots$ halmazt a "V" **pozitív lezártjának** nevezzük. vagyis $V^* = V+ \cup \epsilon$,*

A V+ elemei az egy hosszú szavak, a kettő hosszú szavak, stb., így V+-nak nem eleme az üres szó. Nézzünk erre néhány egyszerű példát:

- Ha V:='a','b'. Akkor V*=ε,'a','b','aa','ab','ba','bb','aaa','aab',.... és V+='a','b','aa','ab','ba','bb','aaa','aab',....
- Az $\alpha \in$ V* azt jelenti, hogy α egy tetszőleges hosszú szó, $L(\alpha) \geq 0$.
- Az $\alpha \in$ V+ azt jelenti, hogy α egy tetszőleges hosszú szó, de nem lehet üres szó, vagyis $L(\alpha) \geq 1$.
- Ha V= 0, 1 , ekkor a V* a bináris számok halmaza (és benne van az ϵ is).
- Ha V= 0 , W= 1 , ekkor a (V $\cup$ W) * = $(01)^n \mid$ n $\in$ N .

Ahhoz, hogy teljesen biztosak legyünk a lezárt, és pozitív lezárt fogalmában, utolsó példaként vegyünk egy egyszerű, és már ismert kifejezést, amely a lezárt fogalmát összekapcsolja a reguláris kifejezések fogalmaival.

A példában a $S := 0, 1, \ldots, 9$ halmazt véve alapul írjuk fel azt a reguláris kifejezést, amelyre az összes egész szám illeszkedik:

$$(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)^+$$

Figyeljük meg a kifejezés végén a plusz jelet (+), amelyet azért helyeztünk el oda, hogy segítségével jelezzük a halmaz (kifejezés) pozitív lezárását. A pozitív lezárt ebben az esetben is azt jelenti, hogy a kifejezés nem generálhatja az üres szót, tehát egy számjegyet mindenképpen le kell írnunk, de tetszés szerint írhatunk többet is bármilyen sorrendben.

Amennyiben a kifejezés végére a $*$ jelet helyezzük, a kifejezés ekkor megengedi azt is, hogy ne írjunk le semmit, és ez a gyakorlati alkalmazás során számos problémához vezethet.

A lezárt fogalma egyébként ismerős lehet a matematikából, de akár az adatbázis kezelés azon területeiről, amelyek a relációk attribútumai közötti függőségeket tárják fel a relációk normalizálása érdekében. Ezekben az esetekben is lezártakat alkalmazunk azzal a különbséggel, hogy ott a függőségekben szereplő összes reláció attribútumot keressük, és nem egy halmaz elemeinek összes lehetséges variációját.

Lezártról beszélünk akkor is, mikor egy programban az egy függvényből kiinduló függvényhívási útvonalakt szeretnénk megtalálni, vagyis azt, hogy a kiindulási függvényből milyen hívási útvonalak vezetnek más függvényekbe.

Ezek a területek azért függenek össze, mert mindegyik esetében a halmazelméleti fogalmak szolgálnak alapként a műveletek felírásánál és alkalmazásánál.

3.15. Formális nyelvek

19. Definíció. *Legyen V^{*1} « A. Egy $L \subseteq V^*$ halmazt az "A" ABC fölötti* ***formális nyelvnek*** *nevezzük.*

4. Megjegyzés. *Vagyis a formális nyelv nem más, mint egy adott ABC jeleiből alkotott tetszőleges hosszú szavak halmazának részhalmaza, vagyis a formális nyelv egy adott ABC jeleiből alkotható, meghatározott szavak halmaza.*

5. Megjegyzés. *A formális nyelv állhat véges sok szóból, állhat végtelen sok szóból és tartalmazhatja az üres szót is.*

Ismét vizsgáljunk meg néhány egyszerű példát:

- A:=a,b, V*1«A. Ekkor az L:='a','ab','abb','abbb','abbbb',... nyelv egy "A" ABC fölötti formális nyelv, mely végtelen sok szót tartalmaz ('a'-val kezdődő, tetszőleges sok 'b'-vel folytatódó szavakból alkotott nyelv).
- A:=a,b, V*1 « A . Ekkor az L:='ab','ba','abab','baab','aabb',... nyelv egy "A" ABC fölötti formális nyelv, mely végtelen sok szót tartalmaz (olyan szavak halmaza, ahol minden szóban annyi 'a'-van, mint 'b').

20. Definíció. *Ha L1,L2 két formális nyelv, akkor az L1*L2:= $\alpha\beta$ / $\alpha \in$ L1 és $\beta \in$ L2. Ezt a műveletet nyelvek közötti kontextus szorzásnak hívjuk.*

A kontextus szorzás disztributív: L1* (L2 $\cup$ L3) = L1*L2 $\cup$ L1*L3.
Egy formális nyelv többféle képpen is megadható:

1. Felsorolással.
2. Megadhatjuk azt az egy, vagy több tulajdonságot, amellyekkel a nyelv szavai rendelkeznek, de az összes többi szó nem.
3. A szavakat alkotó szabály szöveges leírásával.

4. A szavakat alkotó szabály matematikai leírásával.

5. Generatív grammatika segítségével.

Az elemek felsorolása nem éppen a leg célravezetőbb eszköz, és kizárólag véges nyelvek esetén lehet működőképes, de még ezeknél sem minden esetben egyszerű.

$$L1 = \{a, b, c, d, \ldots, z\},$$
$$L2 = \{0, 1, 2, 3, 4, \ldots, 9\},$$

vagy

$$L2 = \{a, ab, abc, \ldots\}.$$

A szöveges leírás talán egy kicsit jobb módszer az előzőnél, de sajnos ennek meg az a hibája, hogy nem egyértelmű, vagyis nem ugyanazt jelenti mindenki számára, és ráadásul nagyon nehéz belőle algoritmust, vagy programot készíteni. Hogy mindezeket megértsük nézzük meg az alábbi példát:

"Az L1 legyen egy olyan nyelv, amely tartalmazza az egész számokat, de csak azokat, amelyek nem nagyobbak háromnál...".

A másik példánk, mikor valamely tulajdonsággal adunk meg egy nyelvet, legtöbbször a matematikai formulát használjuk.

Mikor valamely tulajdonsággal adunk meg egy nyelvet, mint a következő példákban.

- legyen L3 := L1*L2 nyelv (azon számok nyelve, melyek páratlanok, de tartalmaznak legalább egy páros számjegyet). Ez a forma tartalmaz szöveges magyarázatot, amely a matematikai formulák esetén teljesen el is hagyható.

- L := $0^n 10^n | n \geq 1$, vagyis a szám közepén áll egy 1-es, és előtte, mögötte ugyanannyi 0.

- $L_4 = \{a^n b^n | n = 1, 2, 3, \ldots\}$

3.16. Feladatok

1. Adjuk meg az "alma" szó negyedik hatványát.

2. Hatérozzuk meg a következő szó résszavainak a számát: "abcabcdef"

3. Állapítsuk meg, hogy az "ababcdeabc" és az "1232312345" szavak közül melyiknek nagyobb a bonyolultsága.

4. Adjuk meg a következő két ábécé Descartes szorzatát: $A = \{0, 1\}$, és $B = \{a, b\}$.

5. Adjuk meg az előző feladatban definiált két halmaz komplexus szorzatát.

6. Definiáljuk azt a halmazt, amely a természetes számok halmazából tartalmazza a páros számokat.

7. Adjuk meg szöveges leírással a következő, halmaz formában megadott nyelvet: $L1 = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

8. Adjuk meg a matematikai leírását annak a nyelvnek, amely kettő hosszú szavakból áll, ahol a szavakat alkotó szimbólumok első eleme minden esetben 0, a második eleme pedig az angol ábécé tetszőleges szimbóluma.

4. fejezet

Generatív grammatikák

4.1. Generatív grammatikák

Az eddigiekben a nyelvek egyszerű definíciójával foglalkoztunk. Ennek segítségével egy formális nyelvet úgy definiálhatunk, hogy mint halmazt felfogva felsoroljuk az elemeit, a szavakat. Ezzel a módszerrel azonban kis elemszámú, véges nyelvek definiálhatók csak. A "nagyon sok", illetve végtelen sok szót tartalmazó nyelvek azonban felsorolással nem adhatók meg. A fentiekben is volt példa végtelen nyelv megadására, de a "generáló" szabályt egyszerű szöveges leírással, vagy bonyolult matematikai jelölések, és szabályok segítségével adtuk meg.

Ahhoz viszont, hogy később a gyakorlati alkalmazás felől is megközelíthessük a nyelveket, és a hozzájuk tartozó nyelvtanokat, találnunk kell egy általános, és használható módszert a nyelvek definiálására.

A definíciónak tartalmaznia kell a nyelv elemeit alkotó jeleket, a nyelvtant, amely alapján a nyelvi elemek egymáshoz illeszthetőek, valamint olyan segéd változókat, amelyek segítségünkre lesznek a nyelvi elemek összeállításában, vagy éppen azok ellenőrzésében.

Ellenőrzés alatt itt annak a módszernek az alkalmazását értjük, amely módszer egy adott mondatról, vagy szóról eldönti, hogy az eleme-e egy nyelvnek.

Ez azért lesz fontos a számunkra, mert az informatikában a formális nyelveket, és a hozzájuk tartozó elemző algoritmusokat leginkább a nyelvek, pontosabban a programozási nyelvek elemzésére használjuk, és ahhoz, hogy ezt meg tudjuk tenni ismernünk kell a különböző nyelvek, nyelvtanok felépítését, és azt, hogy egy speciális elemzési probléma megoldására (lexikális, szintaktikai, valamint szemantikai elemzések) a nyelvtanok mely csoportját kell majd felhasználnunk.

Amennyiben ismert számunkra a szükséges nyelvtan, az elemzési módszer, és a hozzá tartozó algoritmust, nem kell azzal foglalkoznunk egy egy programozási nyelv szintaktikai elemzőjét hogyan kell elkészítenünk.

21. Definíció (Generatív grammatika). *Egy $G(V, W, S, P)$ formális négyest **generatív grammatikának** nevezzük, ahol:*

- *V : a terminális jelek ABC-je,*
- *W : a nemterminális jelek ABC-je,*
- *$V \cap W =$, vagyis a két halmaz diszjunkt, nincs közös elemük,*
- *$S : S \in W$ egy speciális nemterminális jel, a kezdőszimbólum,*
- *P : helyettesítési szabályok halmaza, ha $A := V \cup W$, akkor $P \subseteq A*xA*$*

Ha $V := a, b$ és $W := S, B$, és $P := (S, aB), (B, SbSb), (S, aSa), (S, \epsilon,$ akkor a $G := (V, W, S, P)$ négyes egy nyelvet ír le. Ennek a nyelvnek a szavai a "a" és "b" jelekből írhatók fel. Az "S" és "B" jelek nem szerepelnek a végső szavakban, csak a szavak létrehozása közben mintegy "segédváltozók", köztes állapotokban használt jelek. A segédváltozók jelei csupa nagybetű (S, B), míg a nyelv terminális jelei csupa kisbetűvel vannak jelölve (a, b). A "aaSabSb" jelsorozatról azonnal látszik, hogy még "nincs befejezve", hiszen még tartalmaz változókat. A változók "nem terminális" jelek, vagyis nem befejező jelek, hiszen a szó generálása nem befejezett. Az "aaabb" jelsorozat viszont már csak "terminális" jeleket tartalmaz, melyek a "befejező", szimbólumok, vagyis elemei a nyelv szavait alkotó ábécének. Mivel az előző jelsorozat már nem tartalmaz "nemterminális" jeleket, csak csupa terminális jelet, ezért a szó generálása befejezettnek tekinthető. A fenti P halmaz egy Descartes szorzat, melynek minden eleme egy páros, pl. (S,aB) alakú. Az továbbiakban ezt inkább $S \to aB$ módon fogjuk jelölni. Ennek megfelelően a fenti P halmaz az alábbi módon írható fel:

$$P := \{(S \to aB), (B \to SbSb), (S \to aSa), (S \to \epsilon)\}.$$

Nézzük meg azt, hogyan lehet alkalmazni a fent leírt definíciót egy konkrét példára. Ahogy láthatjuk a szavak generálását a kezdőszimbólumtól (startszimbólum) kezdjük el. Mivel az "S" egy nemterminális szimbólum, a generálást folytatnunk kell mindaddig, amíg találunk a generálandó kifejezésben nemterminális jeleket. A generálás során két eset állhat elő.

- Az első, mikor a generálás során olyan előállítandó szimbólumhoz (terminális jel) jutunk, amelyhez nem találunk szabályt. Ilyenkor el kell gondolkoznunk azon, hogy a a generálandó szó nem eleme a nyelvnek, vagy rosszul végeztük a helyettesítést.

- A második eset, mikor találunk helyettesítési szabályt, és alkalmazzuk. Az szabály alkalmazás azt jelenti, hogy a megfelelő nemterminális jelet, amit helyettesíteni akarunk a szó építése közben kicseréljük a szabályra (felülírjuk vele).

  pl.: az aSa sorozatból a z S-et helyettesítjük az $S \to \varepsilon$ szabállyal. Ekkor a mondat a következő formát ölti: aa, mivel az üres szóval helyettesítettük a nemterminális S szimbólumot.

A következő helyettesítés az $A \to aB$. így most $A \to aB$ alapján a a generált mondat az "aB". "B" egy nemterminális jel, így folytatnunk kell a szó generálását.

A $B \to SbSb$ szabály alkalmazásával lépjünk tovább. Ebben a köztes állapotban kétszer is szerepel az "S" szimbólum, mint nemterminális jel, így két szálon is folytatnunk kell a szó generálását.

Ezen a ponton észrevehetjük, hogy a szó generálása gyakorlatilag egy szintaxisfa építésével egyenértékű. A szintaxisfának a csomópontjai (elágazásai) a nemterminális szimbólumok, és a levélelemein a terminális szimbólumok megjelenését várjuk el. Amennyiben a levél elemeken balról jobbra haladva összeolvassuk a terminális jeleket, és a jelekből előáll az eredeti, vagyis a generálandó szó, a helyettesítés helyes volt, és a szó előállítható a nyelvtan segítségével.

Helyettesítsük be az első "S" nemterminálist az $S \to aSa$-t, vagyis alkalmazzuk az $aSbSb \to aaSabSb$ szabályt az első "S" nemterminálisra. Ezen a ponton, és az összes többi nemterminális szabályokkal való felcserélése során többféleképpen is folytathatjuk az eljárást, és így a nyelvtan segítségével sokféle szót elő tudunk állítani. Ezért hívják ezt a rendszert generatív grammatikának, vagy produktív nyelvtannak. A nyelvtan által generált szavak alkotják a generált nyelvet, és ezáltal az összes, a nyelv elemeit alkotó szót elő lehet állítani segítségével.

A generatív grammatikában a legfontosabb elem a helyettesítési szabályok halmaza. A másik három elem a szabályokat felépítő jelsorozat építésében segít, és megadja, hogy melyik a startszimbólum, és mely jelek terminálisak, és melyek nemterminálisok.

A szavak, vagy a mondatok ilyen módon történő építését generálásnak nevezzük. A grammatikák gyakorlati alkalmazása során azonban nem az az

elsődleges cél, hogy szavakat építsünk, hanem az, hogy a nyelvtanok szabályainak alkalmazásával el tudjuk dönteni, hogy egy adott szó eleme-e egy nyelvnek. Ezt a fajta elemzést kétféleképpen végezhetjük el.

- Az első módszer az, amely során az előző példához hasonlóan megpróbáljuk az elemzendő mondatot előállítani a nyelvtani szabályok valamilyen sorrendben történő alkalmazásával. ezt a módszer szintaxisfa építésnek nevezhetjük.
- A másik módszer az, mikor az elemezendő mondat szimbólumait megpróbáljuk helyettesíteni a nyelvtani szabályokkal. Ennél a módszernél az a cél, hogy eljussunk a start szimbólumhoz. Amennyiben ez sikerül, a mondat eleme a nyelvnek, másképpen helyes. Ezt a módszert redukciónak nevezzük, és igazság szerint a gyakorlatban, vagyis a programnyelvek elemző algoritmusainál ez az inkább alkalmazható módszer.

Most, hogy tisztában vagyunk a grammatikák alkalmazásának különböző módszereivel, vizsgáljunk meg néhány, a levezethetőségre vonatkozó szabályt.

22. Definíció (Levethetőségi szabály). *Legyen adva $G(V, W, S, P)$ generatív grammatika, és $X, Y \in (V \cup W)*$. X-ből Y levezethető, ha $X \Rightarrow +Y$ vagy $X = Y$. Ennek a jele: $X \Rightarrow *Y$.*

23. Definíció (Egy lépésben levezethető). *Legyen adott $G(V, W, S, P)$ generatív grammatika, és $X, Y \in (V \cup W)*$. $X = \alpha\gamma\beta$, $Y = \alpha\omega\beta$ alakú szavak, ahol $\alpha, \beta, \gamma, \omega \in (V \cup W)*$. Azt mondjuk, hogy X-ből Y* ***egy lépésben levezethető****, ha létezik $\gamma \to \omega \in P$. Ennek a jele: $X \to Y$.*

HA fenti definíció akkor érvényes, ha X és Y két szimbólumsorozat (szó, vagy mondat), melyekben szerepelnek terminális és nemterminális szimbólumok is, valamint X és Y a levezetés két szomszédos mozzanata. X -ből Y egy lépésben levezethető, vagyis X-állapotból Y-állapotba egy szabály helyettesítésével el tudunk jutni, ha az X-t felépítő γ jelsorozatot ki tudjuk cserélni a szükséges ω jelsorozatra. Ennek feltétele, ahogy a levezethetőségi szabálynál láthattuk az, létezzen ilyen szabály a grammatikában.

Az előző példánál maradva "aSbSb"-ból az "aaSabSb" egy lépésben levezethető. X="aSbSb", vagyis α="a", γ="S" és β="bSb". Folytatva Y="aaSabSb", vagyis ω="aSa". Mivel létezik $\gamma \to \omega$ szabály, ezért az X-ből Y egy lépésben levezethető.

6. Megjegyzés. *Vegyük észre, hogy α és β, a cserélendő szakasz előtti és utáni részszó lehet üres szó is.*

24. Definíció (Több lépésben levezethető). *Legyen adva $G(V, W, S, P)$ generatív grammatika, és $X, Y \in (V \cup W)*$. X-ből Y n lépésben levezethető ($n \geq 1$), ha $\exists X_1, X_2, ..., X_n \in (V \cup W)*$, hogy $X \to X_1 \to X_2 \to X_3 \to \ldots \to X_n = Y$. Jele: $X \to +Y$.*

Ahogy a szavak generálásánál láthattuk, a szavak építése közben használunk változókat, és terminális szimbólumokat egyaránt. A generált mondat ezek alapján kétféle lehet. Az első változat az, amelyben még vannak nemterminális jelek, a második pedig az, amelyben nincsenek.

25. Definíció (Mondatforma). *Legyen adva $G(V, W, S, P)$ generatív grammatika. A G egy $X \in (V \cup W)*$ **szót generál**, ha $S \Rightarrow *X$. Ekkor X-et **mondatformának** nevezzük.*

A mondatformát tehát vegyesen tartalmazhat terminális, és nemterminális jeleket. Ez valójában a levezetés köztes állapota, és akkor áll elő, mikor a generálás még nem fejeződött be. Mikor kész vagyunk a helyettesítésekkel, (szerencsés esetben) akkor kapjuk a mondatot.

26. Definíció (Mondat). *Legyen adva $G(V, W, S, P)$ generatív grammatika. Ha G egy X szót generál, és $X \in V*$ (vagyis nincsenek benne nemterminális jelek), akkor X-t mondatnak nevezzük.*

Most, hogy megvizsgáltuk a mondat, és a mondatforma közti különbségeket, definiáljuk a generatív grammatika által generált nyelvet, amely definíció praktikus okokból nagyon fontos a számunkra.

27. Definíció (Generált nyelv). *Legyen adva $G(V, W, S, P)$ generatív grammatika. Az $L(G) = \{\alpha | \alpha \in V*\}$ és $S \to \alpha$ nyelvet a G grammatika által **generált nyelvnek** nevezzük.*

A definíció jelentése, hogy a G startszimbólumából levezethető szavak alkotta nyelvet nevezzük a generált nyelvnek. Vagyis az L minden szavát le kell tudni vezetni S-ből, és az S-ből bármely levezethető szó eleme kell legyen a nyelvnek.

Ahhoz, hogy még jobban megértsük a szabályt, vizsgáljuk meg a $P_1 = \{S \to 00, S \to 11, S \to 0S0, S \to 1S1, S \to \epsilon\}$ szabályhalmazt, ahol 0,1 terminális szimbólumok.

A szabályrendszer segítségével levezethet. szavak az alábbiak: $L(G) =$ "","00","11","0000","010010",.... A $P_2 = \{S \to 0S0, S \to 1S1, S \to \varepsilon\}$ szabályrendszer által generált nyelv is ugyanezekből a szavakból áll, de a P_2 szabályrendszer egyszerűbb.

Mindezekből látható, hogy egy nyelvhez több generatív grammatika is tartozhat, és segítségükkel képesek vagyunk ugyanazokat a szavakat előállítani. Erre a tulajdonságra is tudunk szabályt alkotni, amely szabály a nyelvtanok ekvivalenciáját definiálja.

28. Definíció (Ekvivalencia szabály). *A $G_1(V, W_1, S_1, P1)$ generatív grammatika, valamint a $G_2(V, W_2, S_2, P_2)$ generatív grammatika* ***ekvivalensek****, ha $L(G_1) = L(G_2)$, vagyis a G_1 által generált nyelv megegyezik a G_2 által generáltal.*

Vagyis akkor a grammatikák akkor ekvivalensek, ha az általuk generált szavak ugyanazok. Természetesen a két nyelv csak akkor lehet ugyanaz, ha a terminális, és a grammatikák is akkor lehetnek ekvivalensek, ha terminális jelek azonosak (ezert nincs V_1 és V_2, csak V). De minden másban különbözhetnek egymástól, például, hogy milyen és hány terminális jelet tartalmaznak, mi a startszimbólum, és milyen szabályokból állnak.

A szabályrendszert felépítő párokat megfigyelve szabályosságokat figyelhetünk meg a nyelvtanokban és azok írásmódjában (alakjában). A szabályosságok alapján a nyelveket generáló grammatikákat, és ezzel együtt a nyelveket is osztályozhatjuk, a szabályostól a szabálytalanig.

Mivel azonban a szabályrendszer jellemzi magát a generált nyelvet is, így a nyelveket is osztályozhatjuk. Természetesen annál jobb, minél szabályosabb a leíró nyelvtan, hisz annál szabályosabb a nyelv is, és annál könnyebb a gyakorlatban alkalmazni különböző elemzési algoritmusok írásával.

Mielőtt azonban továbbmennénk ezen gondolat mentén, és belekezdenénk a nyelvek osztályozásába, vegyük észre azt is, hogy a nyelvek és ezáltal a nyelvtanok nem csak kis és nagybetűkből állnak, és nem azért generáljuk, vagy építjük őket, hogy a szabályokat bizonyíthassuk, vagy éppen el tudjuk magyarázni.

Ahogy azt már láthattunk, a grammatikák segítségével nem csak betűkből álló, egyszerű szavakat készíthetünk, hanem szavak halmazaiból álló mondatokat is.

Ilyen módon (és rengeteg befektetett munkával) élő nyelvek szabályait is összeállíthatjuk, mint ahogy azt a nyelvészek is megteszik olykor, vagy a programozási nyelvek alkotói a nyelvek szintaxisleírásainak megalkotására.

Mindezért, és hogy lássunk egy gyakorlatban használt példát, adjuk meg egy kitalált (játék) programozási nyelv szintaxisát leíró szabályrendszert anélkül, hogy a szintaxis leírás eszközeit ismernénk. A nyelv csak néhány egyszerű nyelvi konstrukciót tartalmaz, méghozzá pontosan annyit, hogy az imperatív nyelveknél megismert szekvencia, szelekció, és iteráció megvalósítható legyen a segítségével, és tartalmaz még néhány egyszerű numerikus és logikai operátort, hogy a kifejezés szintaxist is meg tudjuk mutatni.

A nyelvtanban a szemléletesség kedvéért a lexikális elemeket, vagyis a terminális jeleket "" jelek közé tettük, és a nemterminálisokat nem nagybetűvel jelöltük. A nyelvtan tartalmaz olyan szabályokat is, amelyekben szerepel a | (logikai vagy) jel. Ezzel egyszerűen csak a szabály alkalmazásánál döntési lehetőséget vezetünk be, vagy másképpen összevonjuk a szabályokat egy szabállyá.

```
program   ::= "kezd" formok "vege"
formok    ::= form
              | formok
form      ::=  üresutasítás
              |"azonositó" "=" kifejezés
              |beolvas "azonosító"
              |ciklus kifejezés form ciklusvége
              |ha kifejezés akkor form havégére
              |kiír kifejezés

havégére  ::= form havége

kifejezés ::=  "azonosító"
              |"konstansszám"
              |"(" kifejezés ")"
              | kifejezés "+" kifejezés
              | kifejezés "-" kifejezés
              | kifejezés "*" kifejezés
              | kifejezés "/" kifejezés
              | kifejezés "<" kifejezés
              | kifejezés ">" kifejezés
              | kifejezés "<=" kifejezés
              | kifejezés ">=" kifejezés
              | kifejezés "!=" kifejezés
              | kifejezés "==" kifejezés
```

A nyelvtan szintaxis leírása tartalmaz olyan részeket is, amelyek nem képezik szabályként részét a nyelvtannak, csak elemei a szabályoknak. Ezek terminális szimbólumok, de olyanok, amelyek nem atomiak, hanem további elemekre bonthatóak. A szintaxisleíró nyelvtan szempontjából terminális jeleknek tekinthető elemek szerkezetét egy másik nyelvtantípussal tudjuk leírni, amelyet még nem definiáltunk, de ezt a következő fejezetben megtesszük.

Most kizárólag annyit tegyünk meg, hogy kiválogatjuk ezeket az elemeket, majd leírjuk a szerkezetüket, vagyis azt a nyelvtant, amely alapján ezek előállíthatóak.

Van néhány terminális jelünk, amelyeket nem szeretnénk tovább bontani. Ezek a "kezd", a "vége", a "ha", és mindazon terminálisok, amelyeket a nyelvtannal leírt programozási nyelv kulcsszavainak tekintünk.

Vannak viszont olyanok, mint a "konstansszám", és az "azonosító", amelyek sokféle értéket felvehetnek. A "konstansszám" felveheti a 123, de akár az 1224-es értéket is. Nyilván, ahogyan azt már korábban megállapítottuk, az ilyen sokféle elemből álló nyelveknél a felsorolásos módszer nem célravezető, ezért írjuk fel inkább a kifejezést, ami az összes ilyen számot előállítja (vagy másképpen generálja).

Az "azonosító" az valójában az összes változó, és ha lenne ilyen a nyelvtanban, akkor az összes függvény, valamint eljárás neve lehetne. Mindezért az azonosítóknak tartalmazniuk kell az összes kis és nagybetűs karaktert, az összes számjegyet, valamint lehetőség szerint aláhúzásjelet is. Egy lehetséges kifejezés, ami ezt mind leírja a következőképpen nézhet ki:

$$(a - zA - Z0 - 9\_)^{+}.$$

Ez a kifejezés a zárójelek között megadott szimbólumok bármelyikét tartalmazhatja akármilyen sorrendben, de legalább egyet ezek közül mindenképpen tartalmaznia kell.

Vegyük észre, hogy ez a halmaz a $+$ jellel a végén valójában a pozitív lezártja a halmazelemeknek.

Ugyanezen elv mentén most már könnyedén felírhatjuk a konstansok általános alakját is, amely a következőképpen nézhet ki:

$$(0 - 9)^{+}.$$

Ez a lezárt az összes számjegyből összerakható összes lehetséges számot lefedi, de hiányzik az elejéről az előjelet definiáló részkifejezés. Amennyiben ezt is hozzávesszük a kifejezéshez, az z alábbi módon nézhet ki:

$$(+| - |\varepsilon)(0 - 9)^{+}.$$

Ez a kifejezés már megengedi az előjeles, és az előjel nélküli számok írását is a majdani programozási nyelvben, amelyet a szintaxis, és a lexikális elemek ismeretében már könnyedén elkészíthetünk, de az implementációhoz azért nem árt megismerkednünk a nyelvtanok különböző típusaival, és az ezekhez tartozó elemző automatákkal.

A fenti szintaxisleíró nyelvtanban a többi lexikális elem (nemterminális) önmagát definiálja, vagyis kizárólag önmagára illeszkedik, definiálnunk nem kell őket, ezért nem foglalkozunk velük részletesen.

Az ebben a példában ismertetett nyelvtani leírások, vagyis a szintaxis leírás, és a lexikális elemek definiálása mellett egy általános célú programozási nyelv esetén a szintaxis, és a lexikai elemek mellet meg kell tudnunk fogalmazni a nyelvi elemek által közölt jelentésbeli problémákra vonatkozó szabályokat is. Ezt szemantikának nevezzük, és a leírására szintén egy új nyelvtantípus bevezetésére lenne szükségünk. A szemantika, amely a programozási nyelven leírható műveletek működéséhez kapcsolódó axiómák egyszerű leírására szolgál, és arra, hogy segítségével elemezhessük a szemantikai problémákat egy adott programban bonyolult feladat, és jelen jegyzet terjedelmi korlátai miatt erre nincs lehetőségünk.

4.2. Nyelvtanok csoportosítása

Ahogy az előző példában láthattuk, a különböző elemzési problémák, és a különböző típusú nyelvek leírásához minden esetben másfajta nyelvtani leírásokat alkalmaztunk. Nézzük meg, hogy ezek a nyelvtanok, vagy pontosabban a nyelvek megadásához használt generatív grammatikák miben különböznek egymástól.

Két azonos nyelv (ekvivalens grammatika) esetén a terminális jeleknek meg kell egyezniük. Ezt nem kell bizonyítanunk, hiszen ha két nyelv terminális jelei nem egyeznek meg, akkor a nyelvtani szabályokkal nem lehet ugyanazokat a szavakat előállítani.

A nemterminálisokra nincs ilyen kikötés, de a két ekvivalens nyelvtanban ha a szabályok száma eltér, valószínűleg a nemterminálisoké is el fog.

A startszimbólum valójában lényegtelen ebből a szempontból, az lehet ugyanaz, és akár különbözhet is két nyelvtannál, mivel kizárólag annyi a szerepe, hogy a helyettesítési szabályokat ebből a jelből kiindulva kezdjük el alkalmazni a mondatok generálásának érdekében.

A grammatikákban található szabályok halmaza a legfontosabb elem. Ez a halmaz, amiben a grammatikák a leginkább eltérést mutathatnak. Ezen a ponton nem is a szabályok halmaza a lényeges, hanem a benne található szabályok formája, mivel ezek alapján különböztetjük meg, és szintén ez alapján tudjuk csoportosítani a nyelveket (a nyelvtanokat, és a grammatikákat).

A szabályok alakját figyelembe véve a nyelveknek négy nagy csoportja létezik, amely csoportokba kivétel nélkül az összes nyelv besorolható.

4.3. Nyelvtanok Chomsky-féle osztályozása

Legyen adott egy $G(V, W, S, P)$ generatív grammatika, és $\alpha, \beta, \gamma \in (V \cup W)*$ mondatformák, amelyek lehetnek akár ϵ értékűek is, és legyen adott $\omega \in (V \cup W)+$ mondatforma, amely nem lehet ϵ, Szükség van még az $A, B \in W$, nemterminális jelekre, és $a, b \in V$ terminális jelekre.

mindezek alapján felírhatjuk az alábbi, a nyelvtanok különböző osztályaira vonatkozó alábbi definíciókat:

29. Definíció (Mondatszerkezetű nyelvek). *Egy tetszőleges nyelv 0-ás típusú, vagyis* ***mondatszerkezetű*** *(phrase-structure) nyelvnek tekinthető, ha nyelvet generáló nyelvtan minden szabálya $\alpha A\beta \to \gamma$ alakú.*

A mondatszerkezetű nyelvek szabályrendszere nagyon megengedő, valójában nem is tartalmaz semmi komoly kikötést arra vonatkozóan, hogy milyen szabályokat alkalmazhatunk. A nyelvek ezen csoportjába tartoznak a ma is beszélt, vagy korábban használt élő nyelvek. Ezek elemzése nagyon a "szabályok szabálytalansága" miatt. Az ilyen nyelvek közötti konverzió, vagyis a fordítás is komoly erőforrások megmozgatását igényli még az olyan bonyolult automaták számára is mint az emberi agy.

A 0-ás típusú nyelvek esetén lényegében nincs megkötés a szabályokra, ugyanis az, hogy a szabályrendszer bal oldalán állnia kell legalább egy nemterminális szimbólumnak, ez a szabályrendszer használatóságából fakadó megkötés.

30. Definíció (Környezetfüggő nyelvek). *Egy tetszőleges nyelv 1-es típusú, vagyis* ***környezetfüggő*** *(context sensitive) nyelvnek tekinthető, ha nyelvet generáló nyelvtan minden szabálya $\alpha A\beta \to \alpha\omega\beta$ alakú, és megengedett az $S \to \epsilon$ szabály használata.*

A környezetfüggő nyelvtanok szabályai, mint az $\alpha A\beta \to \alpha\omega\beta$ szabály esetén, ha adott a bal oldalhoz hasonló mondatforma, és éppen az A nemterminálist szeretnénk helyettesíteni, akkor bármilyen szabályt alkalmazva a mondatforma eleje α, valamint a nemterminális A bal oldalán található β résszavak nem változhatnak meg.

Ha másképpen nézzük a dolgot, akkor ez azt is jelenti, hogy ahhoz, hogy egy adott szót, vagy résszót helyesnek találjunk, ismernünk kell annak környezetét. Innen is ered a nyelvek ezen csoportjának a "környezetfüggő" elnevezése.

Nézzünk erre egy gyakorlati példát. Ahhoz, hogy valamely általános célú programozási nyelven írt programunkban az $a = 2$ értékadás szemantikai,

vagyis jelentésbeli helyességét meg tudjuk állapítani, ismernünk kell az értékadás előzményeként írt deklarációt, amely az a változót bevezeti, és az ott leírt típust. Amennyiben a változó típusa egész szám, az értékadó utasítás megfelelő, ha nem, akkor hibát találtunk annak ellenére is, hogy ez az utasítás szintaktikailag helyes.

A környezetfüggő nyelvek, és a hozzájuk tartozó nyelvtanok a szemantikus elemzés eszközei a gyakorlatban, és a programozási nyelvek fordítóprogramjainak szemantikus elemzőjéhez ezt a nyelvtantípust implementálják.

Az 1-es típusú nyelvek esetén fontos, hogy a levezetés során a nemterminális szimbólum helyettesítése után a szimbólum környezete (előtte és mögötte levő részszó) ne változzon meg (α és γ). Mivel ekkor a szó lényegében csak hosszabb lehet ($A \to \omega$, és ω nem lehet üres szó, így minden nemterminális jelet legalább egy hosszú jelsorozatra kell cserélni), ezért ezeket a nyelvtanokat hossz nem csökkentő nyelvtanoknak nevezzük.

Ez alól kivételt képez az "S" startszimbólum, amelyet lehet üres szóra cserélni, ha az a szabály szerepel a szabályrendszerben. De a hossz nem csökkentés ekkor is fennáll, hiszen az S előtti és utáni résznek meg kell maradnia.

Az ilyen nyelvtanokban, ha a szó eleje már kialakult (a szó elején már csak terminális szimbólumok szerepelnek), az már nem fog megváltozni.

31. Definíció (Környezetfüggetlen nyelvek). *Egy tetszőleges nyelv 2-ás típusú, vagyis* ***környezetfüggetlen*** *(context free) nyelvnek tekinthető, ha nyelvet generáló nyelvtan minden szabálya $A \to \omega$ alakú, és megengedett az $S \to \epsilon$ szabály.*

A környezetfüggetlen nyelvek, ahogyan azt korábban már láthattuk, a szintaxis leírás, és a szintaktikai elemzés eszközei. A szabályok bal oldalán kizárólag egy nemterminális jel állhat, amely lehetővé teszi, hogy a programozási nyelvek szabályait definiálhassuk.

A környezet függetlenség miatt az ilyen nyelvek elemzése viszonylag egyszerű, és gyors, mivel az elemzés nem igényel visszalépéseket. Miután egy tetszőleges szöveg elejét elemeztük, azt akár el is dobhatjuk, mivel az elemzés további része nem függ a már elemzett szövegtől, és amit elemeztünk, az már biztosan helyes.

A fordítóprogramok szintaktikai elemzői ezt a nyelvtantípust használják a nyelvtani elemzések elvégzésére. Az elemzések során szintaxisfa generálást, vagy redukciót is használhatunk annak eldöntésére, hogy az elemzett mondat helyes, vagy sem. Az első esetben a generált szintaxisfa levélelemein jelennek meg az elemzett mondat szimbólumai, a második esetben pedig a terminálisok helyettesítése során a startszimbólumig kell redukálnunk az elemzendő szöveget.

A 2-es típusú nyelvek esetén fontos, hogy a szabályok bal oldalán csak egyetlen darab nemterminális jel állhat, amelyet valamilyen, legalább egy hosszú jelsorozatra kell cserélni. Itt nem kell figyelni a jel előtti és mögötti környezetre, csak a jelre. Ezek a nyelvtanokat hossz nem csökkentő nyelvtanoknak tekinthetjük.

32. Definíció (Reguláris nyelvek). *Egy tetszőleges nyelv 3-ás típusú, vagyis* ***reguláris*** *(regular) nyelvnek tekinthető, ha nyelvet generáló nyelvtan minden szabálya $A \to a$ és $A \to Ba$, vagy $A \to a$ és $A \to aB$ alakú. Az első esetben jobb reguláris, a másodikban bal reguláris nyelvről beszélünk.*

A 3-as típusú nyelvek esetén fontos, hogy a szó felépítése egy irányban halad. Először a szó bal oldala alakul ki, és az irány nem változik, hiszen a csere során újabb terminális szimbólumot teszünk a generált szóba, és egy újabb nemterminálisat a terminális jel jobb oldalára. Itt a köztes állapotokban nemterminális jelből egyszerre mindig csak egy lehet, és az mindig a köztes szó jobb végén található.

Ebben a nyelvtan típusban nincs kitétel a $S \to \varepsilon$ szabályra, mert a generálás bármikor abba hagyható.

Az viszont fontos, hogy vagy bal reguláris, vagy jobb reguláris szabályok szerepeljenek egy adott nyelvtanban, mivel, ha mindkét típusból fordul elő egyszerre, akkor a nyelvtan nem lesz reguláris.

A reguláris nyelvtanok fontos eszközei a fordítóprogramok lexikális elemző komponensének, de számos egyéb helyen is előfordulnak az informatika területein. Reguláris kifejezéseket használunk az operációs rendszerek parancsainál a keresésekhez használt feltételek definiálására, de a különböző programozási nyelvekben is ezeket alkalmazzuk a minta alapú keresések feltételeinek megfogalmazására. A reguláris nyelvek gyakorlati alkalmazásaival később még részletesen foglalkozunk.

4.4. A Chomsky-féle osztályozás fontossága

Ahogyan azt a fejezetben láthattuk, a különböző nyelvtanokat felhasználhatjuk a különböző típusú elemzési problémák megoldására. A környezetfüggő nyelvtanok a szemantikai elemzés megvalósítására alkalmasak, a környezetfüggetlen nyelvek a szintaktikai elemzés eszközei, és a reguláris nyelvek amellett, hogy a lexikális elemzésre használjuk őket, a legtöbb keresési művelet paramétereinek a megadására is igen hasznosak.

Amennyiben ismerjük a megoldandó probléma jellegét (lexikális, keresési, szemantikai, szintaktikai alapokon nyugvó probléma), akkor adott hozzá a

megoldás kulcsa is, vagyis a nyelvtan típus, valamint a nyelvtan típusa alapján a későbbiekben a megoldó algoritmus is (automata osztály, amely az adott nyelvtan elemzője). Így nem kell minden egyes problémánál újra kitalálnunk, hogy hogyan kell azt megoldani.

Másrészt az algoritmusok - mint ismeretes - problématípusok (problémaosztályok) megoldására szolgáló lépéssorozatok. A generatív grammatikákkal kapcsolatban több, számítógéppel megoldható probléma merül fel, melyekre algoritmusok készíthetőek. Nagyon fontos annak eldöntése, hogy ha egy konkrét generatív grammatika egy konkrét problémájának megoldására kifejlesztett algoritmus alkalmazható-e más grammatika ugyanazon problémájára, illetve mennyi módosítással alakítható át. Amennyiben a két grammatika ugyanazon Chomsky-osztályba tartozik, úgy egy jól megfogalmazott algoritmus váza a paraméterektől eltekintve elvileg alkalmazható lesz.

A harmadik fontos indok, ami miatt fontos ismerni egy grammatika Chomsky-osztályát, hogy igazolható bizonyos problémakörökről, hogy nem készíthető hozzá általános érvényű megoldó algoritmus.

4.5. Állítások a Chomsky-osztályokkal kapcsolatban

A nyelvek osztályozásából, és az ott bemutatott definícióknak van néhány következménye.

- A $G(V, W, S, P)$ egy bal reguláris nyelvtan $\Leftrightarrow$ ha $\exists$ $G'(V, W', S', P')$ jobb reguláris nyelvtan, úgy hogy $L(G) = L(G')$, vagyis a G, és a G' által generált nyelv megegyezik.

- Egy K nyelv i típusú ($i \in 0, 1, 2, 3$), ha létezik olyan $G(V, W, S, P)$ generatív grammatika, amelyre $L(G) = K$, és G szabályrendszere i. Chomsky osztályú.

Mindezek alapján K^2 esetén K egy környezetfüggetlen nyelv, mert létezik olyan grammatika, amely 2. Chomsky osztályú, és a K nyelvet generálja.

Mivel egy nyelvhez több generáló nyelvtan is tartozhat, azok elképzelhető, hogy különböző Chomsky osztályba tartoznak. Ezért a K^2 esetén K legalább környezetfüggetlen nyelv, de ha találunk egy olyan nyelvtant, amely reguláris, és ugyanezt a nyelvet generálja, akkor K-ról be lehet bizonyítani, hogy reguláris nyelv. Minél nagyobb sorszámú osztályba tudunk sorolni egy nyelvet, annál egyszerűbb a nyelvtan, és annál megengedőbb a nyelv.

Ezek alapján azt is megállapíthatjuk, hogy a nyelvek bizonyos szempontból részhalmazai egymásnak.

Azt is megfigyelhetjük, hogy a generáló nyelvtanok szabályaira egyre szigorúbb megkötések vannak, de a megkötések nem zárják ki egymást. Ha egy szabályrendszer megfelel a 2. osztály megkötéseinek, akkor megfelel az 1. és a 0. osztály megkötéseinek is (azok megengedőbbek). Ezért $L3 \subseteq L2 \subseteq L1 \subseteq L0$ állítás igaz.

Ha egy L nyelv $i \in \{0, 1, 2, 3\}$ típusú $\Rightarrow L \in \epsilon$ és $L\{\epsilon\}$ is i típusú, vagyis abban, hogy egy nyelv milyen osztályú, az ϵ-nak nincs semmilyen szerepe.

Nézzük meg az alábbi állítást. Ha L_1 és L_2 reguláris nyelv $\Rightarrow L1 \cup L2$ is reguláris. Ez azt jelenti, hogy , ha L1 reguláris, akkor tartozik hozzá egy $G_1(V, W_1, S_1, P_1)$ reguláris nyelvtan. Hasonlóképpen, L_2-hoz is tartozik egy $G_2(V, W_2, S_2, P_2)$ reguláris nyelvtan.

Ezek alapján feltételezhetjük, hogyW_1 és W_2 diszjunkt halmazok, vagyis nincs közös elemük (ellenkező esetben indexelhetjük a a közös elemeket). Azt is feltételezhetjük, hogy G_1 és G_2 is jobb-reguláris.

Az $L_1 \cup L_2$ nyelvben olyan szavak fordulnak elő, amelyek vagy L_1-nek, vagy L_2-nek elemei. Ha L_1-nek elemei, akkor van olyan szabály, amely $S_1 \rightarrow \ldots$ alakú. A jobb-reguláris nyelvtani szabályok szerint ezen szabály vagy $S_1 \rightarrow a$, vagy $S_1 \rightarrow B_1 a$ alakú. Hasonlóan, az L_2-beli szavakra van $S_2 \rightarrow a$ vagy $S_2 \rightarrow B_2 a$ alakú kezdő lépés.

Az állítás, amely szerint $L1 \cup L2$ is reguláris könnyen bizonyítható, ha sikerül egy olyan reguláris nyelvtant definiálni, amely épp az $L1 \cup L2$ nyelv szavait generálja.

Példaként legyen $G_3(V, W_3, S_3, P_3)$ egy olyan generatív grammatika, ahol $W_3 := W_1 \cup W2 \cup \{S_3\}/\{S1, S_2\}$, vagyis legyenek benne a W_1 és W_2 terminális jelei, kivéve az S_1 és S_2 eredeti startszimbólumokat, és vegyünk bele egy új jelet, az S_3-t, amely majd a G_3 startszimbóluma lesz.

az $S_3 \in W_3 P_3$ szabályait pedig építsük fel úgy, hogy vegyük az összes szabályt a P_1-ből, de minden S_1-t cseréljünk le S_3-ra, valamint hasonlóan a P_2 minden szabályát vegyük át, de S_2-t cseréljük le S_3-ra. Ez biztosítja, hogy a fent definiált W_3 nem terminális szimbólumhalmaz megfelelő lesz.

A fenti G_3 pontosan az $L1 \cup L2$ beli szavakat állítja elő, hiszen a fentiek szerint elkészített P_3 szabályrendszer az S_3-ból kiindulva tudja generálni mind az L_1-beli, mind az L_2-beli szavakat.

A P_3-beli szabályrendszer alakja viszont értelemszerűen szintén jobb reguláris, hiszen az $S_1 -> S_3$ és $S_2 -> S_3$-ra vonatkozó cseréktől eltekintve ezek a szabályok mindegyike jobb reguláris volt.□

- Ha L_1 és L_2 reguláris nyelv $\Rightarrow L1 \cap L2$ is reguláris.
- Ha L_1 és L_2 reguláris nyelv $\Rightarrow L1 * L2$ is reguláris.
- Ha L reguláris nyelv $L1*$ is reguláris.

- Minden véges nyelv 3-as típusú, vagyis reguláris.
- Létezik olyan 3-as típusú reguláris nyelv, amely nem véges.

Itt a harmadik pont némi magyarázatra szorul. Amennyiben L reguláris nyelv, akkor létezik olyan $G(V, W, S, P)$ generatív grammatika, amely reguláris, és éppen az L nyelvet generálja. Feltételezhetjük, hogy a P-beli szabályok jobb-regulárisak, és minden szabálya $S \to a$, $S \to Ba$ alakú.

Legyen $G'(V, W, S, P')$ grammatikában P' olyan szabályrendszer, ahol a grammatika V, W, S komponensei az eredeti G grammatikákból származnak, de a P' oly módon állítjuk elő, hogy vesszük P-beli szabályokat, és kiegészítjük az őket tartalmazó halmazt újabb szabályokkal úgy, hogy minden $S \to a$ alakú szabály esetén beleveszünk egy $S \to aS$ szabályt is.

Így a G' grammatika jobb-reguláris marad, és generálja az $L1*$ nyelv szavait. Ez utóbbiak ugyanis olyan szavak, amelyek L1-beli szavak n-szeres $(n = 1, 2, \ldots)$ konkatenációiból állnak elő.

4.6. Chomsky-féle normál alak

Def: Egy G(V,W,S,P) generatív grammatika Chomsky-féle normál alakban van, ha minden P-beli szabály $A \to BC$ vagy $A \to a$ alakú (ahol $A, B, C \in W, a \in V$).

Számos okból fontos lehet, hogy egy adott nyelvtant minimalizáljunk, vagyis megszabadítsuk az olyan felesleges nem terminális jelektől, amelyek egyetlen terminálló levezetésben sem jelennek meg.

Egy "A" nemterminális két okból lehet felesleges:

- az "A" nem vezethető be mondatformában, vagyis nincs olyan szabálysorozat, hogy az "S" mondatszimbólumból indulva valamilyen aAb mondatformát kapjunk (pl.: valamely $a, b \in (V \cup W)*$ szavakra).
- Az "A"-ból nem lehet terminálni, vagyis nincs olyan $a \in V*$, hogy $A \to *a$ létezne.

Az első típusba tartozó felesleges nemterminálisokat a következőképpen határozhatjuk meg egy $G = (V, W, S, P)$ nyelvtan esetén:

Legyen $U_0 = \{S\}$, és

$$U_i + 1 = U_i \cup \{A | \exists BaAb \in P, A \in W, B \in U_i, a, b \in (V \cup W)*\}, (i >= 0).$$

Ekkor, mivel W véges, létezik olyan i, hogy $U_i = U_i + 1$. Es ekkor a $W \neq U_i$ nemterminálisok feleslegesek, mert nem érhetőek el az S-ből kezdődő levezetésekkel.

A második típusba tartozó felelsleges nemterminálisok meghatározása egy $G = (V, W, S, P)$ nyelvtan esetén a következő (rekurzív) módon történhet:

Legyen $U_0 = \{A | legyen A \to a \in P, A \in W, a \in V*\}$, és

$$U_i + 1 = U_i \cup \{A | \exists A \to b \in P, A \in W, b \in (U_i \cup V)*\}, (i >= 0).$$

Ekkor szintén a W végessége miatt van olyan i, hogy $U_i = U_i + 1$, és ekkor az W U_i nemterminálisok feleslegesek, mert belőlük nem vezethető le terminális (vagy üres) szórész. Amennyiben az S nincs benne az U_i halmazban, akkor a generált nyelv üres.

Ha G nem üres nyelvet generál, akkor világos, hogy az így meghatározott felesleges nemterminálisokat, és az összes olyan szabályt, amely tartalmaz ilyen nemterminálist (akár a bal, akár a jobboldalán) elhagyva a kapott G' nyelvtan ekvivalens az eredetivel, és a termináló levezetések megmaradnak, így a generált nyelv nem változik.

33. Definíció (Normál alak). *Egy környezetfüggetlen nyelvtanról azt mondjuk, hogy Chomsky-féle normálalakban van, ha minden szabálya $A \to a$ vagy $A \to BC$ alakú, ahol $A, B, C \in W$ és $a \in V$.*

Minden ε - mentes környezetfüggetlen nyelv generálható olyan nyelvtannal, amely Chomsky-féle normálalakú, vagy másképpen minden G környezetfüggetlen nyelvtanhoz van olyan vele ekvivalens környezetfüggetlen nyelvtan, amely Chomsky normálalakú.

34. Definíció. *Minden, legalább 2-es típusú (környezetfüggetlen) Chomsky-osztályú nyelvtan átalakítható Chomsky-féle normál alakra.*

4.7. Feladatok

1. feladat: A 0,1 jelekből alkossunk olyan szavakat, amelyek páros hosszúak, és szimmetrikusak. Megoldás: $P := S \to 0S0, S \to 1S1, S \longrightarrow \epsilon$ Megj.: a fenti megoldás 2-es típusú. Megoldás: $P := S \longrightarrow 0B, B \longrightarrow S0, S \longrightarrow 1C, C \to S1, S \longrightarrow \epsilon$ Megj.: a fenti megoldás is 2-es típusú.

2. feladat: A 0,1,...,9 jelekből alkossunk pozitív egész számokat. Megoldás: $P := S \to 0S, S \to 1S, \ldots, S \to 9S, S \longrightarrow \epsilon$ Megj. a fenti megoldás 2-es típusú. Megoldás:

$$P := \{S \to 0S, S \to 1S, \ldots, S \to 9S, S \to 0,$$

$$S \to 1, ..., S \to 9\}$$

Megj.: a fenti megoldás 3-as típusú.

3. feladat: A 0,1,...,9 jelekből alkossunk pozitív egész számokat, de azok ne kezdődjenek 0-al. Megoldás: $P := \{S \to 1B, S \to 2B, \ldots, S \to 9B, S \to 1, S \to 2, \ldots, S \to 9, B \to 0, B \to 1, \ldots, B \to 9, B \to 0B, B \to 1B, \ldots, B \to 9B\}$ Megj.: a fenti megoldás 3-as típusú.

4. feladat: A x, *, + és a ( ) segítségével alkossunk komplex matematikai kifejezéseket. Megoldás:

$$P := \{S \to A, S \to S + A, A \to A * B,$$

$$A \to B, B \to X, B \to (S)\}$$

Megj.: a fenti megoldás 2-es típusú.

5. feladat: A 0,1 jelekből alkossunk olyan szavakat, amelyek tetszőleges (akár nulla) darab 0-al kezdődnek, és tetszőleges (akár nulla) darab 1-essel végződnek. Megoldás: $P := S \to 0S, S \to 1A, A \to 1A, A \to 1, S \to 0, S \to 1$ Megj.: a fenti megoldás 3-as típusú.

6. feladat: A 0,1 jelekből alkossunk olyan szavakat, amelyek tetszőleges (akár nulla) darab 0-al kezdődnek, és legalább ugyanennyi (vagy több) darab 1-essel végződnek. Megoldás: $P := S \to 0S1, S \to S1, S \to \epsilon$ Megj.: a fenti megoldás 2-es típusú. Megoldás: $P := S \to 0B, B \to S1, S \to \epsilon$ Megj.: a fenti megoldás 2-es típusú.

7. feladat: A 0,1 jelekből alkossunk olyan szavakat, amelyekben tetszőleges pozíción bár, de összesen páratlan darab 1-es szerepel. Megoldás: $P := S \to 0S, S \to 1A, S \to 1, A \to 1S, A \to 0A, A \to 0$ Megj.: a fenti megoldás 3-as típusú.

8. feladat: Az 1 jelből alkossunk páratlan darab betűből álló szavakat. Megoldás: $P := S \to 1A, A \to 1S, S \to 1$ Megj.: a fenti megoldás 3-as típusú.

9. feladat: Az 0,1 jelekből alkossunk legalább kettő darab 1-essel kezdődő szavakat. Megoldás:

$$P := \{S \to 1A, A \to 1B, A \to 1,$$

$$B \to 1B, B \to 0B, B \to 0, B \to 1\}$$

Megj.: a fenti megoldás 3-as típusú.

10. feladat: Az 0,1 jelekből alkossunk legalább kettő darab 1-essel végződő szavakat. Megoldás: $P := S \to 0S, S \to 1S, S \to 1A, A \to 1$ Megj.: a fenti megoldás 3-as típusú.

5. fejezet

Reguláris kifejezések

5.1. Reguláris kifejezések

Reguláris kifejezésekkel az informatika szinte minden területén találkozhatunk. Kiváltképp a számítógépeken működő operációs rendszerek konzol ablakaiban, ahol a különböző kereséseket végezzük. Ilyen reguláris kifejezés lehet, ha éppen a háttértárunkon keressük az összes *.txt* kiterjesztésű fájlt, vagy a k betűvel kezdődő, és *.doc* kiterjesztésű fájlokat. Az első esetben a $*.txt$, a másodikban a $k*.doc$ mintát alkalmazhatjuk az adott operációs rendszer parancssorában (a megfelelő parancs kiadását követően).

Az ilyen fajta keresések esetén nem a keresett szöveg minden előfordulását adjuk meg, hanem azok általános formáját, vagyis egy olyan mintát, amelyre az összes számunkra értékes kifejezés illeszkedik.

Gondoljuk csak végig, hogy milyen nehéz lenne megkeresni az összes olyan autómárkát egy viszonylag hosszú szövegben, amely autómárkák előtt valamilyen szín leírása található, de tegyük fel, hogy a színt számkóddal, vagy szövegesen is meg lehet adni, és nem feltétlenül használtak a színek leírásánál ékezetes betűket.

Ilyen, és hasonló esetekben nagyon hasznos a reguláris kifejezések használata, mivel a keresés kivitelezéséhez nem kell az összes lehetséges esetet leírnunk, csak azt a reguláris kifejezést, amely az összes lehetséges esetet leírja.

A keresések mellett a fordítóprogramok egyik fontos komponense is ezt a nyelvtan típust használja a nyelvi konstrukciók, vagyis a programelemek felismerésére. Ezt a komponenst lexikális elemzőnek nevezzük, és később még szót ejtünk róla.

5.2. Reguláris kifejezések, és a reguláris nyelvtanok

Mielőtt továbbmennénk, adjuk meg a reguláris nyelvek informális definícióját.

35. Definíció. *Legyen A egy abc, és V « A. Ekkor reguláris halmaznak tekintjük az alábbiakat:*

- *- $\emptyset$ üres halmaz*
- *- $\{\epsilon\}$ az egyelem., csakis az ϵ-t tartalmazó halmaz*
- *- $\{a|a \in V\}$ az egyetlen egy hosszú szót tartalmazó halmazt*

7. Megjegyzés. *A fenti halmazok formális nyelveknek tekinthetőek, hiszen szavak halmazai. Könnyű belátni, hogy a fentiek mindegyike 3-as típusú, vagyis reguláris nyelv.*

Ha P és Q egy-egy reguláris halmaz, akkor reguláris halmazok még az alábbiak is:

- a, $P \cup Q$ amely halmazt $P + Q$-val jelöljük,
- $b, \{ab|a \in P, b \in Q\}$, amit $P * Q$-val fogunk jelölni, és
- $c, P*$ is

A reguláris halmazokon értelmezett halmazműveleteket reguláris kifejezések létrehozásának nevezzük.

Pl: $L := (+ + - + \epsilon) \times (0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9) \times (0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9)*$

Vagyis a szó első karaktere vagy "+" vagy "-" vagy üres jel, második karaktere valamilyen 10-es számrendszerbeli számjegy, és a lista tetszőleges számú számjeggyel folytatódhat, vagyis ez nem más, mint a pozitív vagy negatív vagy előjel nélkül felírt 10-es számrendszerbeli számok nyelve.

8. Megjegyzés. *A fenti példában az egyszerűség kedvéért az egyelemű halmazokat a halmazképző jelek nélkül írtuk fel. A forma helyesen az alábbi módon kezdődik:*

$(\{+\} + \{-\} + \{\epsilon\}) \times (\{0\} + \{1\} + \{2\} + \{3\} \ldots$

de felírhatjuk az alábbi módon is:

- $A := \{+, -, \epsilon\}$
- $B := 0, 1, 2, 3, 4, 5, 6, 7, 8, 9$
- $L := A \times B \times B*$

5.3. Reguláris kifejezések alkalmazása

Ahhoz, hogy a reguláris kifejezések készítését a matematikai jelölésrendszerrel megadott formuláktól egészen az implementációig megismerjük, vegyünk az előző részben ismertetett egyszerű példát, amely az előjeles egész számokat írja le.

Vizsgáljuk meg a lehetőségeinket az általános forma megadásához. Az előjeles egészek leírva tartalmazhatják (kezdődhetnek) a +, valamint a − szimbólumokat, és az összes számjegyet, amely számjegyeket a $0, 1, \ldots, 9$ szimbólumokkal adhatunk meg.

(Praktikus okból, vagyis, hogy az implementáció ne legyen túlságosan bonyolult, azt az esetet kihagytuk, amikor a szám nem előjellel kezdődik.)

A számjegyeket érdemes egy halmazzal jelölnünk. Erre a momentumra az implementációnál még kitérünk.

A halmaz, amelyet N-nel jelölünk, a következőképpen adott:

$N := 0, 1, \ldots, 9$

Amennyiben azt az esetet nem vesszük, ha nem adnak meg előjelet, a reguláris kifejezés a következőképpen néz ki:

$$(+|-)N^+,$$

ahol a ()- zárójelek a csoportképzés eszközei, vagyis a zárójelek közé írt, | jellel elválasztott szimbólumok közül bármelyik szerepelhet azon a helyen, de egyszerre csak egy.

Az N a korábban említett halmaz, amely a számjegyeket leíró szimbólumokat tartalmazza, és a + jelentése, hogy a halmaz pozitív lezártját kell vennünk, vagyis a szimbólumokból alkotható összes lehetőséget. Ez az N halmaz pozitív lezártja, tehát a halmaz összes lehetséges hatványhalmazainak az uniója (lásd.: **??**. fejezetben).

A fenti reguláris kifejezést megadhatjuk a matematikai formulától eltérő módon is úgy, ahogyan az operációs rendszerek parancssorában, vagy a magas szintű programozási nyelvek esetén is megtehetjük.

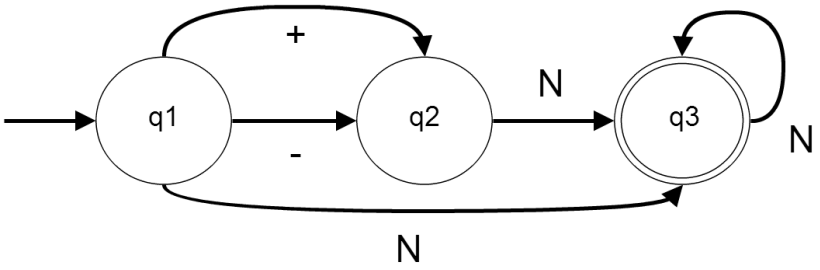
$$[+, -](0 - 9)+$$

Ez a leírás közel ugyanazt jelenti, mint a matematikai formula, de az írásmód, és a kifejezés alkalmazása eltér.

Ahhoz, hogy a reguláris kifejezést implementálhassuk, elsőként modellezünk kell a működését. A modellalkotás fontos lépése a tervezésnek, és az implementációnak. A matematikai modell segít a tervezésben, a megértésben és a verifikáció elvégzésének elengedhetetlen feltétele (verifikáció: a terv és a modell közti transzformációs lépések ellenőrzése).

Az elvont, absztrakciós modell viszont a matematikai modell megértéséhez, és az implementáció kivitelezéséhez elengedhetetlen eszköz.

A fenti kifejezés modellje a következő:



A **??**. ábrán látható irányított gráf ekvivalens a reguláris kifejezéssel, de már tartalmazza a kifejezést alkalmazó automata állapotait, és az állapotok megváltoztatásához szükséges állapotátmeneteket.

Vizsgáljuk meg a modellt, és írjuk fel azt az automatát, amely majd implementálható lesz valamely általunk választott magas szintű programozási nyelven.

Az automata mindezek alapján a következő módon írható le:

$$A(V, A, A_0, A_v, \delta),$$

ahol

- V halmaz, és az input ABC elemeit tartalmazza, amely elemek a következő szimbólumok: $\{+, -, 0, \ldots, 9\}$.
- Az A halmaz az automata belső állapotainak a halmaza.
- Az $A_0 \in A$ a kezdőállapot,
- Az A_v a befejező, vagy másképpen az elfogadó állapotok halmaza.
- a δ az a kétváltozós függvény, amelynek az első paramétere az automata aktuális állapota, a második pedig az aktuálisan vizsgált szimbólum.

(Az automatákkal a róluk szóló fejezetekben még foglalkozunk.)

Most viszont konkretizáljuk a felsorolásban szereplő elemeket az reguláris kifejezésre.

$$V = \{+, -, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$
$$A = \{q0, q1, q2\}$$
$$A_0 = \{q0\}$$
$$A_v = \{q2\}$$
$$\delta = \{(q0, +, q1), (q0, -, q1), (q1, N, q2), (q2, N, q2)\}$$

A δ függvény megadásánál két dologra kell felhívnunk a figyelmet. Az első, hogy a függvény állapotátmeneteinek megadásakor a rendezett hármasok első két eleme a függvény két paramétere, a harmadik pedig az eredménye, vagyis ha például a $(q0, +, q1)$ hármast vesszük, akkor a delta függvény a következő formát ölti:

$$\delta(q0, +) \rightarrow q1$$

A szintén a δ megadásánál az utolsó két állapotátmenet esetében a számjegyek helyett az azokat tartalmazó halmazt írtuk a második elem helyére. Ez az implementációnál lesz a segítségünkre, abban, hogy a lehetséges állapotátmenetek számát csökkenthessük (ez a lépés azt eredményezi, hogy a magas szintű nyelven írt program rövidebbé válik).

Az automata implementációja innentől nem bonyolult, mivel csak a δ függvényt kell elkészítenünk, és egy olyan ismétlést, amely segítségével a δ függvényt alkalmazni tudjuk a vizsgált szimbólumsorozat minden elemére.

Ahhoz, hogy el tudjuk készíteni az implementációt, elsőként határozzuk meg a leendő program változóit, vagyis az egyes állapotok értékeit tartalmazó attribútumokat.

Szükségünk lesz egy olyan változóra, amely az automata belső állapotait tartalmazza. Erre a célra a szöveg típust alkalmazhatjuk, hogy ne térjünk el nagyon a modellben alkalmazott $q0, q1, q2$ jelölésektől.

A hatékonyság érdekében bevezethetünk egy újabb állapotot. Ebbe az állapotba akkor kerül majd az automata, ha már az elemezés közben kiderül, hogy a vizsgált szimbólumsorozat nem megfelelő.

Mikor ebbe az állapotba kerül az automata, többé nem szabad kiengednünk belőle, mert ezen a ponton biztosan kiderült a hiba, és megállíthatjuk az elemző automata futását.

Szükségünk lesz ezen kívül egy szintén szöveg típusú változóra, amelyben az elemzendő szimbólumsorozatot tároljuk, és egy szám típusúra, amellyel a szöveg szimbólumait indexeljük.

Mielőtt azonban elkészítenénk a konkrét programozási nyelven írt változatot, adjuk meg program leíró nyelvi változatát.

```
SZÖVEG Állapot = "q0"
SZÖVEG InputSzalag
EGÉSZ CV = 0

FÜGGVÉNY delta(SZÖVEG Állapot0, SZÖVEG InputSzalagElem)
  SZÖVEG BelsőÁllapot

  VÁLASZT ÖSSZEFŰZ(Állapot, csere(InputSzalagElem))
    "q0+" ESETÉN BelsőÁllapot = "q1"
    "q0-" ESETÉN BelsőÁllapot = "q1"
    "q0N" ESETÉN BelsőÁllapot = "q1"
  KÜLÖNBEN
    BelsőÁllapot = Error
  VÁLASZTÁS VÉGE

  delta = BelsőÁllapot
FÜGGVÉNY VÉGE

BE : InputSzalag
EGÉSZ Hossz = HOSSZA(InputSzalag)

AMÍG Allapot != Error ÉS CV < Hossz ISMÉTLÉS
  Állapot = delta(Állapot, InputSzalag[CV])
ISMÉTLÉS VÉGE

HA Állapot !=  Error AKKOR
  KI : "Igen"
KÜLÖNBEN
  KI : "Az InputSzalag[CV] nem megfelelő"
HA VÉGE
```

Vegyük észre, hogy a leírónyelvi változatban az automatát kissé átalakítottuk a modellhez képest.

Az egyik változtatás, hogy a *delta* függvény ágaiban nem vettük számításba az N halmaz minden elemét, hanem csak a halmaz nevét. Ez nem fog problémát okozni az implementáció során. Az N elemei helyett ott is csak a halmaz nevét fogjuk használni de úgy, hogy mikor az input szalagról az N egy eleme érkezik a függvény sorrendben második aktuális paramétereként, azt egy függvény átalakítja, vagyis a az N karakter adja vissza. Az *ÖSSZE-*

FŰZ, a *HOSSZ* függvényekről, és a *HALMAZ* típusról feltételezzük, hogy az adott (implementációra használt) magas szintű nyelv tartalmazza azokat.

```
FÜGGVÉNY csere(SZÖVEG allapot0, SZÖVEG InputSzalagElem0)
  HALMAZ H = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
  HA ELEME(InputSzalagElem0, H) AKKOR
    csere = "N"
  HA VÉGE
  csere = InputSzalagElem0
FÜGGVÉNY VÉGE
```

Amennyiben ez a függvény is rendelkezésünkre áll, elkészíthetjük a teljes program imperatív, funkcionális, vagy objektum orientált változatát.

A teljesség kedvéért a fejezetben mindhárom megoldást ismertetjük.

Az első legyen az objektum orientált megoldás, amelyet C# nyelven adunk meg. Az elemző automata egy osztály, ahol az input szalag bejárása egy *while* ciklussal történik, és az N halmaz elemeit szövegtípusban tároljuk, és az $ELEME$ függvény helyett a szöveg (string) típus $IndexOf$ metódusát használjuk.

```
class elemzo
{

    public string state = "q0";
    private string input = "";

    public string Automata(string input)
    {
        this.input = input;
        int index = Elemzes();
        if (index != -1)
            return String.Format("Hibás karakter
                        (\"{0}\") a(z) {1}. indexnél",
            input[index], index);
        return "";
    }

    private int Elemzes()
    {
        int i = 0;
        while (i < input.Length && state != "error")
```

```
        {
            state = Delta(state, input[i]);
            i++;
        }
        if (state == "error")
            return i - 1;
        return -1;
    }

    private string Delta(string state0, char c)
    {
        string instate = "error";
        switch (state0 + Replace(c))
        {
            case "q0+":
            case "q0-":
                instate = "q1";
                break;
            case "q1N":
            case "q2N":
                instate = "q2";
                break;
            default:
                instate = "error";
                break;
        }
        return instate;
    }

    private string Replace(char jel)
    {
        string jelAsString = jel.ToString();
        int szam;
        if (Int32.TryParse(jelAsString, out szam)
                        && szam >= 0 && szam <= 9)
            return "N";
        return jelAsString;
    }
}

class Program
```

```
{
    static void Main(string[] args)
    {
        Parser parser = new Parser();
        string input = "+123F";
        string message = parser.Automata(input);
        Console.WriteLine("Az elemzendõ bemenet:
                                        {0}", input);
        Console.WriteLine("Elemzés eredménye: {0}.",
             message.Length > 0 ? message : "helyes");
    }
```

Az imperatív nem sokban különbözik az objektum orientált változattól. A különbség az osztály definíció hiányában van, és abban, hogy a használt típusok sem osztály alapúak.

```
string Allapot = "q0";
string InputSzalag;
int CV = 0;

string csere(string allapot0, string InputSzalagElem0)
{
   string H = "0,1,2,3,4,5,6,7,8,9";
   if (pos(InputSzalagElem0, H) > -1)
   {
     return "N";
   }
   return InputSzalagElem0;
}

string delta(string Allapot0, string InputSzalagElem)
{
  string BelsoAllapot;

  switch (Allapot + csere(InputSzalagElem))
  {
    case "q0+": BelsoAllapot = "q1"; break;
    case "q0-": BelsoAllapot = "q1"; break;
    case "q0N": BelsoAllapot = "q1"; break;
```

```
    default: Belsoallapot = "Error"; break;
  }
  return BelsoAllapot;
}

InputSzalag = "+123";
int Hossz = length(InputSzalag);

while (Allapot != "Error" && CV < Hossz)
{
  Allapot = delta(Allapot, InputSzalag[CV]);
}

if (Allapot !=  "Error")
{
  print("helyes");
}
else
{
  print("A \s szimbólum hibás", InputSzalag[CV]);
}
```

Miután megismertük az imperatív változatot, elkészíthetjük a funkcionális nyelvűt is. Ez a verzió a funkcionális nyelvek viszonylag nagy kifejező ereje miatt rövidebb lesz, de a megvalósítását a kedves Olvasóra bízzuk.

Mindhárom program úgy működik, hogy az input szöveg elemeit egyesével veszi, vagyis ezzel az adattal, és az aktuális állapotot tartalmazó változó értékével minden lépésben meghívja a *delta* függvényt, majd annak visszatérési értékét elhelyezi az állapotot tároló változóban (amelyet paraméterként is megkap).

Ezt a lépéssorozatot addig folytatja, amíg ki nem derül, hogy az input szöveg nem felel meg a szabályoknak, vagy amíg el nem éri a szöveg végét. Ekkor mindkét esetben csupán annyi a feladat, hogy megvizsgáljuk az program aktuális állapotát, vagyis azt, hogy az *Állapot* változóban milyen érték van.

Amennyiben az nem a "q2" állapotot, vagy az "Error" szöveget tartalmazza, az elemzett szöveg hibás, máskülönben, vagyis "q2" esetén a szöveg helyes (megfelel a kifejezés szabályainak).

Ezek a megoldások mind a korábban megadott reguláris kifejezés implementált változatai, és sajnos mindegyiknek ugyanez a tény jelenti a hiányos-

ságát is.

Mikor hiányosságot említünk, arra gondolunk, hogy mindhárom program kizárólag egyetlenegy reguláris kifejezést implementál.

Sokkal jobb megoldáshoz jutnánk, ha úgy készítenénk el a programot, hogy a nyelvtan, vagyis a reguláris kifejezést implementáló állapot átmeneteket leíró lépések paraméterként kerülnének a programba.

A pontosság kedvéért ez azt jelenti, hogy a delta függvény lehetséges paramétereit (állapot és az input szöveg aktuális eleme), valamint a hozzájuk tartozó visszatérési értékeket (állapotok) a program az elemzendő szöveg mellett szintén paraméterként kapja meg.

5.4. Feladatok

1. Készítsünk reguláris kifejezést, amely a programozási nyelvekben használatos egyszerű azonosító szimbólumokat ismeri fel. Az azonosító nem kezdődhet csak az angol ABC nagybetűivel, a második karaktertől kezdődően viszont felváltva tartalmazhat betűket és számokat, valamint aláhúzás jelet.

2. Implementáljuk azt az a reguláris kifejezést, amely az egyes feladatban szerepel, és az egyszerű azonosítókat ismeri fel.

3. Készítsünk reguláris kifejezést, amely aposztrófokkal határolt szöveg (string literál) általános formáját írja le. Az ilyen módon megadott szöveg nem tartalmazhat aposztrófokat csak párban, ezen kívül viszont bármilyen szimbólum szerepelhet benne.

4. Készítsük el az előző feladatban ismertetett reguláris kifejezés gráf reprezentációját is, vagyis definiáljuk a kifejezést felismerő automata állapotait és állapotátmeneteit.

5. Adjuk meg azt a reguláris kifejezést, amely az összes előjeles, és előjel nélküli egész, valamint valós számot leírja. Vegyük figyelembe, hogy az ilyen kifejezés a számok normál alakú formáját is definiálja.

6. Írjunk programot, amely a bemenetére érkező szöveget elemzi, és felismeri benne az előjeles, és előjel nélküli egész számokat. A program nagyon hasonlít a fejezetben bemutatott programhoz annyi különbséggel, hogy ez a változat felismeri az előjel nélküli számokat is.

7. Készítsük el az előző feladatban ismertetett program leíró nyelvi változatát.

6. fejezet

Környezetfüggetlen nyelvek és szintaxisfa

6.1. Környezetfüggetlen nyelvek alkalmazása

A Chomsky 2-es típusú, vagyis a környezetfüggetlen nyelvtanok, ahogy azt a nyelvek osztályozásánál láthattuk, a szintaktikai elemzés eszközei. Ezért a gyakorlatban is leginkább a programozási, és egyéb nyelvek nyelvtani leírásinál alkalmazzák őket, valamint a programozási nyelvek fordítóprogramjainak a nyelvtani elemző komponensében.

Bővítsük tovább a környezetfüggetlen nyelvtanokról szerzett tudásunkat néhány új szabály bevezetésével.

- Egy $\alpha \in V*$ szó pontosan akkor levezethető S startszimbólumból, ha konstruálható hozzá levezetési fa (szintaxis fa).
- Egy 2-es típusú generatív grammatika nem egyértelmű, ha létezik olyan α szó amely szót a grammatika generál $L(G)$, és amelyhez két különböző (nem izomorf) levezetési fa is konstruálható.
- Egy 2-es típusú nyelv kizárólag akkor egyértelmű, ha egyértelmű generatív grammatika segítségével definiálható.

A fenti állítások lényege, hogy segítségükkel eldönthető az adott szóról hogy az generálható-e az adott nyelvtannal. Hogy ezt a mondatot megválaszolhassuk, konstruálnunk kell a szóhoz levezetési fát. A levezetési fát a konstruálást befejezve balról jobbra, felülről lefele olvasva amennyiben a levél elemeken a vizsgált szó jelenik meg (az ϵ-nak nincs jelentősége a levélelemeken), akkor az eleme a grammatika által generált nyelvnek.

A levezetési fán egy helyettesítéssel egyszerre több új nemterminális jel is megjelenik. A továbbhaladás szempontjából választani kell, hogy melyiket helyettesítsük tovább. Ez a mozzanat döntést igényel, és számos esetben egy szóhoz több levezetési fa is konstruálható.

Ez a lépés a második, és a harmadik állítás miatt érdekes, mivel azok azt mondják ki, hogy csak akkor egyértelmű a grammatika, ha nem találunk egy adott szóhoz két különböző levezetési fát.

- Amennyiben a levezetési fa konstruálása közben mindig a legbaloldalibb nemterminális szimbólumot helyettesítjük tovább, úgy a levezetést kanonikusnak, vagy baloldali levezetésnek nevezzük.

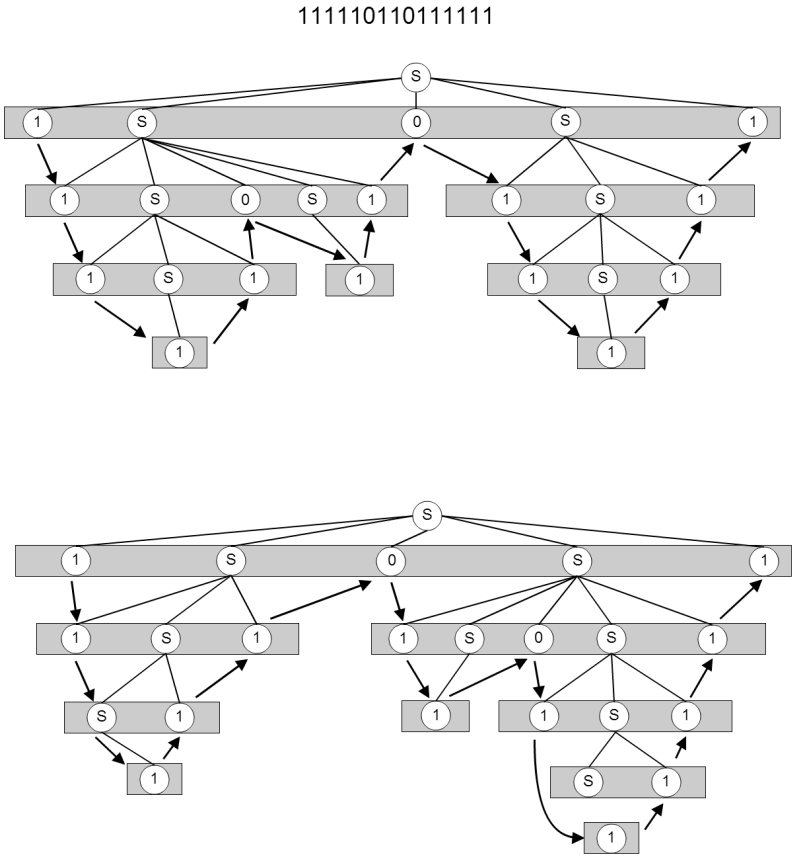
- Egy generatív grammatika nem egyértelmű, ha létezik olyan $\alpha \in L(G)$ szó amelyhez legalább két kanonikus levezetés is tartozik.

- Amennyiben a levezetési fát felülről lefelé a start szimbólumból kiindulva konstruáljuk, úgy ezt a folyamatot top-down stratégiának, vagy szintaxisfa építésnek nevezzük.

- Amennyiben a levezetési fát alulról felfelé elemezzük a szabályok behelyettesítésével, úgy a folyamatot bottom-up stratégiának, vagy redukciónak nevezzük.

A szintaxisfa építése hossz nemcsökkentő stratégia, mivel a mondatforma folyamatosan bővül. Ha egy konkrét szóhoz konstruálunk levezetési fát, s amennyiben a levélelemek között szereplő terminális jelek száma már több, mint a szó hossza, úgy abbahagyhatjuk a konstruálást, mert az már biztosan nem fog sikeresen befejeződni.

Ekkor azt is megtehetjük, hogy nem dobjuk el a teljes szintaxis fát, hanem visszalépünk, és próbálunk más helyettesítéseket használni. Ennek a módszernek a neve visszalépéses elemzés, bár nem túl hatékony a gyakorlati alkalmazások területén, mert a visszalépések miatt lassú az elemző algoritmusa.



Amennyiben nem visszalépéseket alkalmazunk, hanem kanonikus levezetést, a szó eleje (baloldali része) folyamatosan kialakul. Amennyiben a szó eleje a mondatformában a terminális szimbólumokon eltér a generálandó szótól, akkor a levezetés biztosan rossz, ezért más utat kell választani.

Vizsgáljunk meg néhány állítást arra nézve is, hogy az általunk környezetfüggetlennek tekintett nyelvek valóban azok-e, és arra is, hogy ezt el lehet-e eldönteni.

- Ha L_1, L_2 környezetfüggetlen nyelv akkor $\Rightarrow L_1 \cap L_2$ nem biztos, hogy szintén környezetfüggetlen, mivel két halmaz metszete lehet véges is, és egy korábbi állításunk alapján minden véges nyelv reguláris.

- Ha L_1, L_2 környezetfüggetlen nyelvek, $L_1 \cup L_2$ is környezetfüggetlen nyelv.

- Ha L környezetfüggetlen nyelv $\Rightarrow L*$ is környezetfüggetlen nyelv.

Sajnos algoritmikusan eldönthetetlen, hogy egy kettes nyelv, vagy egy olyan nyelvtan, amely egy kettes nyelvet generál egyértelmű-e vagy sem, mi-

vel nincs általánosan használható algoritmus, amely eldönti a problémát, de azt tudjuk, hogy minden reguláris nyelv és nyelvtan egyértelmű.

Annak eldöntésére, hogy egy nyelvtan kettes, vagy hármas típsú, a szabályok alapján tudunk következtetni, de vegyük figyelembe azt is, hogy a nyelvek egymás részhalmazai, vagyis a 0-ás típusú nyelvektől haladva, a szabályokat egyre szigorítva eljutunk a reguláris nyelvekig.

Mindezért ha találunk egy nyelvet amely reguláris, az megfelelhet a többi nyelvtan szabályainak is, mert azok sokkal megengedőbbek. Ezért számos esetben azt mondjuk, hogy egy nyelv legalább i. osztályú, és nem azt, hogy pontosan i. osztályú ($i \in \{0, 1, 2, 3\}$).

6.2. Szintaxis elemzők

A szintaktikus elemzők a fordítóprogramok azon részét alkotják, amely eldönti az adott programozási nyelven írt programról, hogy az nyelvtanilag helyes-e, vagyis, hogy megfelel-e a nyelvi szabályoknak. Pontosabban a szintaktikus elemzők feladata valamely jelsorozat levezetési fájának előállítása.

6.3. Rekurzív leszállás módszere

Erre számos módszer létezik. Az egyik ilyen algoritmus a rekurzív leszállás módszere, amely módszer a magas szintű programozási nyelvek függvény hívási mechanizmusát használja ki. A lényege, hogy a kettes típusú nyelvtanok minden szabályához egy eljárást rendel.

Például az $E \to Ea$ szabályhoz a

```
eljaras E()
kezd
  E()
  leptet(a)
vege
```

eljárást, amely tartalmazza az elemzendő szöveg bejárására, valamint az aktuálisan vizsgált terminális jel elfogadására alkalmas eljárást:

```
eljaras leptet(szimbolum)
kezd
  ha szimbólum == inputszalag[i] akkor
     i = i + 1
  különben
```

```
      hiba
   ha vége
 vége
```

Amennyiben az i egy globális változója a programnak, és az input szalagon balról jobbra fel van írva az elemzendő szöveg, valamint rendelkezésünkre áll az összes szabályhoz tartozó eljárás, akkor csak meg kell hívnunk a nyelvtan startszimbólumához tartozó eljárást, amely rekurzívan, vagy csak szimpla hívással meghívja sz összes többi szabályhoz tartozó eljárásokat.

Ha a hívási láncban az összes eljáráshívás lefutását követően visszatérünk a start szimbólumhoz, akkor az elemzett szó helyes, egyébként a hiba helye minden szabály eljárásában adott az i index által (pontosan az "inputszalag" i-edik eleme tartalmazza a helytelen szimbólumot).

Természetesen ez, és az összes többi eljárás csak akkor készíthető el, ha a nyelvtan egyértelmű, és minden szimbólum helyettesítéséhez találunk szabályt, és pontosan egyet.

6.4. Early-algoritmus

ez az algoritmus lényegében megegyezik az egyszerű visszalépéses elemző algoritmussal. A startszimbólumból indulva minden lehetséges helyettesítést megvizsgál. Amennyiben valamely helyettesítés során egy terminális szimbólum kerül a levezetett mondatforma elejére, úgy ellenőrzi, hogy az megfelelő-e. Amennyiben nem, ezt a mondatformát eldobhatjuk. Minden megtartott első lépésű továbbhelyettesítést külön-külön megvizsgálunk, és megpróbáljuk ezekből kiindulva az összes lehetséges továbbhelyettesít kipróbálni. Hasonlóan szűrjük ki a hibás útvonalakat. Amennyiben célhoz érünk, úgy a szó előállítható, és ismert a helyettesítési sorozat.

6.5. Coke-Younger-Kasami (CYK) algoritmus

Ez az algoritmus csak akkor használható, ha a nyelvtan Chomsky-normál alakban van. Mivel minden környezetfüggetlen nyelv felírható normálalakra, ezért ez az algoritmus használható nyelvosztályra. Igazság szerint az algoritmus azt mondja meg, hogy a normál alakú nyelvtan mely szabályainak sorozatával vezethető le az adott mondat, és nem azt, hogy az eredeti nyelv szabályaival, de ebből már következtethetünk a megoldásra.

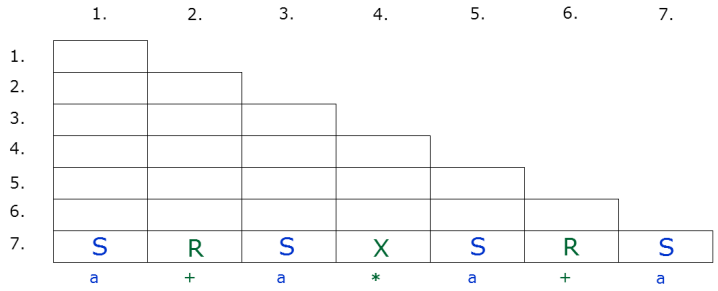
Az elemzés alulról-felfelé halad, és egy alsó háromszögmátrixot állít elő, amely celláiba a normál forma nemterminális szimbólumai kerülnek.

Nézzünk meg egy konkrét példát az elemző működésére. Legyen adott a $G(V, W, S, P)$ generatív grammatika, ahol a nemterminális szimbólumok a következők: $W = \{S, A, M, K, P, R, X, Z\}$. A terminális szimbólumok halmaza $V = \{a, (, +,)\}$, és a helyettesítési szabályok:

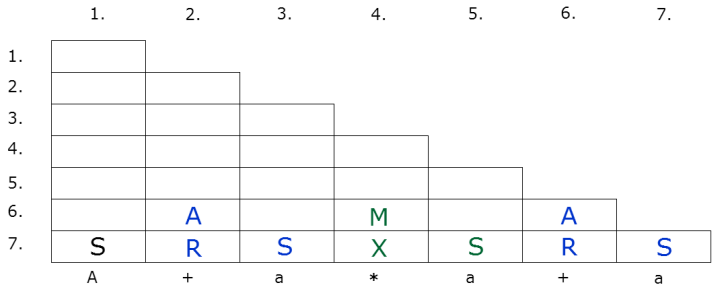
$$\{(S \to SA), (S \to SM), (S \to KP)(A \to RS), (M \to XS),$$

$$(P \to SZ), (R \to +), (X \to *), (K \to (Z)), (S \to a)\}$$

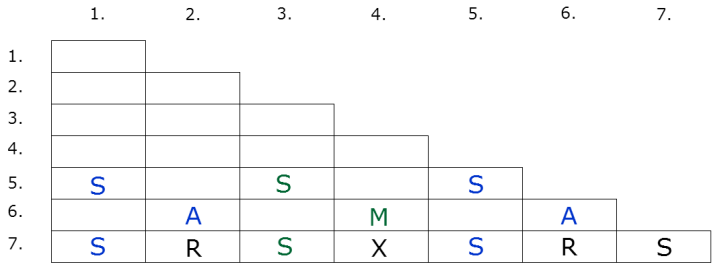
A fenti nyelvtan olyan numerikus kifejezéseket generál, minta az $a+a*(a+a)$. Legyen a példa generálandó szó az $a + a * a + a$. A háromszögmátrixot a legalsó sorának kitöltésével kezdjük. A k. cellába kerül be az a nemterminális jel, amelyből a generálandó szó k. terminális jele előállítható (1 hosszú rész-szavak).



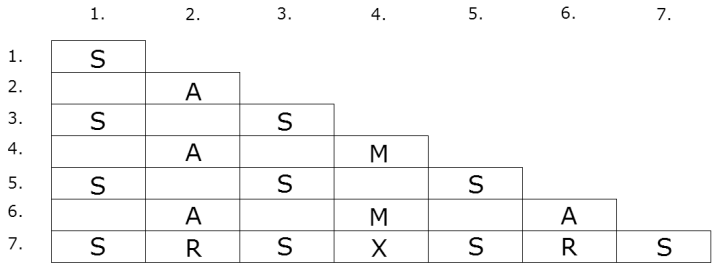
A következő lépésben a mátrix 6. sora kerül kitöltésre: a cellákba bekerül az a nemterminális jel, melyből kiindulva generálható a cél-szó adott pozíciójától kiinduló 2 hosszú rész-szava. A mátrix [6,2] cellájába azért került be az A szimbólum, mert lehetséges a cél szó 2. karakterétől kiinduló 2 hosszú rész-szó (+a) generálása az $A \to RS$, az $R \to +$, és az $S \to a$ szabályok alapján.



A következő lépésben az 5. sorba olyan nemterminálisok kerülnek be, amelyekből kiindulva 3 hosszú rész-szavak generálhatóak:



Tovább folytatva az eljárást az alábbi mátrixot kapjuk:



Az [1,1] cellába lévő S szimbólum azt mutatja, hogy belőle kiindulva előállítható a szó azon rész-szava, amely az első karaktertől kezdődik, és 7-1+1 hosszú. Ez gyakorlatilag megegyezik a teljes rész-szóval, így a kívánt szó előállítható az S szimbólumból kiindulva.

6.6. Szintaxisdiagram

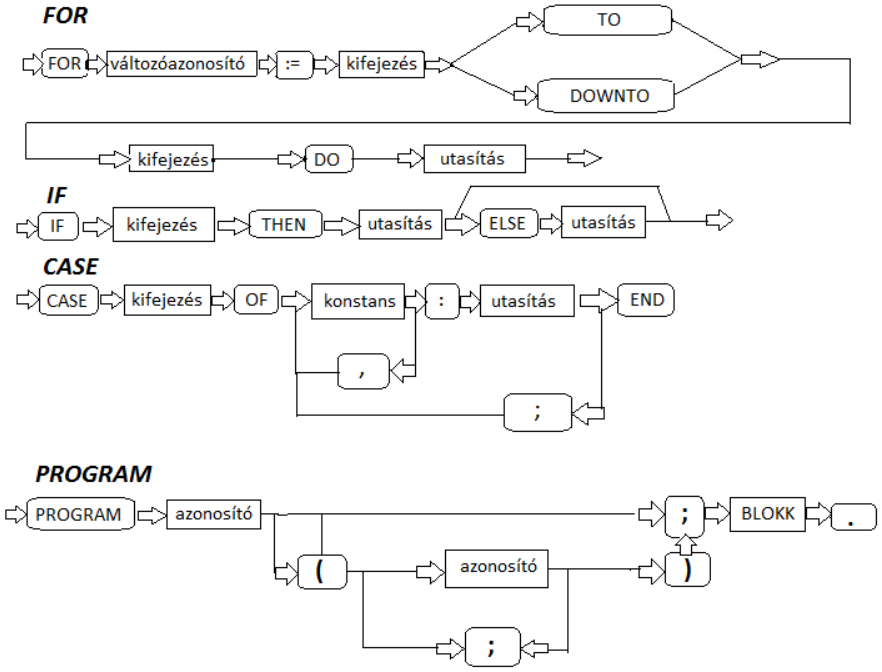
Az elemző algoritmusok ismertetését követően nézzünk meg néhány olyan szintaxis leíró eszközt is, amely segítségünkre lehet a fenti algoritmusokkal elemezhető nyelvek szintaxisának megtervezésére és leírására.

A szintaxisdiagram nem precíz matematikai eszköz, de látványos és gyors megértését teszi lehetővé a szintaxisfának. Jellemző felhasználási területe a programozási nyelvek szintaxisának leírása.

A gömbölyű sarkú téglalapjaiba a terminális elemek kerülnek, a szögletes sarkú téglalapokba pedig a nemterminális jelek.

A teljes szintaxisleírás esetén természetesen a nemterminális elemek felépítését is meg kell adni egy későbbi szabályban mindaddig lebontva a leírást, amíg eljutunk a legegyszerűbb elemekig, amelyek lerajzolásához már csak terminális elemeket használunk fel.

Ez pontosan úgy működik, mint a nyelvek szintaxisának megadása BNF, vagy EBNF formában, csak itt grafikusan ábrázoljuk a szintaxist.



6.7. EBNF - Extended Bachus-Noir forma

Az EBNF forma is szinatxisleíró eszköz, mely az előzőekben ismertetett szintaxisdiagramtól eltérően nem grafikus, hanem karakteres jelölésrendszert használ. Gyakran használjuk programozási nyelvek szintaxisának leírására, de alkalmas parancssori (command-line) operációs rendszerbeli parancsok definiálására is.

Az alábbi listában láthatjuk azokat azon komponensek leírását, amelyekkel az EBNF felírható (ez egy lehetséges EBNF megvalósítás, létezhetnek ettől eltérőek is).

- ::== szabály definiálása, Pl.: *program* ::= *elejekozepevege*,
- . (pont) a adott szabály vége,
- ... tetszőleges számban ismétlődő rész leírása (iteráció)
- [...] egyszer vagy nullaszor ismétlődő rész (opcionális elemek leírására),
- (...) egységbe foglalás, vagy csoportképzés
- | alternatív választó szimbólum (vagy)

- ... terminális jelsorozat

A fenti jelölésrendszer alapján példaként kialakítható az alábbi szintaxis leírás:

```
AdditívMűveletiJel ::== "+" | "-" | "or".
EgyszerűKifejezés  ::== [ Előjel ] Tag { AdditívMűveletiJel Tag }.
Kifejezés          ::== EgyszerűKifejezés
                                [ RelációsJel EgyszerűKifejezés ]
ÉrtékadóUtasítás   ::== ( Változó | Függvényazonosító )
                                ":=" Kifejezés.
CaseUtasítás       ::== "Ágazás" EsetIndex Eset { ";" Eset }
                       [ ; ] "Elágazásvége".
IfUtasítás         ::== "Ha" LogikaiKifejezés "Akkor" Utasítás
                       [ "Különben" Utasítás ].
ForUtasítás        ::== "Ciklus" Ciklusváltozó ":="
                                Kezdőérték "Ismétel" Végérték
                                "Ismétel" Utasítás.
FeltételesUtasítás ::== IfUtasítás | CaseUtasítás.
```

7. fejezet

Automaták

7.1. Automaták

Ahogy az eddigi fejezetekből ez kiderül, a generatív grammatikákat, vagy másképpen a produktív nyelvtanokat felhasználhatjuk arra, hogy egy adott szót, vagy mondatot generáljunk, de arra is, hogy megállapítsuk ugyanezekről a szavakról, vagy mondatokról, hogy azok elemei egy adott nyelvnek.

Ez a folyamat viszont, legalábbis kézzel elvégezve nagyon lassú, és könnyedén elrontható. A szabályok megkeresése, és alkalmazása, valamint a behelyettesítések elvégzése még rövid nyelvtani szabályrendszer esetén is körülményes.

Mindezért szükségünk van arra, hogy a nyelvtani elemzésekhez programokat, vagy legalább algoritmusokat konstruáljunk.

Ezeket az algoritmusokat automatáknak, vagy állapot átmenet gépeknek is tekinthetjük. Minden nyelvtan típushoz, amely része a Chomsky osztályoknak találunk hozzá tartozó automata osztályt, amely az adott nyelvcsaládot felismeri.

Az automata osztály, és a felismerés definícióját természetesen később megadjuk.

Összefoglalva, az eddigiekben azzal foglalkoztunk, hogy hogyan kell helyes szavakat, mondatokat generálni, vagy éppen eldönteni, hogy azok elemei-e az adott nyelvnek.

Ebben a fejezetben azzal foglalkozunk, hogy az elemzési műveleteket hogyan lehet automatizálni, és azzal is, hogy melyik nyelvtan típushoz milyen automatát kell konstruálnunk.

Az alábbiakban részletezett konstrukciók (automaták) tehát a programozási nyelvek szintaktikai elemzőinek is tekinthetőek. Összesen 4 fajta automatát vizsgálunk meg a 4 Chomsky osztálynak megfelelően.

Minden bonyolultabb (megengedőbb) Chomsky osztályhoz egyre bonyolultabb automata tartozik majd. Amennyiben a nyelvtanunk egy programozási nyelvet ír le, úgy azt az automatát (szintaktikai elemző algoritmust) kell használni az adott programozási nyelv fordítójának, amely Chomsky osztályba az adott programozási nyelvünk a generáló szabályrendszere alapján tartozik.

Logikus észrevétel, hogy amennyiben a nyelvünk szabályrendszere egyszerűbb (amely természetesen a programozónak is kedvező), úgy a fordítóprogramunk is egyszerűbb lesz.

Az alábbiakban megoldó algoritmusokról, és az algoritmusok működéséhez szükséges típusokról, segédváltozókról lesz szó.

Egyes nyelvek esetén a felismerés egyszerű lépéssorozat, mely véges számú lépésben garantáltan véget ér, és biztosan választ ad arra a kérdésre, hogy egy vizsgált szimbólumsorozat eleme-e az adott nyelvnek.

Bonyolultabb nyelvek esetén a megoldás nem garantált. Nullás típusú nyelvek esetén nem is létezik általános algoritmus, csak módszer (amely nem garantáltan vezet eredményre véges sok lépésben). Többek között ezért is kell törekedni a minél egyszerűbb nyelvtanok keresésére.

7.2. Automata általános megadása

Az automata egy olyan állapotátmenet gép, amely egy input szóról el tudja dönteni, hogy az megfelel-e az automatában implementált nyelvtani szabályoknak.

Egy általános automata az alábbi komponenseket tartalmazza (vagyis ezekből áll elő):

1. Van **input szalagja**, amely cellákra van osztva, és minden cellában egy-egy szimbólum található. Az input szalag általában véges, és épp olyan hosszú, amilyen hosszú az ellenőrzendő szó.

2. Az input szalaghoz tartozik egy un.: **olvasó fej**, amely mindig egy cella fölött áll, és ezen aktuális cellából képes kiolvasni a szimbólumokat. Az olvasó fej lépked az input szalag cellái között egyesével balra, vagy jobbra.

3. Létezhet egy **output szalagja**, amely cellákra van osztva, és minden cellában szintén egy-egy szimbólum állhat. Az output szalag lehet véges, vagy végtelen. Azokban a cellákban, amelyekbe még nem írt semmit az automata, egy speciális szimbólum, az un.: BLANK jel kerül.

4. Az output szalaghoz egy író-olvasó fej tartozhat. Ennek segítségével az automata egy jelet írhat az aktuális cellába, vagy olvashat onnan. Az író-olvasó fej szintén léptethető a cellák fölött balra és jobbra.

5. Az automatának vannak **belső állapotai**, melyek között az automata váltogathat egy megadott séma alapján. Az automata egyszerre mindig pontosan egy állapotban lehet (aktuális állapot).

Az automatákat, amellett, hogy állapot átmenet gépek, felfoghatjuk programként, vagy függvényként is. Az automata input szalagja egy karaktereket tartalmazó tömb (string), amely éppen olyan hosszú, amennyi karaktert elhelyeztünk rajta. Az író olvasó fej valójában a szövegtípus szelektor függvénye, és egy index, amely alapján az ahhoz az indexhez tartozó elemet (szimbólum, vagy karakter) ki lehet olvasni.

Az input szalagon az olvasófej jobbra-balra léptetését a szövegtípushoz rendelt index értékének megváltoztatásával oldhatjuk meg.

Az állapotok hasonlóak a típusról szóló fejezetben bemutatott adattípus, és az állapotinvariáns fogalmához. A gyakorlati megvalósítása az automata állapotainak az, hogy egy változóhoz rendelünk különböző értékeket egy halmazból, úgy mint a $\{q_0, 0_1, \ldots, q_n\}$, és minden lépésben értékül adjuk a halmaz valamely elemét egy szintén szövegtípusú változónak, pl.: $Allapot = q_0$.

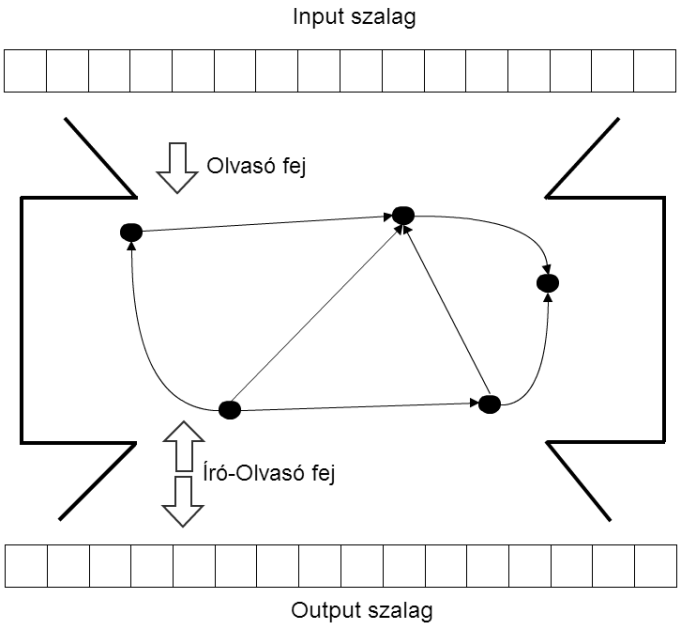
Az output szalag egyszerűbb automatáknál nem is szükséges, de ennek megvalósítása szintén történhet szövegtípusú változók írása-olvasása révén.

Az input szalag végigolvasása pedig egy egyszerű ismétléssel megoldható, amely a 0 indextől halad az input szalag hosszáig, és minden lépésben olvas egy elemet a szalagról.

```
egész I = 0
ciklus amig I < HOSSZ(inputszalag) simétel
  I = I + 1
  // állapotátmenetek
  Allapot = delta(Allapot, inputszalag[I])
ciklus vége
```

Ez a program gyakorlatilag lefedi egy egyszerűbb véges automata működését. A ciklus végigjárja az inputszalagot, és minden lépésben új állapotot rendel az aktuális állapothoz úgy, hogy alkalmazza rá az automata állapotátmenet függvényét (erről később még szót ejtünk).

A ciklus lefutás után már csak annyi a dolgunk, hogy megvizsgáljuk az automata megálláskor aktuális befejező állapotát, és a tartalma alapján eldöntjük, hogy az elemzett szó helyes, vagy sem...



7.3. Véges automaták

A véges automaták a legegyszerűbb automata osztályt alkotják. Nincs output szalagjuk, sem író-olvasó fejük. Az input szalag hossza véges, és éppen olyan hosszú, mint az input szó (ugyanannyi cellából áll, amennyi a szó hossza).

36. Definíció (Véges automata). *Egy G(K, V, δ, q_0, F) formális ötöst véges automatának nevezzünk, ahol:*

- *K: az automata belső állapotainak véges halmaza,*
- *V: az input ABC, vagyis az input szalagra írható szimbólumok, amelyek az automatában implementált nyelvtan terminális jelei,*
- *δ : állapotátmeneti függvény,$\delta \subseteq KxV \rightarrow K$*
- *$q_0 \in K$: speciális belső állapot, a kezdőállapot*
- *$F \subseteq K$: befejező, (elfogadó) állapotok halmaza*

Az automata attribútumai közül az első, vagyis a K tartalmazza azokat a belső állapotokat, amelyek között az automata váltogathat. Az állapot átmenetekért a delta függvény, amelynek az egyik paramétere az aktuális állapot, a másik az input szalagról éppen olvasott jel.

A terminális jelek alkotják az elemzendő mondatokat, és ezek kerülnek az input szalagra. Ez az automata osztály balról jobbra végigjárja az input szalagot, és minden lépésben meghívja a delta függvényt az éppen soron következő szimbólummal, és az aktuális állapottal paraméterezve.

Ezt a lépéssorozatot mindaddig ismétli, amíg az input szalag minden szimbólumát át nem adta a delta függvénynek, vagyis, amíg végig nem olvasta az input szimbólumok sorozatát.

A definíció, és a hozzá tartozó magyarázat alapján az automata működése az alábbi lépésekre osztható fel:

1. az automata q_0 kezd. állapotban van,
2. az input szalagon az input szó szimbólumai helyezkednek el, balról-jobbra, folytonosan.
3. az olvasó fej az input szalag legbaloldalibb cellája fölött áll,
4. az olvasó fej beolvassa az aktuális jelet az input szalagról
5. ezen jel, és az aktuális belső állapot ismereteben, a δ függvényben megfogalmazottak szerint újabb belső állapotba vált
6. az olvasó fejet lépteti egyel jobbra
7. az előző négy lépést ismétli, amíg az input szalag végére nem ér
8. az automata megáll, ha az olvasó fej lelép a szalagról (jobboldalon)

Amennyiben az automata megáll, meg kell vizsgálni, hogy milyen az aktuális állapota. Amennyiben az F-beli (elfogadó) állapot, akkor az automata a szót elfogadja. A megállás és elfogadás-t meg később pontosítjuk.

Mivel az automata minden lépésben olvas egy szimbólumot az input szalagról, és mindig jobbra lep, ezert az automata biztosan megáll n lépés végrehajtása után (az n azonos a szalag hosszával).

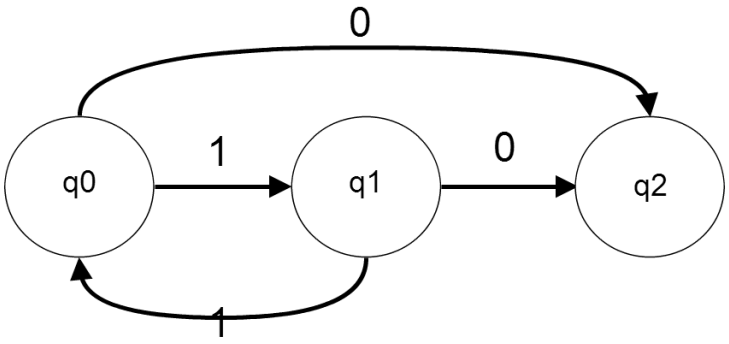
A δ függvény függvény egy aktuális állapot ($k \in K$) és egy input jel ($v \in V$) alapján megadja, hogy milyen új állapotba ($k' \in K$) lepjen, vagyis $\delta \subseteq KxV \rightarrow K$.

A δ függvény megadható táblázattal, vagy gráffal is. A gráf alapú megadás szemléletesebb, de a táblázatos forma sokkal inkább alkalmas az implementációs lépések elvégzéséhez.

a. táblázattal

δ	K	V	K
	q0	0	q2
	q0	1	q1
	q1	0	q2

b.gráffal



A definíciókat követően vizsgáljunk meg egy olyan példát, amely teljes és determinisztikus, ezután már megadhatjuk a teljesség, és a végesség definícióját is. Legyen adva P := S$\rightarrow 1A, \rightarrow 1A, \rightarrow 1A$ a grammatika szabályrendszere, amely páratlan számú 1-esekből álló szavakat generál. A feladat az, hogy olyan automatát konstruáljunk, amely a fenti nyelvtan szabályainak megfelelő szavakat elfogadja, a többit szót elutasítja.

Az automata belső állapotainak a halmaza: $K := \{q_0, q_1\}$, az input ábécé egy elemű: $V := 1$, és az elfogadó állapotok halmaza is csak egy állapotot tartalmaz: $F := \{q_1\}$. Az automata delta táblázata a következő:

δ	K	V	K
	q0	1	q1
	q1	1	q0

Elemezzük meg az automatát működés közben. Legyen az input szó az 111. Az automata q_0 kezdőállapotból indul. Beolvasásra kerül az első 1-es, és ekkor az automata átlép a q_1 állapotba, valamint a fej jobbra lép egyet.

Beolvasásra kerül a második 1-es. Az automata átvált a q_0 állapotra, és a fej jobbra lép.

Beolvassuk a a harmadik 1-est. Az automata átlép q_1 állapotba, és a fej egyet jobbra lép. Mivel a fej lelépett a szalagról, az automata megáll. Ekkor az automata q_1 állapotban van, és $q_1 \in F$, amiért az automata elfogadja a szót.

Az 1111 input szóra levezetve ugyanezt, az automata q_0 állapotban fejezné be a működését, ami nem elfogadó állapot. Ebben az esetben az automata a szót elutasítja.

7.4. Parciális delta leképezés

Legyen adott $P := \{S \to 1A, A \to 1B, A \to 1, B \to 1B, B \to 0B, B \to 0, B \to 1\}$ a nyelvtan szabályrendszere, amely szabályrendszer segítségével legalább két 1-essel kezdődő szavakat tudunk generálni. A feladat olyan automatát konstruálni, amely a generálás szabályainak megfelelő szimbólumsorozatokat elfogadja, a többit elutasítja.

Az automata belső állapotai a következők: $K := \{q_0, q_1, q_2\}$, A terminális jelek a szalagon: $V := \{0, 1\}$, és az elfogadó állapotok egyelemű halmaza: $F := \{q_2\}$. Az automata delta függvénye táblázattal megadva a következő:

δ	K	V	K
	q0	1	q1
	q1	1	q2
	q2	1	q2
	q2	0	q2

Ha megvizsgáljuk az automatát működés közben, láthatjuk, hogy az első két egyes beolvasása során az q_0-ból q_1-be, majd q_1-ből q_2-be lép. Ezután akár nullát, akár egyest olvas a szalagról, mindenképpen q_2 állapotban marad, ami egyben az elfogadó állapot is.

Nézzük meg azt az esetet, mikor a szalagon levő szó 0-val kezdődik. Ekkor a q_0 állapotban 0-t olvasva az automata nem tud mit lépni, mert a δ függvénye erre a lehetőségre nincs felkészítve. Ekkor az automata rögtön megáll.

Amennyiben a szó közepén áll meg a fenti oknál fogva, úgy az automata nem fogadta el a szót. A fentihez hasonló esetekben, amikor a δ nem minden eshetőségre ad egyértelmű lépést, a δ leképezést parciálisnak (részlegesnek) nevezzük. Ellenkező esetben δ függvény teljesnek.

7.5. Determinisztikus és nem determinisztikus működés

Legyen adott a $P := \{S \to 0S, S \to 1S, S \to 1A, A \to 1\}$ szabályrendszer, amely legalább két 1-esre végződő szavak generálására alkalmas. A feladat, hogy olyan automatát konstruáljunk, amely a generálás szabályainak megfelelő szavakat elfogadja, a többit elutasítja.

Az automata belső állapotai a következők: $K := \{q_0, q_1, q_2\}$, az input ábécé halmaza két elemű, és az elfogadó állapotok halmaza a $V := \{0, 1\}, F := \{q_2\}$. A szabályrendszer alapján az automata delta táblázata az alábbi módon írható fel:

δ	K	V	K
	q0	0	q0
	q0	1	q0
	q0	1	q1
	q1	1	q2

Ha megvizsgáljuk az automatát működés közben, láthatjuk, hogy q_0 állapotban 1-est olvasva nem egyértelmű, hogy mi a következő lépés.

A δ leképezés alapján ez a q_0, és a q_1 is lehetne. Az egymást követő elemzési lépések során lépések során előfordulhat, hogy 1-es szerepel a az input szóban. Ez nem biztos, hogy a szó végét jelzi. Ha igen, akkor q_0 állapotban kell maradnia az automatának. Az utolsó előtti 1-est olvasva kell

átlépni q_1-be, majd onnan az újabb 1-est olvasva q_2-be. Mivel ekkor elfpgy a szó, és q_2 egyben elfogadó állapot is, az automata felismeri a szót.

Amennyiben a δ nem egyértelmű, mert egy adott állapothoz és input jelhez két (vagy több) különböző új belső állapotot is rendelhetünk, úgy azt mondjuk, hogy a δ leképezés nem determinisztikus. Ellenkező esetben a δ determinisztikus.

Nem determinisztikus esetben az automata maga dönti el (véletlenszerűen), hogy melyik lehetőséget választja a továbblépéshez. Ez azt is jelenti, hogy kétszer megadva ugyanazt a szót, az automata nem ugyanazt a lépéssorozatot választja, és egyik esetben elfogadja, a másikban elutasítja ugyanazt a szót. Példaként az 110011 input szóra az automata a q_0, q_0, q_0, q_0, q_1, q_2 lépéssorozatot is választhatja, és elfogadja a szót, de maradhat végig q_0 állapotban és így nem fogadja el a szót, de q_0, q_1, q_2 sorozatba is kezdhet, majd parciális okoknál fogva leáll és nem fogadja el a szót.

Mivel azonban a szó jó, megfelel a nyelv szabályainak, így az automata el kellene hogy fogadja azt. Hogy a működés se változzon, az elfogadás is eredményes lehessen, ezért a nem determinisztikus δ függvény esetén azt mondjuk, hogy az automata akkor fogadja el a szót, ha létezik olyan eset (sorozat), amelyben elfogadja.

7.6. Automaták ekvivalenciája

37. Definíció (Elfogadás). *Egy $A(K, V, \delta, q_0, F)$ véges automata egy L nyelvet felismer, ha minden $\alpha \in L$ szóra az automata megáll, és a szót felismeri, és minden $\beta \in L$ szóra az automata a szót elutasítja.*

38. Definíció (A felismert szó). *Egy $A(K, V, \delta, q_0, F)$ véges automata által felismert szavakból alkotott L nyelvet az automata által felismert nyelvnek nevezzük.*

39. Definíció (Automaták ekvivalenciája). *Egy $A(K, V, \delta, q_0, F)$ véges automata egy $A'(K', V, \delta', q_0', F')$ automatával ekvivalens, ha a két automata ugyanazt a nyelvet ismeri fel.*

40. Definíció (Automaták ekvivalenciája). *Egy $A(K, V, \delta, q_0, F)$ parciális véges automatához mindig konstruálható olyan $A'(K', V, \delta', q_0', F')$ teljes és véges automata úgy, hogy a két automata ekvivalens.*

A konstrukció lényege, hogy az automatát ki kell egészíteni egy új állapottal, és minden parciális esetet úgy kell lekezelni, hogy az automata ebbe az új állapotba kerüljön. Ezután tetszőleges jel beolvasása esetén már ebben az állapotban kel tartani, és ez az állapot ne legyen elfogadó állapot.

41. Definíció. *Egy $A(K, V, \delta, q_0, F)$ nem determinisztikus véges automatához mindig konstruálható olyan $A'(K', V, \delta', q_0', F')$ determinisztikus és véges automata úgy, hogy a két automata ekvivalens.*

A fenti két tétel értelmében tehát minden automata visszavezethető teljes és determinisztikus működésre, ami igen jó tulajdonság, mivel az automaták nagy részénél ugyanúgy, mint a programok és függvények esetén a parciális, valamint a nem determinisztikus működés nem szerencsés.

A jegyzetben található automata osztályok esetében egyedül a Turing gép az, amely delta függvénye parciális kell legyen, az összes többinél ezt a tulajdonságot érdemes kiküszöbölni, és a fenti néhány definíció értelmében ezt meg is tudjuk tenni.

7.7. Automata konfigurációja

42. Definíció (Automata konfigurációja). *Egy $A(K, V, \alpha, q_0, F)$ véges automata konfigurációja egy (α, q) formális kettes, ahol α az input szalagon meg hatra levő szó, q pedig az aktuális belső állapot.*

Az automata konfigurációjára tekinthetünk úgy is, hogy amennyiben kikapcsoljuk az automatát működés közben, majd visszakapcsoljuk, és vissza akarjuk állítani a kikapcsoláskori állapotot, akkor az input szalagra vissza kell írni a meg hatra levő szót, majd vissza kell állítani a kikapcsoláskori belső állapotot.

Érthető, hogy az input szóból mar beolvasott rész ezen szempontból érdektelen, hiszen az olvasó fej csak jobbra mozoghat, így a mar beolvasott részről többe nem olvas az automata.

Az indító konfiguráció (ω, q_0), ahol ω a teljes input szó. Az automata feldolgozás közbeni köztes konfigurációja (α, q_i) állapotsorozat. A záró konfigurációja (ϵ, q'). Egy záró konfiguráció elfogadó, ha $q' \in F$, vagyis a záró konfiguráció állapota eleme az elfogadó állapotok halmazának.

Az automata a működése gyakorlatilag nem más, mint konfigurációk sorozata. A kezdő konfigurációra alkalmazva a δ függvényt megkapjuk a következő konfigurációt, ezt a lépést ismételve kapjuk a konfigurációsorozatot.

Nézzünk erre is egy példát: Legyen a véges automatánk belső állapotainak halmaza $K := \{q_0, q_1, q_2\}$, az input jelek halmaza $V := \{0, 1\}$, és az elfogadó állapotok egyelemű halmaza $F := \{q_2\}$. Ekkor az elemző automata delta függvénye táblázatos formában a következő:

δ	K	V	K
	q0	0	q0
	q0	1	q0
	q0	1	q1
	q1	1	q2

A felismerendő input szó legyen a 1011. Ekkor egy lehetséges konfigurációsorozat: $(1011, q_0) \to (011, q_0 \to (11, q_0) \to (1, q_1) \to (\varepsilon, q_0)$. Mivel a záró konfigurációban a hátralévő input szó ε és $q_2 \in F$, így ez esetben az input szót az automata elfogadta.

Előállhat a fentiek alapján egy olyan konfiguráció sorozat is, amely a $(0, q_0)$ párosra nincs a delta függvény értelmezve. Ekkor az automata a feldolgozás közben megáll. Mivel nem csak az ε van hátra az input szóból, így érdektelen, hogy a megállási állapot eleme-e az F halmaznak! Az automata ebben a menetben az input szót elutasítja. Ugyanakkor mivel az automata nem determinisztikus, így ebből az egy esetből nem szabad messzemenő következtetést levonni!

Legyen most a véges automatánk belső állapotainak a halmaza $K := \{q_0, q_1, q_2\}$, az input ábécé a $V := \{0, 1\}$, és az elfogadó állapotok halmaza az $F := \{q_2\}$. Az automata delta táblázata ekkor a következő:

δ	K	V	K
	q0	0	q0
	q0	1	q0
	q0	1	q1
	q1	1	q2
	q2	0	q0

Legyen az elemzendő szavunk a 10110. Ekkor egy lehetséges konfiguráció sorozat a $(10110, q_0),) \to (0110, q_0) \to (110, q_0) \to (10, q_1) \to (0, q_2) \to (\varepsilon, q_0)$.

Vegyük észre, hogy a feldolgozás közben az automata felvette a q_2 állapotot, amely elfogadó állapot! De ez nem jelenti azt, hogy az automatának

meg is kell állnia!

Az automata kizárólag akkor áll meg, amikor a δ képtelen folytatni a feldolgozást (mikor nincs elfogytak az input jelek, vagy input jel még van, de az adott input jel és belső állapot párosra nincs megfelelő lépés).

Az automata ekkor az input szót elutasítja, mert bár a feldolgozás végig tudott menni az input szalagon, de a végén nem elfogadó állapotban állt meg!

Ez a konfiguráció sorozat pontosan n elemű, ha az automata sikeresen végigolvasta az n hosszú input szót. Ekkor a sorozat biztosan megszakad (az automata megáll), hiszen nem lehet következő elemet meghatározni.

És a lépéssorozat kevesebb, mint n elemű, ha az utolsó konfigurációra nem tudjuk alkalmazni a δ függvényt. Ekkor a δ függvény parciális, hiszen egy teljes. függvény minden esetben alkalmazható, amíg van input jel az input szalagon.

A lépéssorozat ugyanarra az input szóra más-más elemeket tartalmazhat, ha a δ függvény nem determinisztikus. Ekkor ugyanis létezhet olyan lépés, amelyre a δ függvény több következő konfigurációt is képes megadni, így többször alkalmazva a δ függvényt legalább ezen a ponton (és ettől a ponttól kezdve) eltérő lépéseket kaphatunk.

A generálás közben több ilyen pont is lehetséges, így nehézséget okozhat, hogy feltérképezzük az összes lehetséges konfiguráció-sorozatot, amely az adott szó esetén előfordulhat.

Ugyanakkor a nem determinisztikus automata az input szót elfogadja, ha a lehetséges lépéssorozatok közül van olyan, amelynek az utolsó eleme elfogadó konfiguráció.

43. Definíció (Elfogadás definíciója véges automatánál). *Egy $A(K, V, \delta, q_0, F)$ véges automata egy ω input szót elfogad, ha létezik olyan konfiguráció sorozat, amely során a δ leképezés véges sokszori alkalmazása során az automata induló konfigurációja (δ, q_0) átvihető az (ε, q') záró konfigurációba, és $q' \in F$. Ellenkező esetben az automata az input szót elutasítja.*

A fenti definíció egyformán jó a determinisztikus, és a nem determinisztikus, valamint a teljes és parciális esetekre is.

A parciális esetben nem létezik olyan sorozat, amelynek során az input szó ε-ra csökkenhetne, hiszen ekkor az automata menet közben meg fog állni, és a szalagon még lenni kell szó-résznek.

Nem determinisztikus működés esetén akkor helyes a szó, ha van olyan konfiguráció sorozat, amely esetén az automata bejárja a szót, és elfogadó állapotban áll meg.

7.8. A véges automata működésének elemzése

Az automata működését az alábbi felsorolásban szereplő pontok elemzésével jobban megérthetjük:

- Egy n hosszú input szó esetén a véges automata legfeljebb **n lépés** végrehajtása után biztosan megáll (korábban is megállhat parciális működés esetén).
- Az automata minden esetben biztosan megáll. Ennek oka, hogy minden lépésben olvas be jelet a szalagról, és mindig lépteti a fejet, és mindig jobbra. Ezért legfeljebb n lépés múlva a fej biztosan lelép a szalagról.
- Ha egy helyes szót táplálunk az automatába, az automata pontosan n lépés után megáll, és elfogadó állapotba kerül (ha az automata jól választja meg a lépéssorozatot).
- Az automata pontosan n lépés után megáll, de nem elfogadó állapotban van (ha az automata nem determinisztikus, és rossz útvonalat választott.
- Vagy az automata kevesebb, mint n lépésen belül megáll (ha az automata nem determinisztikus, és parciális, és rossz útvonalat választott).
- Ha egy helytelen szót elemzünk az automatával, az pontosan n lépés után megáll, de nem elfogadó állapotban van (nem determinisztikus és determinisztikus esetben is).
- Vagy az automata kevesebb, mint n lépésen belül megáll (parciális működés).
- Az automata azért áll meg, mert nem tudja folytatni a feldolgozást a szalagról lelépve. Ugyanis a delta függvény igényli minden lépésben egy jel beolvasását az input szalagról, mely a szalagról lelépés után nem kivitelezhető.

7.9. Minimál automata

Mivel egyetlen nyelvhez több felismerő automata is tartozhat, próbáljuk meg megkeresni ezek közül a legjobbat. Ezen automata azért lesz legjobb, mert állapotainak száma a legkisebb mind közül. Ezen automatát minimál automatának nevezzük.

A minimál automata megszerkesztéséhez indulásképpen szükségünk van egy determinisztikus és teljes automatára. Ennek ismeretében módszer ismert a minimál automata előállítására (a már létező automatánk állapotainak számának minimalizálására).

Bizonyítható (a bizonyítást nem közöljük), hogy minden determinisztikus véges automata esetén ezen módszer lépéssorozata eredményre vezet, és a minimál automata a jelölésrendszerétől eltekintve egyértelműen létezik.

7.10. Véges automata leírás és feldolgozó algoritmus

A véges automata programja gyakorlatilag a fenti definíciók alapján konstruált algoritmus.

```
AktAllapot  := q0
FejIndex  := 0
NemDetMukodesVolt := HAMIS
CIKLUS AMÍG FejIndex<SzalagHossz
  Jel := Szalag[ FejIndex ]
  FvOutput := DeltaTablazat( Jel, AktAllapot )
  N := Darabszama(FvOutput )
  HA N=0 AKKOR
    HA NemDetMukodesVolt AKKOR
Kiiras:"Nem fogadta el (nem det.)!"
Kiiras:"Több eredmény is lehet!"
    KULONBEN
Kiiras: "Az input szó helytelen (parc. műk.)!"
    HVEGE
  HVEGE
  Valasztas := 0
  HA N>1 AKKOR
    Valasztas := veletlenszam( 0 .. N-1 )
    NemDetMukodesVolt := IGAZ
  HVEGE
  AktAllapot := UjAllapotok[ Valasztas ].UjBelsoAllapot
  FejIndex++
CVÉGE
HA AktAllapot eleme ElfogadoAllapotok AKKOR
  Kiiras: "Az automata az input szót elfogadta"
KULONBEN
```

```
    HA NemDetMukodesVolt AKKOR
      Kiiras:"Nem fogadta el (nemdet. műk.)"
      Kiiras:"Több eredmény lehet"
    KULONBEN
      Kiiras: "Az input szó biztosan helytelen!"
    HVEGE
  HVEGE
```

A fentiekből látszik, hogy az AktAllapot változót olyanra kell deklarálni, amelyben tárolhatóak lesznek az aktuális állapotok. Ez megoldható, ha az állapotokat besorszámozzuk, ekkor a változó lehet például egész típusú.

Hogy az egészek milyen altípusa szükséges, az eldönthető a K halmaz számosságából, tekintve hogy ekkor sorszámok várhatóak az algoritmus futása közben.

A DeltaTablazat() függvény kétparaméteres, ahol az első paraméter típusát a szalagon felfedezhető jelek típusa (V) határozza meg. Ha ez leírható ASCII karakterekkel, akkor a típus char. Ha azonban a V halmaz számossága ennél nagyobb, akkor szintén megoldás lehet a V halmaz elemeinek sorszámozása, és valamilyen egész típus használata.

A NemDetMukodesVolt logikai változóra azért van szükség, mert ha a szó feldolgozása közben egyszer is választás elé lett állítva az automata, akkor ennek sajátossága miatt a válasz csak egyszeri működésre értelmezett.

A végső válasz előállításához igazából a visszalépéses keresést kellene használni, és a feldolgozás végén visszatérni minden választáshoz, és kipróbálni a további választási útvonalakat is. E módosított algoritmus ezen változtatás miatt jóval bonyolultabb felírású lenne, de az automata válasza végleges lenne, vagy ELFOGADTAM vagy NEM_ELFOGADTAM típusú lenne.

7.11. Baar-Hiller lemma

44. Definíció. *Elgyen adott egy L nyelv, és egy $A(K, V, \delta, q_0, F)$ véges automata, amely az L nyelvet felismeri. Ekkor, ha létezik egy n pozitív egész szám, és egy olyan $\alpha \in L$ szó, hogy $L(\alpha) \geq n$, akkor igazak az alábbiak:*

- *az α szónak létezik olyan $\alpha = \beta\omega\gamma$ felbontása hogy $L(\beta\omega\gamma) \leq n$, és $L(\omega) \geq 1$),*
- *$\forall i \geq 0$ esetén a $\beta\omega^i\gamma \in L$.*

Legyen egy véges automata belső állapotainak száma n. Amennyiben az input szalagra irjuk az α szót (amely a nyelvnek eleme, így az automata

elfogadja), akkor a felismerés közben az automatának legalább egy belső állapotot legalább kétszer kell felvennie.

Tegyük fel, hogy ezen állapot, amelyet az automata legalább kétszer felvesz, q'. Ekkor az automata a két azonos belső állapot között legalább egy szimbólumot beolvas.

Jelöljük az α szó azon részét, amelyet az automata addig olvas be, amíg először jut el a q' állapotába β-val, és azt a szakaszt, amíg q'-ből újra q'-be jut ω-val, valamint az α maradék részét γ-val.

Ha a szó $\beta\omega\omega\delta$ alakú, az automata a szó felismerése közben β szakasz beolvasása után q' állapotba kerülne.

Folytatva a beolvasást az első ω szakasz után újra q' állapotban, majd a második ω szakasz után újra q' állapotban kerülne.

Ebből a q'-ből a maradék γ beolvasása után kizárólag elfogadó állapotba kerülhet az automata, mivel tudjuk róla, hogy $\beta\omega\gamma$ esetén is oda került volna.

Látható, hogy a középső ω szakaszt akárhányszor egymás után fúzhetjük, az automata legfeljebb annyiszor kerül (minden ω szakasz végére) q' állapotba.

Az ω szakaszok száma tehát a szó elfogadását nem befolyásolja. Így a $\beta\gamma$, $\beta\omega\gamma$, $\beta\omega\omega\gamma$, $\beta\omega\omega\omega\gamma$, ... alakú szavak is elemei az L nyelvnek.

Amennyiben tehát van egy L nyelvünk, és ismert, hogy az őt felismerő véges automata például 4 belső állapottal rendelkezik, és mi találunk legalább egy olyan szót, amely 4-nél hosszabb, de eleme a nyelvnek, akkor végtelen sok szót találhatunk, így a nyelv végtelen.

Ha van egy L nyelvünk, és ismert, hogy az őt felismerő véges automatának példaként 6 belső állapota van, úgy a nyelv akkor nem üres, ha létezik 6-nál rövidebb szó, amelyet a nyelv elfogad.

A 6 szimbólumnál hosszabb szó létezése setén, létezne hatnál rövidebb szó is ($\beta\gamma$).

Mindezek alapján elmondhatjuk, hogy konstruálható algoritmus, mely el tudja dönteni, hogy van-e olyan szó, amelyet egy automata felismer. Az algoritmusban használt módszer lényege, hogy végig kell próbálgatni az összes lehetséges kombinációt az n-nél rövidebb szavakra.

Ha találunk ilyen szót, akkor a nyelv nem üres. Ha egyetlen n-nél rövidebb szót sem ismer fel az automata, akkor egyetlen szót sem fog!

A végigpróbálgatásnál természetesen figyelni kell, hogy a nem determinisztikus működés esetén az egyszeri próba sikertelensége nem jelenti azt, hogy egy újabb próba viszont nem lesz sikeres.

7.12. Számítási kapacitás

45. Definíció (Számítási kapacitás). *Azonos típusú automaták halmazát absztrakt géposztálynak nevezzük, és az absztrakt géposztály számítási kapacitása azon formális nyelvek halmaza, amelyet a géposztály valamely automatája felismer.*

A nem determinisztikus véges automataosztály és a determinisztikus véges automata osztály számítási kapacitása megegyezik.

Ez meglepőnek tűnhet, hiszen a nem determinisztikus működés a determinisztikus működés kiterjesztése, és úgy tűnik, hogy a nem determinisztikus automaták több lehetőséggel rendelkeznek. Ezek alapján következtethetnénk arra is hogy ezen automaták számítási kapacitása nagyobb.

A számítási kapacitásuk viszont egyenlő, mivel minden nem determinisztikus véges automatához konstruálható vele ekvivalens determinisztikus véges változat.

A véges automataosztály számítása kapacitása megegyezik a reguláris nyelvek osztályával.

Ennek bizonyítására vizsgáljuk meg a következő példát. Legyen adott egy véges automata, amely egy nyelvet felismer.

Vegyük a delta függvény definícióját, és q_0 kezdőállapot helyébe helyettesítsük be az S szimbólumot (nem terminális jelet).

A $q_1, q_2, \ldots, q_n$ helyében pedig $S_1, S_2, \ldots, S_n$ nemterminális jeleket. Ekkor pontosan annyi nemterminális jelünk lesz, ahány belső állapota van az automatának.

Ha a delta függvény táblázatának valamely sora $(q_n, a) \to q_m$ alakú, akkor ahhoz rendeljük hozzá az $S_n \to aS_m$ alakú helyettesítési szabályt, így a jobb reguláris nyelvtanoknak megfelelő szabályokat kapunk.

Bizonyítható (ezt most nem bizonyítjuk), hogy a fenti szabályok alkalmazásával generált szavakat az automata felismeri, de egyéb szavakat nem.

46. Definíció (Véges automaták számítási kapacitása). *Minden reguláris nyelvhez konstruálható olyan véges automata, amely az adott nyelvet felismeri, valamint, egy véges automata által felismert nyelv biztosan reguláris.*

7.13. Verem automaták

A verem automaták számos attribútumukban különböznek a véges automatáktól. Első, legszembetűnőbb különbség, hogy a verem automatának van egy speciális komponense, a verem. A veremhez tartozó író-olvasó fej nem mozoghat szabadon, helyette mindig a legjobboldali (legfelső) cellába írhat,

majd minden írás után jobbra mozdul (lefelé). Olvasáskor mindig csak a legjobboldali cellát képes kiolvasni, de ezt megelőzően balra mozog, ami a verem esetében a felfelé mozgást szimbolizálja.

Ez a veremkezelés sajátosságai miatt van így. A LIFO adatszerkezetek működése miatt mindig a verem tetejére írunk, és olvasni is csak onnan tudunk.

47. Definíció (Verem automata). *Egy* $G(K, V, W, \delta, q_0, z_0, F)$ *formális hetest verem automatának nevezünk, ahol:*

- K*: az automata belső állapotainak véges halmaza,*
- V*: az input ABC, az input szalagra írható jelekkel,*
- W*: a verembe írható, és onnan kiolvasható jelek halmaza, amelyre igaz, hogy az automatában alkalmazott nyelvi szabályok terminális, és nemterminális jeleit tartalmazza,*
- δ*: állapotátmeneti függvény,* $\delta \subseteq Kx(V \cup \{\varepsilon\})xW \to KxW*$,
- $q_0 \in K$*: speciális belső állapot, a kezdőállapot,*
- $z_0 \in W$*: speciális jel, a "verem üres" szimbólum,*
- $F \subseteq K$*: az elfogadó állapotok halmaza.*

A verem automaták fontos részei a fordítóprogramoknak. A veremautomata egy változata, az üres veremmel felismerő, táblázatos elemző a szintaktikai elemzés jól ismert algoritmusa, és a legtöbb programozási nyelv fordítóprogramja is ezen az elven működik.

Az automata működése a következő lépésekből áll. Induláskor az automata q_o kezdő állapotban van, és a veremben csak egyetlen jel van, a z_0.

Az input szalagon az input szó jelei vannak felírva, balról-jobbra folytonosan, és az olvasó fej az input szalag legbaloldalibb cellája fölött áll.

Működés közben az olvasó fej vagy olvas az input szalagról, és a veremből is kiolvassa a verem tetején levő jelet.

Táblázatos elemzők esetén ezt a két szimbólumot használja az elemző táblázatának celláiban lévő szabályok indexelésére úgy, hogy például a veremből olvasott elem lesz a sorindex, és a szalagon lévő szimbólum az oszlopindex. A két index által meghatározott cella tartalmazza a szabályt, amit alkalmazni kell.

(Az, hogy a táblázatot hogyan kell elkészíteni, és az automata hogyan alkalmazza a cellákban lévő szabályokat, túllépi a jegyzet kereteit, ezért ezeket nem közöljük.)

Általánosan az automata az olvasott szimbólumok alapján, és az aktuális belső állapot ismeretében, a δ függvényében megfogalmazottak szerint belső állapotot vált vált, majd a verembe ír egy szót (szimbólumsorozatot), majd az olvasó fejet léptetheti jobbra, de ezt nem minden esetben kell megtennie.

Az automata megáll, ha az olvasó fej jobbra lelép a szalagról. Ekkor meg kell vizsgálni, hogy milyen az aktuális belső állapota. Amennyiben az F-beli (elfogadó) állapot, akkor az automata az elemzett szót felismerte. A megállás és elfogadás-t még később pontosítjuk.

Mivel az automata nem minden ciklusléptésben olvas egy jelet az input szalagról, e miatt nem mindig lépteti az olvasó fejet, ezert nem biztos, hogy megáll n lépés végrehajtása után.

A veremautomaták esetén a delta függvény háromváltozós függvény. A paraméterei egy aktuális belső állapot állapot ($k_i \in K$), egy input jel ($v_i \in V$) vagy nem olvasás esetén ε, és egy szimbólum a veremből ($w_i \in W$).

A paraméterek alapján a függvény megadja, hogy milyen új állapotba kerüljön az automata, és mit írjon a verembe ($W*$).

A verem automata delta függvénye lehet parciális, vagy teljes, valamint lehet nem determinisztikus, illetve determinisztikus is. Az esetek kezelése lényegében megegyezik a véges automatáknál látottakkal.

48. Definíció (Felismerés definíciója). *Egy $G(K, V, W, \delta, q_0, z_0, F)$ verem automata egy L nyelvet felismer, ha minden $\alpha \in L$ szóra az automata megáll, és a szót felismeri, és minden $\beta \notin L$ szóra az automata a szót elutasítja.*

A véges automatákhoz hasonlóan egy verem automata által felismert szavakból alkotott L nyelvet az automata által felismert nyelvnek nevezzük.

Valamint két veremautomata ekvivalens, ha ugyanazt a nyelvet ismerik fel.

Egy $G(K, V, W, \delta, q_0, z_0, F)$ parciális verem automatához mindig konstruálható olyan $G'(K', V, W', \delta', q'_0, z'_0, F')$ teljes véges automata úgy, hogy a két automata ekvivalens.

A nem determinisztikus esettel nem ilyen egyértelmű a helyzet. Sajnos nem készíthető olyan általános algoritmus, amely tetszőleges nem determinisztikus veremautomata alapján elkészíti annak determinisztikus ekvivalens változatát.

Az alábbi delta függvény egy veremautomata nem determinisztikus delta függvénye A q_0 állapotában, amikor a veremben a z_0 van legfelül, és a szalagon 1-es a szimbólum, akkor az automatának van választási lehetősége:

1. vagy nem olvas a szalagról semmit (ε-t olvas),

2. vagy beolvashatja az 1-es szimbólumot,

mivel mindkét esetre létezik kimenete.

δ	K	V ∪ {ε}	W	K	W*
1	q0	ε	z0	q0	z0
2	q0	1	z0	q1	zxy

A példából láthatjuk, hogy a veremautomaták eseten a nem determinisztikus eset vizsgálatánál ügyelni kell az ε olvasási lehetőségekre is.

7.14. Veremautomata konfigurációja

49. Definíció (Veremautomata konfigurációja). *Egy $G(K, V, W, \delta, q_0, z_0, F)$ veremautomata konfigurációja egy (α, q, β) formális hármas, ahol α az input szalagon még hátra levő szó, q az aktuális belső állapot, β a veremben levő szó.*

Az automata induláskori konfigurációja (ω, q_0, z_0), ahol ω a teljes input szó. Az automata feldolgozás közbeni köztes konfigurációja valamely (α, q, β). A normál működés záró konfigurációja $(\varepsilon, q', \beta')$.

50. Definíció (Veremautomata elfogadás definíciója). *Egy véges automata egy ω input szót elfogad, ha létezik olyan lépéssorozat, amelynek során a δ leképezés véges sokszori alkalmazása révén az automata induló konfigurációja (ω, q_0, z_0) átvihető az $(\varepsilon, q', \beta')$ záró konfigurációba, és $q' \in F$. Ellenkező esetben az automata az input szót elutasítja.*

A fenti definíció megint megfelelő a δ minden működési módjára.

7.15. Delta leképezés elemzése

δ	K	V ∪ {ε}	W	K	W*
1	q0	ε	z0	q0	z0
2	q0	1	z	q1	zxy
3	q0	1	z	q0	ε
3	q1	1	z	q1	Z

1. Magyarázat: Az automata nem olvas az input szalagról, és olvasás előtt-után q_0 állapotba kerül, valamint a veremből a z_0 szimbólumot olvassa. Ezt a sort az automata önmagában végtelen sokszor ismételheti (végtelen ciklus).

2. Az automata egy jelet vesz ki a veremből (z), és három jelet tesz vissza (zxy). Ezen sor végrehajtása után a verem aktuális mérete 2-vel növekszik.

3. Az automata egy jelet vesz ki a veremből (z), és nem tesz vissza semmit (ε). Ezen sor végrehajtása után a verem aktuális 1-es csökken.

4. Az automata egy jelet vesz ki a veremből (z), és egy jelet tesz vissza (z). Ezen sor végrehajtása után a verem aktuális mérete nem változik.

7.16. A verem automata működése

A verem automata nem mindig áll meg. Előfordulhat, hogy az input szalagról végtelen lépésen keresztül sem olvas, csak veremműveletetek végez (olvas a veremből, és ír). Vizsgáljunk meg néhány esetet:

- Ekkor csak a verem változik, meg a belső állapot. A veremautomata akkor kerül ilyen végtelen ciklusba, ha a lépéssorozatban előfordul az, hogy kétszer is ugyanazon belső állapotban van, és a verem tetején is ugyanaz a jel áll.

- Ha egy helyes szót írunk az input szalagra az automata legfeljebb n lépés után megáll, és elfogadó állapotba kerül (mikor jól választja meg a lépéssorozatot).

- De előfordulhat az is, hogy az automata legfeljebb n lépés után megáll, de nem elfogadó állapotban van (ekkor nem determinisztikus, és rossz útvonalat választott).
- És végül az automata megállhat (akár kevesebb, mint n lépésen belül, de akár több mint n lépés után is) parciális okokból, és ha rossz útvonalat választott.
- Az automata nem áll meg (nem determinisztikus, végtelen ciklusba esik).

Ha egy helytelen szót táplálunk az automatába:

- Az automata legfeljebb n lépés után megáll, de nem elfogadó állapotban van (nem determinisztikus és determinisztikus esetben is),
- Az automata legfeljebb n lépés után megáll parciális okokból (nem determinisztikus és determinisztikus esetben is),
- Az automata nem áll meg (ekkor még determinisztikus is lehet).

7.17. Veremautomata számítási kapacitása

Létezik olyan verem automata, amely nem szerepel az F halmaz. Ezen automata úgy jelzi, hogy felismer egy szót, hogy a működésének végén üres lesz a verem (vagyis a verem tetején levő jel z_0). Ezeket az automatákat üres veremmel felismerő automatáknak nevezzük.

Minden $G(K, V, W, \delta, q_0, z_0)$ üres veremmel felismerő verem automatához konstruálható olyan $G(K, V, W, \delta, q_0, z_0, F)$ verem automata, amelyek ekvivalensek. Ez az állítás fordítva is igaz.

A hagyományos veremautomatát úgy kell átalakítanunk, hogy az a megállás előtt ürítse ki a vermet egy speciális delta függvénybeli lépéssorozattal, melynek során ha az input szalagról ε szimbólumot olvas, akkor a veremből egy jelet olvas, és a helyére minden egy ε-t tesz vissza.

Amennyiben egy üres veremmel felismerő verem automatát szeretnénk átalakítani, akkor a felismerés végén csak akkor adjunk vissza elfogadó állapotot, ha az automata elfogadó állapotban van, és a verem üres, vagyis a veremben kizárólag a z_0 szimbólum található.

51. Definíció (Veremautomata számítási kapacitása). *A verem automaták automataosztályának számítása kapacitása megegyezik a környezetfüggetlen nyelvek osztályával.*

52. Definíció (Az üres veremmel felismerő számítási kapacitása). *Az üres veremmel felismerő verem automaták automataosztályának számítása kapacitása megegyezik a hagyományos veremautomaták automataosztály számítási kapacitásával.*

Mindezek alapján elmondhatjuk, hogy minden 2-es típusú nyelvhez konstruálható olyan verem automata, amely az adott nyelvet felismeri, valamint, a veremautomaták általa felismert nyelv legalább 2-es típusú.

A veremautomaták felülről kompatibilisek a véges automatákkal, ugyanis ha a veremautomata mindig z_0-t olvas és ír a verembe, és minden lépésnél olvas a szalagról, akkor visszakapjuk a véges automatát.

Más részről viszont ha egy veremautomata felismer egy nyelvet, arról lehet tudni, hogy legalább 2-es típusú, de lehet akár 3-as is.

7.18. Példa veremautomata

A példa algoritmusban szereplő automatában implementált nyelvtan a következő: $P = \{S \to 1S1, S \to 0S0, S \to \varepsilon\}$, az automata elfogadó állapota a B, és az automata üres veremmel felismerő automata.

K	V ∪ {ε}	W	K	W*
q0	1	z0	q0	1
q0	0	z0	q0	0
q0	0	1	q0	10
q0	0	0	q0	00
q0	1	1	q0	11
q0	1	0	q0	01
q0	1	1	A	ε
q0	0	0	A	ε
A	1	1	A	ε
A	0	0	A	ε
A	ε	z0	B	z0

A bevezető alapján a feldolgozó algoritmus a következőképpen írható le:

```
AktAllapot  := q0
VeremBerak( Z0 )
```

```
 FejIndex  := 0
 NemDetMukodesVolt := HAMIS
 CIKLUS VÉGTELEN
    VeremTeteje := VeremKiolvas()
    Jel1 := ""
    Jel2 := Szalag[ FejIndex ]
    FvOutput1 :=
    DeltaTablazat( Jel1,VeremTeteje, AktAllapot )
    FvOutput 2 :=
    DeltaTablazat( Jel1,VeremTeteje, AktAllapot )
    N1 := Darabszama( FvOutput 1 )
    N2 := Darabszama( FvOutput 2 )
    ELÁGAZÁS
HA N1=0  ÉS N2=0 AKKOR
Ciklus_Befejez
HVEGE
HA N1>0 ÉS N2>0 AKKOR
NemDetMukodesVolt := IGAZ
Valasztas := VeletlenSzam( 0..1 )
HA Valasztas=0 AKKOR
    JEL := Jel1
    FvOutput := FvOutput 1
KULONBEN
    JEL := Jel2
    FvOutput := FvOutput 2
HVÉGE
      HVÉGE
      HA N1=0 ÉS N2>0 AKKOR
JEL := Jel2
FvOutput := FvOutput 2
      HVÉGE
      HA N1>0 ÉS N2=0 AKKOR
JEL := Jel1
FvOutput := FvOutput 1
      HVÉGE
    EVÉGE
    N := Darabszama( FvOutput )
    Valasztas := 0
    HA N>1 AKKOR
  Valasztas := veletlenszam( 0 .. N-1 )
  NemDetMukodesVolt := IGAZ
```

```
    HVEGE
    VerembeKiirSzo(FvOutput[ Valasztas ].VeremSzo )
    AktAllapot := FvOutput[ Valasztas ].UjBelsoAllapot
    HA JEL > "" AKKOR
FejIndex++
    HVÉGE
  CVÉGE
  HA FejIndex == SzoHossza AKKOR
    HA AktAllapot eleme ElfogadoAllapotok AKKOR
      Kiiras: "Az automata az input szót elfogadta"
    KULONBEN
      HA NemDetMukodesVolt AKKOR
Kiiras:"Nem fogadta el,
de Nemdeterminisztikus feldolgozas volt!"
Kiiras:"Újabb futtatás lehet más eredményt ad!"
      KULONBEN
Kiiras: "Az input szó biztosan helytelen!"
      HVEGE
    HVEGE
  KULONBEN
    HA NemDetMukodesVolt AKKOR
      Kiiras:"Nem fogadta el, de
      Nemdeterminisztikus feldolgozas volt!"
      Kiiras:"Újabb futtatás lehet más
      eredményt ad!"
    KULONBEN
      Kiiras: "Az input szó biztosan
      helytelen (parciállis leállás)!"
    HVEGE
  HVÉGE
```

Magyarázat az algoritmushoz: A működés során ki kell próbálni, hogy vannak-e a delta függvényben output sorok arra az esetre, ha nem olvasnánk jelet az input szalagról, valamint azt is meg kell vizsgálnunk, hogy ha beolvassuk a következő jelet, akkor van-e a delta függvényben erre vonatkozó eredmény.

Amennyiben mindkettő igaz, mert vannak output sorok, akkor a delta függvény máris nem-determinisztikus. A ciklus akkor áll le, ha adott helyzetben egyáltalán nincs delta output. Ez előfordulhat azért, mert elfogyott a szó, de előfordulhat menet közben is. Ez utóbbi esetben biztosan nem fogadjuk el a szót.

Ha a szó elfogyott, akkor még meg kell vizsgálni, hogy elfogadó állapotban van-e az automata.

A választott output tartalmazza a verembe írandó jelsorozatot is, és az új választott állapotot is. A szalagon a fejet csak akkor kell léptetni, ha olvastunk a szalagról.

A VeremKiolvas() függvénynek vissza kell adnia a veremből a legfelső szimbólumot. Amennyiben a verem üres lenne, úgy a verem-üres szimbólumot z_0-t kell visszaadni.

8. fejezet

Turing gépek

8.1. Turing gépek

A Turing gépek olyan automaták, amelyeknek nincs külön output szalagjuk, de az "input" szalagról nem csak olvasni képesek, hanem írni is.

Ezen túl az input szalag mindkét irányban végtelen, és a szalagon induláskor az ellenőrzendő szó szerepel balról jobbra folytonosan felírva, a többi cellában pedig egy speciális jel, a BLANK jel.

A Turing gép nem miníg áll meg. A Turing gépeknél a végtelen szalagja miatt soha nem lép le arról. A Turing gép kizárólag a parcialitás miatt állhat meg, ezért a delta függvénye függvénye mindig parciális kell, hogy legyen.

Ebben az esetben a delta függvény parcialitása nem hiba, és nem megoldandó probléma, hanem éppen hogy jó tulajdonság.

53. Definíció (Turing gép). *A* $A(K, V, W, \delta, q_0, B, F)$ *formális hetest Turing automatának nevezzük, ahol*

- K*: az automata belső állapotainak véges halmaza,*
- V*: az input ABC, vagyis a felismerendő nyelv ábécéje,*
- W*: output ABC, vagysis a szalagra írható jelek ábécéje, ahol* $V \subseteq W$*,*
- δ *: az állapotátmeneti függvény,* $\delta \subseteq KxW \to KxWx\{\leftarrow, \rightarrow\}$*,*
- q_0*: az automata kezdőállapotai,* $q_0 \in K$*,*
- B*: blank jel,* $B \in W$*,*
- F*: elfogadó állapotok véges halmaza* $F \subseteq K$*.*

Az automata működése a következő lépésekre bontható fel. Induláskor az automata q_0 kezdő állapotban van, és az input szalagon az input szó jelei vannak felírva, balról-jobbra folytonosan, valamint az író/olvasó fej az input szalag legbaloldalibb cellája fölött áll.

Működés közben az olvasó fej az input szalagról beolvas egy szimbólumot, majd a szimbólum, és az aktuális belső állapot ismeretében, delta függvényben megfogalmazottak szerint belső állapotot vált. Ekkor egy jelet ír vissza a szalag aktuális cellájába, és lépteti a fejet vagy balra, vagy jobbra.

Az automata megáll, ha δ függvény (parciális) nincs értelmezve az aktuális input jel és belső állapotra.

Bár a Turing gép minden ciklusléptésben olvas egy jelet az input szalagról, de lelépni képtelen róla, mivel az végtelen mindkét irányban. Így az automata nem biztos, hogy megáll n lépés után, és az sem biztos, hogy megáll.

Turing gépek konfigurációja, számítási folyamat, felismerés

54. Definíció (Turing gép konfigurációja). *Egy $G(K, V, W, \delta, q_0, B, F)$ Turing gép konfigurációja egy (α, q, β) formális hármas, ahol α az input szalagon az író/olvasó fejtől balra levő szó, q az aktuális belső állapot, β az író/olvasó fej alatti, és tőle jobbra levő szó.*

A fej a β szó első karakterén áll. Az α, β szavak nem tartalmazzák a végtelen sok BLANK jelet, vagyis az α előtt, és a β után már csupa BLANK jel áll.

Az automata konfigurációja alapján lehet tudni minden lényeges információt a működés közbeni állapotról, és ennek ismeretében, a konfiguráció visszatöltése után az automata képes folytatni a folyamatot.

A Turing gép kezdő konfigurációja $(\varepsilon, q_0, \omega)$ hármas, ω az input szó. A működés közbeni konfigurációja valamely (α', q', β') hármas A befejező konfigurációja valamely (α, q, β) hármas.

A delta függvény egy (q, a) páron van értelmezve $(q \in K, a \in W)$. $\delta(q, a) = (q', a', \leftarrow)$ alakú sorokból áll, ahol a (q, a) párhoz hozzárendel egy új állapotot (q'), és egy jelet visszaír a szalagra (a'), és megadja, hogy a fejnek balra, vagy jobbra kell lépnie.

55. Definíció (Turing gép számítási folyamata). *A Turing gép valamely (α, q, β) konfigurációja megállító konfiguráció, ha a δ függvény a (q, a) párra nincs értelmezve.*

A konfigurációknak egy olyan sorozatát, melynek első eleme a kezdő konfiguráció, minden további eleme az előzőből kapható a δ függvény alkalmazásával, számítási folyamatnak nevezzük.

56. Definíció (Elfogadó állapot). *Egy számítási folyamatot teljes számítási folyamatnak nevezzük, ha a sorozat utolsó konfigurációja megállító konfiguráció, valamint a Turing gép egy $\omega \in V*$ szót felismer (elfogad), ha az $(\varepsilon, q_0, \omega)$ kezdő konfiguráció egy teljes számítási folyamat során átvihető valamely (α, q, β) megállító konfigurációba, és $(q \in F)$.*

A fenti definíció minden esetre jó. alkalmazható a nem determinisztikus, és a determinisztikus esetekre is. A parciális-nem parciális esetet viszont nincs értelme külön venni, mert a Turing gép nem lehet teljes.

A Turing gépek működésének elemzése

A Turing gép nem mindig áll meg. Előfordulhat, hogy az input szalag két cellája között alternáló mozgást végez a végtelenségig. De akár az is, hogy minden jel beolvasása után folyamatosan jobbra (vagy mindig balra) lép a végtelenségig.

Amennyiben a delta függvény nem parciális, úgy a Turing gép soha nem fog megállni.

A Turing gép által a végtelen szalagra írt szó nem feltétlenül folytonos. A felírt szó-szakaszok között BLANK cellák állhatnak.

Nézzünk meg néhány, az automata működése közben előforduló lehetséges esetet, mikor helyes szót adunk az automatának elemzésre.

- , az valahány (akár kevesenn mint n) lépés után megáll, és elfogadó állapotba kerül.
- Az automata valahány lépés után megáll, de nem elfogadó állapotban van (nemdeterminisztikus, és rossz útvonalat választott).
- Az automata nem áll meg (nemdeterminisztikus eset, rossz az útvonal).

És vizsgáljunk meg olyan eseteket is,mikor helytelen az input szó.

- Az automata valahány lépés után megáll, de nem elfogadó állapotban van (nemdeterminisztikus és determinisztikus esetben is).
- Az automata nem áll meg.

Elemezzünk most egy konkrét példát, amely megmutatja, hogyan elemez egy adott szót egy olyan Turing gép, amely a következő konponensekből épül fel: Az automata input ábécéje a $V := \{0, 1\}$, a belső állapotok halmaza a $K := \{a, b, c, d, e\}$, és a szalagra visszarható jelek halmaza a $W := \{0, 1, B\}$, és az elfogadó állapotok egyelemű halmaza a az $\{a\}$.

Az automata delta függvénye táblázatos formában a következő:

δ	K	V	K	W	Irány
	a	0	b		←
	a	1	c		←
	a	B	-	-	—
	b	0	b	0	←
	b	1	b	1	←
	b	B	d	B	→
	c	0	c	0	←
	c	1	c	1	←
	c	B	c	B	→
	d	0	d	0	→
	d	1	e	0	→
	d	B	a	0	→
	c	0	d	1	→
	c	1	e	1	→
	c	0	a	1	→

A delta függvény nem a szavak felismerését végzi el, hanem megfordítja, vagyis tükrözi azt a közepére. (Az input szó 01011.)

A fenti automata egy másik lehetséges megadása a következő ábrán látható:

δ	α	b	c	d	E
0	B b	0 b←	0 c←	0 d→	0 d→
1	B c	1 b←	1 c←	0 e→	1 e→
B		B b→	B e→	0 d→	1 a→

57. Definíció (rekurzívan felsorolható nyelvek). *Egy nyelvről azt mondjuk, hogy rekurzívan felsorolható, ha van olyan Turing gép, amely az adott nyelvet felismeri, valamint egy nyelv rekurzív, ha létezik olyan Turing gép, amely az adott nyelvet felismeri.*

Az automata minidg megáll, még azokra a szavakra is, amelyek nem elemei a nyelvnek.

A mondatszerkezetű nyelvek osztálya megegyezik a rekurzívan felsorolható nyelvek osztályával, vagyis a Turing gépek a 0-s típusú nyelvek felismerő

automatái Mindezek alapján minden 0-ás típusú nyelvhez konstruálható felismerő Turing gép, és viszont. Az 1-es típusú nyelvek felismerő automatái speciális Turing gépek.

8.2. Turing gépek változatai

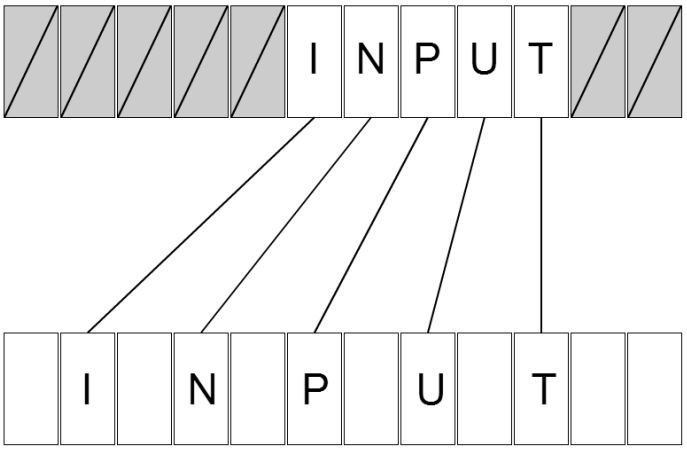
Az egyirányú végtelen szalaggal rendelkező Turing gépek a szalag csak egyik irányban végtelen, a másik irányban le lehet róla lépni.

Be lehet bizonyítani, hogy ezen automaták ekvivalensek a mindkét irányban végtelen szalagúakkal.

Ennek módszere az, hogy a kétirányban végtelen szalagú Turing gép celláihoz hozzárendeljük az egyirányban végtelen szalag celláit oly módon, hogy a páros cellákba kerülnek a jobb oldali cellák, a páratlan cellákba a bal oldali cellák.

Ezután átalakítjuk az eredeti, mindkét irányban végtelen szalaggal dolgozó Turing gép delta függvényét úgy, hogy a megfelelő cellára lépés módját az új technikával végezze, akkor a két automatában azonos felismerési képességet nyerünk, vagyis a két automata ekvivalens lesz.

A másik változat a Több szalagú Turing gépek. Ennek a típusnak több, esetleg minkét irányban végtelen hosszú szalagja van. A delta függvény "input" adatai nem csak egy szalagról származnak, hanem mindegy ikről, és mindegyikre minden lépésben írni kell, valamint minden szalag író olvasó fejét léptetnie kell valamilyen irányba. Ez az automata is lehet ekvivalens a végtelen szalagú Turing géppel.



Turing-gép leírás elemzése

```
  AktAllapot  := q0
  FejIndex  := 0
  NemDetMukodesVolt := HAMIS
  CIKLUS VÉGTELEN
    Jel := Szalag[ FejIndex ]
    FvOutput := DeltaTablazat( Jel, AktAllapot )
    N := Darabszama( FvOutput )

    ELÁGAZÁS
      HA N=0  AKKOR
Ciklus_Befejez
      HVEGE
      HA N=1 AKKOR
  Valasztas := 0
      HVEGE
      HA N>1 AKKOR
    NemDetMukodesVolt := IGAZ
    Valasztas := VeletlenSzam( 0..N-1 )
      HVÉGE
    EVÉGE

    Szalag[ FejIndex ] := FvOutput[ Valasztas ].OutputJel
    AktAllapot     := FvOutput[ Valasztas ].UjBelsoAllapot
    HA FvOutput[ Valasztas ].FejIrany == JOBBRA AKKOR
      FejIndex++
    KÜLÖNBEN
      FejIndex--
    HVÉGE
  CVÉGE

  HA AktAllapot eleme ElfogadoAllapotok AKKOR
    Kiiras: "Az automata az input szót elfogadta"
  KULONBEN
    HA NemDetMukodesVolt AKKOR
  Kiiras:"Nem fogadta el (nemdet.)
  Kiiras:"Több eredmény lehet"
    KULONBEN
Kiiras: "Az input szó helytelen!"
    HVEGE
```

```
HVÉGE
```

Előfordulhat, hogy a Turing gép végtelen ciklusba kerül. Ezt a lehetőséget nagyon nehéz felismerni, bár vannak rá módszerek. Általános esetben megjósolni, hogy egy Turing gép egy konkrét szót már nem fog felismerni - lehetetlen.

8.3. Lineárisan korlátolt Turing gépek

A lineárisan korlátolt Turing gépek jellemzői, hogy az input/output szalag nem végtelen, hanem mindkét irányban véges.

Az input szalag hossza attól függ, hogy milyen hosszú az input szó, és léteznek olyan Turing gépek, amelyek a szó felismerése közben pont olyan hosszú szalaggal dolgoznak, mint maga az input szó hossza.

Minden ilyen típusú Turing gépet egy olyan automata osztályba sorolhatunk, ahol az osztálynak az a jellemző attribútuma, hogy hányszoros hosszú szalaggal dolgoznak, de minden ilyen osztályba tartozó gép ekvivalens az egyszeres szalaggal dolgozó automata osztályba tartozókkal.

Mivel a véges szalagú Turing gép akkor is megállhat, ha lelép a szalagról, így a megállás, és elfogadás definíciója más, mint a végtelen szalagú Turing gépeknél.

Ahhoz, hogy ezt belássuk, csak annyit kell tennünk, hogy kiegészítjük az input szót balról egy $\rightarrow$, jobbról pedig egy $\leftarrow$ jellel. Az egyik jelzi a szó bal végét, a másik a jobb végét. Az automata akkor lépett le a szalagról, ha ezen jelek valamelyikét eléri.

58. Definíció (A felismert nyelv). *A lineárisan korlátolt Turing gépek által felismert nyelvek környezetfüggőek, és viszont.*

A Turing gépekről, és egyáltalán az automatákról tanultak alapján könnyedén beláthatjuk, hogy a Turing gép a számítógépek matematikai modelljeként is felfogható.

Az input output szalag a számítógép memóriája, és ha jobban megismerjük az automata működési elvét, akkor felismerhetjük az egyes Neumann elveket is, mint a tárolt program elve.

Ahhoz, hogy teljesen tisztában legyünk ezen automata osztály működésével, mindenképpen készítsük el a fejezetben ismertetett feldolgozó algoritmus valamely általunk választott, és lehetőleg magas szintű programozási nyelven implementált változatát.

9. fejezet

Jegyzékek

9.1. Irodalomjegyzésk

Hernyák Zoltán: Formális nyelvek és automaták EKTF, jegyzet 2006
http://aries.ektf.hu/ serial/kiralyroland/download/FormNyelv_v1_9.pdf

Csörnyei Zoltán: Fordítóprogramok, Typotex Kiadó, 2006 ISBN: 9639548839

Bach István:Formális nyelvek, Typotex, 2001, ISBN 9639132 92 6
http://www.typotex.hu/download/formalisnyelvek.pdf

Révész György: Bevezetés a formális nyelvek elméletébe, Akadémiai Kiadó, Budapest, 1979.

Demetrovics János – Jordan Denev – Radiszlav Pavlov: A számítástudomány matematikai alapjai, Nemzeti Tankönyvkiadó, Budapest, 1994.

B. A. Trahtenbrot: Algoritmusok és absztrakt automaták, Műszaki Könyvkiadó, Budapest, 1978.

Ádám András – **Katona Gyula** – **Bagyinszki János**: Véges automaták, MTA jegyzet, Budapest, 1972.

Varga László: Rendszerprogramozás II., Nemzeti Tankönyvkiadó, Budapest, 1994.

Fazekas Attila: Nyelvek és automaták, KLTE jegyzet, Debrecen, 1998