

Formális nyelvek és a fordítóprogramok alapjai

2023/2024/2. félév, B szakirány

NAGY SÁRA előadásai és gyakorlatai, valamint
DR. HORPÁCSI DÁNIEL előadásai alapján

Utolsó módosítás: 2024. június 4.

Tartalomjegyzék

1. Szavak és nyelvek	1
1.1. Alapvető fogalmak	1
1.2. Műveletek	2
1.2.1. Műveletek szavak felett	2
1.2.2. Műveletek nyelvek felett	3
2. Nyelvtanok és osztályozásuk	5
2.1. Nyelvek definiálási módjai	5
2.2. Nyelvtanok	5
2.3. A Chomsky-féle grammatikatípusok	7
2.4. Nyelvtani transzformációk	9
2.4.1. Epsilon-mentesítés (ϵ -mentesítés)	9
2.4.2. Nyelvek normálformája	10
2.5. Automaták	11
2.6. Zártsági tételek	11
3. Reguláris (3-as típusú) nyelvtanok	13
3.1. Reguláris nyelvek	13
3.2. 3-as típusú nyelvtanok normálformája	15
3.3. Véges automaták	16
3.3.1. 3-as típusú nyelvek kapcsolata a véges automatákkal	18
3.3.2. Minimális véges determinisztikus automata	19
4. A 2-es és 3-as nyelvcsalád viszonya	21
4.1. Szükséges feltétel 3-as típusú nyelvekre	21
4.2. Szükséges és elégséges feltétel 3-as típusú nyelvekre	21
5. Környezetfüggetlen (2-es típusú) nyelvtanok	23
5.1. A szóprobléma kérdése	23
5.2. 2-es típusú nyelvtanok normálformája	24
5.3. Nyelvtan redukálása	24
5.4. Veremautomaták	25
6. Fordítóprogramok	29
6.1. Fajtái	30
6.1.1. Fordított programozási nyelvek	30
6.1.2. Értelmezett programozási nyelvek	30
6.1.3. Fordítás végrehajtás közben	32
6.2. Fordítóprogramok fejlődése	33

6.3.	Logikai felépítése	33
6.3.1.	Analízis	33
6.3.2.	Szintézis	35
6.4.	Szerkesztés és végrehajtás	37
7.	Lexikális elemzés	39
7.1.	A tokenizáció	39
7.2.	A lexikális elemzés elvei	40
7.3.	Implementációja	40
7.4.	Tokenhez csatolt információk	42
7.5.	Lexikális hibák	42
8.	Szintaktikus elemzés	43
8.1.	Grammatikai előfeltételek	43
8.2.	Felülről lefele elemzés	45
8.3.	Alulról felfele elemzés	50
9.	Szemantikus elemzés	53
9.1.	Szimbólumtábla	53
9.2.	Attribútumnyelvtan	55
10.	Az Assembly alapjai	61
10.1.	Az Assembly forrásfájl	62
11.	Kódgenerálás	65

Előszó

Ez a jegyzet az ELTE IK *Formális nyelvek és a fordítóprogramok alapjai* c. tantárgy anyagát dolgozza fel, amit B szakirányon (*Szoftvertervező specializáció*) tanítanak. A tantárgy több, korábbi tárgynak az összeillesztéséből alakult ki, emiatt eltér attól, amit más szakirányokon oktatnak. Az a legjelentősebb eltérés, hogy az elméleti anyag leginkább a fordítóprogramok részhez szükséges ismereteket készíti elő. A formális nyelvekről szóló előadásokat NAGY SÁRA, a fordítóprogramokról szólókat meg DR. HORPÁCSI DÁNIEL tartották.

A jegyzet fejezetenként feldolgoz egy-két előadást. Bizonyos „előadásokat” összeolvasztottam, mert didaktikai szempontból egybetartoztak, másokat meg szétszedtem. A fejezetek végére odaillesztettem azon feladatsorokat, melyeket a gyakorlaton néztünk. Ezek is a témák átszervezése szerint több témakört is érintenek. Megoldókulcsot nem tartalmaz, ám az elméleti részek közé igyekeztem magyarázatokat beszúrni, melyek segíthetnek a gyakorlati feladatok megoldásában.

A jegyzet készítésének idején a vizsgán nem kérték számon a tételek bizonyítását, így ennek szellemében vázlatosabban voltak leadva az előadásokon. Akit érdekelnek, azok az alábbi jegyzetek közül szemezgethetnek – valamint ezeket használtam fel ezen jegyzet elkészítéséhez. Egy apró megjegyzés: az [1.]-es forrásra néha úgy hivatkozok, hogy „*a régi jegyzet*”, de ez senkit ne tévesszen meg. Az online elérhető jegyzetekhez kattintható hivatkozást is mellékeltem.

Igyekeztem a legjobb tudásom szerint összeállítani a jegyzetet, ennek ellenére előfordulhatnak benne elgépelések, hibák, stb. Ha találsz ilyet, kérlek értesíts e-mailben a(z) ap3558@inf.elte.hu címen.

Sikeres felkészülést kívánok!

Kiss-Bartha Nimród

Felhasznált források:

- [1.] Dr. Hunyadvári László, Manhertz Tamás – Automaták és formális nyelvek (*Utolsó frissítés: 2006. január 13.*) [pdf]
- [2.] Az előadások diasorai (2024.) (*Canvason elérhető*)
- [3.] Dr. Ásványi Tibor – Algoritmusok és adatszerkezetek II. előadásjegyzet [pdf]
- [4.] további források

1. fejezet

Szavak és nyelvek

1.1. Alapvető fogalmak

1.1.1. definíció (Ábécé). Egy Σ véges és nemüres halmazt **ábécének** hívunk. Ennek elemeit **betűknek** hívjuk.

Az ábécé jele a szakirodalomban változhat – például az előadáson V -vel jelöltük, azonban *Algoritmusok és adatszerkezetek II.*-ből Σ volt a jele. A jegyzet ezt az utóbbit fogja használni.

1.1.2. definíció (Szó és hossza). Az $u \in \Sigma^*$ véges sorozatot egy **sztringnek** vagy **szónak** nevezzük, melynek hosszát az $\ell : \Sigma^* \rightarrow \mathbb{N}$ függvény jelöli úgy, hogy

$$\forall u \in \Sigma^* : 0 \leq \ell(u) < \infty.$$

Speciális esete az **üres szó**, melynek jele $\boxed{\varepsilon}$ és $\ell(\varepsilon) = 0$.

A *sztring* és *szó* elnevezés felcserélhető, ám jellemzően szónak hívunk egy véges betűsorozatot, ha tudjuk róla, hogy az egy nyelvnek egy szava.

A Σ^* jelöli azon véges sorozatok halmazát, melyeket a Σ ábécé betűiből képeztünk. Ennek eleme az üres sorozat vagy üres szó is, azaz $\boxed{\varepsilon} \in \Sigma^*$. Megállapodunk abban, hogy

$$\boxed{\Sigma^+ := \Sigma^* \setminus \{\varepsilon\}}.$$

A **legsűkebb ábécét** egy szóra nézve az alábbi módon jelöljük: $\Sigma(u) \subseteq \Sigma$.

Egyes szerzők az abszolútérték jelet használják a szó hosszának jelölésére, azaz $|u| = \ell(u)$. A jegyzet az ℓ betűvel fogja jelölni, ugyanis ez kevésbé félreérthető.

Lekérdezhetjük, hogy egy adott szó mennyit tartalmaz egy adott betűből. Például ha $\Sigma := \{a, b\}$, akkor

$$\ell_a(u) \quad (u \in \Sigma^*)$$

azt jelöli, hány darab a betű található az u szóban.

1.1.3. definíció (Nyelv). A $L \subseteq \Sigma^*$ halmazt **nyelvnek** nevezzük (azaz a nyelv egy halmaza, ami szavakat tartalmaz).

Speciális nyelvek:

- üres nyelv: $\boxed{\emptyset}$ vagy $\boxed{L_\emptyset}$
- üres szót tartalmazó nyelv: $\boxed{\{\varepsilon\}}$ vagy $\boxed{L_\varepsilon}$

Annak ellenére, hogy a halmaz rendezetlenül tárolja az elemeit, hagyományosan **lexikografikus sorrendben** szoktuk felsorolni a nyelv szavait. Ez ábécé szerinti elsődleges és hossz szerinti másodlagos rendezést jelent.

1.1.4. definíció (Nyelvcsalád). Legyenek $L_1, L_2, \dots, L_k \subseteq \Sigma^*$ ($k \in \mathbb{N}^+$) nyelvek egy ábécé felett. Ekkor a $\mathcal{L} := \{L_1, L_2, \dots, L_k\}$ halmazt **nyelvcsaládnak** vagy **nyelvosztálynak** hívjuk.

1.2. Műveletek

1.2.1. Műveletek szavak felett

1.2.1. definíció (Konkatenáció). Legyen $u := u_1u_2 \dots u_n$ és $v := v_1v_2 \dots v_m$ két szó Σ^* felett ($u, v \in \Sigma^*$). Ekkor

$$uv := u_1u_2 \dots u_nv_1v_2 \dots v_m$$

az u és v **konkatenációja** ($uv \in \Sigma^*$). Jele általában nincs (néha ponttal (\cdot) jelezzük).

A konkatenáció tulajdonságai – $\forall u, v, w \in \Sigma^*$:

1. asszociatív: $u(vw) = (uv)w$,
2. nem kommutatív: $uv \neq vu$,
3. Σ^* -ra zárt művelet (nem vezet ki a halmazból),
4. egységeleme az üres szó (ε): $\varepsilon u = u\varepsilon = u$,
5. $(\Sigma^*, \cdot, \varepsilon)$ egy egységelemes félcsoporthat alkot.

1.2.2. definíció (Hatványozás). Legyen $u \in \Sigma^*$ és $n \in \mathbb{N}$.

$$u^n := \begin{cases} \varepsilon & (n = 0) \\ u & (n = 1) \\ u^{n-1}u & (n > 1). \end{cases}$$

A fenti definíció *balrekurzív*, de ugyanúgy működne, ha *jobbrekurzív*an definiálnánk.

A konkatenációt és a hatványozást **reguláris műveleteknek** nevezzük.

1.2.3. definíció (Megfordítás). Legyen $u_1 u_2 \dots u_n =: u \in \Sigma^*$. Ekkor

$$u^R := u^{-1} := u_n u_{n-1} \dots u_2 u_1$$

Jele: $\boxed{u^{-1}}$ vagy $\boxed{u^R}$.

Ismertetünk további, szavakkal kapcsolatos alapfogalmakat. $\forall u, v \in \Sigma^*$,

1. **Részszó:** $\exists w_1, w_2 \in \Sigma^* : u = w_1 v w_2$.
2. **Prefix:** $v \sqsubseteq u \iff \exists w \in \Sigma^* : u = v w$.
3. **Suffix:** $u \sqsupseteq v \iff \exists w \in \Sigma^* : u = w v$.
4. **Valós prefix (\sqsubset), valós suffix (\sqsupset):** a megfelelő definíció, továbbá $v \neq \varepsilon \wedge v \neq u$.

A jelöléseket az *Algoritmusok és adatszerkezetek II.* jegyzetből kölcsönöztem.

1.2.2. Műveletek nyelvek felett

Az eddig megismert, szavakon értelmezett műveleteket kiterjesztjük a nyelvek szintjére, valamint a már jól ismert halmazműveleteket is megvizsgáljuk, hogyan viselkednek a nyelvek felett.

1.2.4. definíció (Unió). Legyen $L_1, L_2 \subseteq \Sigma^*$. Ekkor

$$L_1 \cup L_2 := \{u \in \Sigma^* \mid u \in L_1 \vee u \in L_2\}.$$

Tulajdonságai:

1. kommutatív: $L_1 \cup L_2 = L_2 \cup L_1$
2. asszociatív: $L_1 \cup (L_2 \cup L_3) = (L_1 \cup L_2) \cup L_3$
3. egységeleme a(z) üres nyelv (L_\emptyset): $L \cup L_\emptyset = L_\emptyset \cup L = L$

1.2.5. definíció (Metszet). Legyen $L_1, L_2 \subseteq \Sigma^*$. Ekkor

$$L_1 \cap L_2 := \{u \in \Sigma^* \mid u \in L_1 \wedge u \in L_2\}.$$

1.2.6. definíció (Komplementer). Legyen $L \subseteq \Sigma^*$. Ekkor

$$\bar{L} := \Sigma^* \setminus L.$$

Tulajdonságai:

1. $L \cup \bar{L} = \Sigma^*$
2. $L \cap \bar{L} = L_\emptyset$

1.2.7. definíció (Konkatenáció). Legyen $L_1, L_2 \subseteq \Sigma^*$. Ekkor

$$L_1 L_2 := \{uv \mid u \in L_1 \wedge v \in L_2\}.$$

Tulajdonságai:

1. a nyelvek felett is asszociatív, de nem kommutatív (ahogyan a szavak esetében)
2. egységeleme az üres nyelvet tartalmazó nyelv (L_ε): $LL_\varepsilon = L_\varepsilon L = L$
3. a nyelvek halmaza a konkatenációra nézve egység elemes félcsoport alkot
4. kétoldali disztributivitás áll fenn az unióval:

$$\begin{aligned} L(L_1 \cup L_2) &= LL_1 \cup LL_2 \\ (L_1 \cup L_2)L &= L_1 L \cup L_2 L \end{aligned}$$

5. vigyázat: a metszettel nem áll fenn a disztributivitás:

1.2.8. definíció (Nyelv hatványa). Legyen $L \subseteq \Sigma^*$ és $n \in \mathbb{N}$.

$$L^n := \begin{cases} L_\varepsilon & (n = 0) \\ L & (n = 1) \\ L^{n-1}L & (n > 1). \end{cases}$$

Felhívjuk a figyelmet a következő, látszólag hasonló, ám eltérően működő műveletre.

1.2.9. definíció (Nyelv megfordítása). Legyen $L \subseteq \Sigma^*$ nyelv. Ekkor

$$L^{-1} := \{u^{-1} \mid u \in L\}$$

jelöli az L nyelv megfordítását.

A nyelv megfordításának jelentése: minden szavát megfordítjuk. Ellenben a **nyelv hatványa emelése** arról szól, hogy a nyelv szavait összekonkatenáljuk egymással az összes lehetséges módon – vagyis **nem szavankénti hatványozást jelent!**

A következő művelet a hatványozást „emeli egy magasabb szintre”.

1.2.10. definíció (Nyelv lezártja, iteráltja). Legyen $L \subseteq \Sigma^*$. Ekkor

$$L^* := L^0 \cup L^1 \cup L^2 \cup L^3 \cup \dots = \bigcup_{i \geq 0} L^i.$$

Pozitív lezártja:

$$L^+ := L^* \setminus L_\varepsilon = \bigcup_{i \geq 1} L^i.$$

Az alábbi műveleteket nevezzük **reguláris műveleteknek**: unió, konkatenáció, lezáras.

2. fejezet

Nyelvtanok és osztályozásuk

2.1. Nyelvek definiálási módjai

1. Felsorolással: $L := \{pa, \tau a, ka\}$.
2. Logikai formulával (invariánssal): $L := \{a^n b^n \mid n \in \mathbb{N}\}$.
3. Strukturális rekurzióval: megszámlálhatóan végtelen nyelveken végrehajtunk véges számú elemi műveletet.

$$L := \{ab\}^* \{cd\}$$

4. Algoritmussal
5. Matematikai gépekkel (automatákkal)
6. Produkciós rendszerekkel (szabályokkal)

A továbbiakban a produkciós rendszerekkel fogunk részletesebben foglalkozni.

2.2. Nyelvtanok

2.2.1. definíció (Nyelvtan). *Nyelvtannak (vagy grammatikának) nevezzük az alábbi négyest:*

$$G := (N, T, P, S),$$

ahol

- N a **nemterminális jelek** halmaza,
- T a **terminális jelek** halmaza (ábécé),
- P a **produkciós szabályok** halmaza,
- S a **startszimbólum** (vagy kezdőszimbólum).

Kiemelünk pár tulajdonságot, amik a nyelvtan összetevőire teljesülnek.

- $N \cup T = \emptyset$, azaz a nemterminálisok és terminálisok halmaza diszjunktak.
- $S \in N$, azaz a startszimbólum egy nemterminális jel.
- P elemeit **produkciós szabályoknak** nevezzük, melyeket az alábbi módon írunk le:

$$(p, q) \in P \iff p \longrightarrow q \in P.$$

- A szabály bal oldalának alakja: $p \in (T \cup N)^* N (T \cup N)^*$. *Jelentése:* legalább egy nemterminálisnak muszáj szerepelnie a szabály bal oldalán.
- A szabály jobb oldalának alakja: $q \in (T \cup N)^*$.
- A szabály két oldalát a „ \longrightarrow ” jellel választjuk el.
- A $(T \cup N)^*$ halmaz elemeit **mondatformáknak** nevezzük. A fogalom azért szükséges, ugyanis meg akarjuk különböztetni, hogy mikor beszélünk „tisztán” szóról és mikor terminálisok és nonterminálisok vegyes véges sorozatáról.

2.2.2. definíció (Nyelvtan által generált nyelv). Legyen $G := (N, T, P, S)$. Ekkor a G nyelvtan által generált nyelv azon szavak halmazát jelenti, melyek **közvetlenül vagy közvetetten levezethetők a G -ből**, vagyis

$$L(G) := \left\{ u \in T^* \mid S \xrightarrow[G]{*} u \right\}.$$

Pár szót a jelölésről. A $*$ arra utal, hogy mennyi lépésben tudunk eljutni az S kezdőszimbólumból az u szóig. Véges sok lépésszámot kell jelentsen. Akár konkrét értéket is megadhatunk. A G csupán arra utal, hogy a G nyelvtan generálja a szóban forgó szót. Ha a kontextusból egyértelmű, akkor elhagyhatjuk.

Továbbá figyeljük meg, hogy eltérő nyilat (\implies) használunk arra, amikor mondatformából vezetünk le egy szót. Az előző eset (\longrightarrow) csupán a produkciós szabály jobb és bal oldalának elválasztására szolgált.

A definícióban szerepelt olyan megfogalmazás, hogy közvetetten, illetve közvetlenül levezetünk egy szót a startcsúcsból. A levezetés ezen két fajtáját itt definiáljuk.

2.2.3. definíció (Közvetlen levezetés). Legyen $G := (N, T, P, S)$ egy adott nyelvtan, valamint legyen $u, v \in (T \cup N)^*$ két mondatforma. Azt mondjuk, hogy a v **mondatforma közvetlenül levezethető az u mondatformából**, ha

$$\exists u_1, u_2 \in (T \cup N)^*, \exists x \longrightarrow y \in P : u = u_1 x u_2 \wedge v = u_1 y u_2.$$

Jelölése: $u \xRightarrow[G]{*} v$.

2.2.4. definíció (Közvetett levezetés). Legyen $G := (N, T, P, S)$ egy adott nyelvtan, valamint legyen $u, v \in (T \cup N)^*$ két mondatforma. Azt mondjuk, hogy a v **mondatforma közvetetten levezethető az u mondatformából**, ha

$$\exists k \in \mathbb{N}, \exists x_0, x_1, \dots, x_k \in (T \cup N)^*, u = x_0 \wedge v = x_k, \forall i \in [0..k-1] : x_i \xRightarrow[G]{*} x_{i+1}.$$

Jelölése: $u \xRightarrow[G]{*} v$.

Szavakban: létezik egy k elemből álló mondatformák sorozata, melynek legeleje az u és legvége a v . Ezek között egyesével haladva közvetlenül levezethetők az egyes mondatformák úgy, hogy a sorozatban a soron következőbe jutunk el.

2.2.5. definíció (Nyelvek ekvivalenciája). Legyen G_1, G_2 két nyelvtan.

- G_1 és G_2 **ekvivalensek**, ha $L(G_1) = L(G_2)$.
- G_1 és G_2 **kvázi-ekvivalensek**, ha $L(G_1) \setminus L_\varepsilon = L(G_2) \setminus L_\varepsilon$, azaz csak az üres szó generálásában térnek el.

2.2.1. tétel. Nem minden nyelv írható le nyelvtannal.

2.3. A Chomsky-féle grammatikátípusok

2.3.1. definíció (Chomsky-féle grammatikátípusok). A $G = (N, T, P, S)$ nyelvtan i -típusú ($i = 0, 1, 2, 3$), ha P szabályhalmazára teljesülnek a következők:

0. **típus** ($i = 0$) – Nincs korlátozás.

1. **típus** ($i = 1$) – **környezetfüggő nyelvtan:**

P minden szabálya $u_1 A u_2 \rightarrow u_1 v u_2$ alakú, ahol $u_1, u_2, v \in (N \cup T)^*$, $A \in N$, és $v \neq \varepsilon$, kivéve az $S \rightarrow \varepsilon$ alakú szabályt, de ekkor S nem fordul elő egyetlen szabály jobb oldalán sem.^a

2. **típus** ($i = 2$) – **környezetfüggetlen nyelvtan:**

P minden szabálya $A \rightarrow v$ alakú, ahol $A \in N$, $v \in (N \cup T)^*$.

3. **típus** ($i = 3$) – **reguláris nyelvtan:**

P minden szabálya vagy $A \rightarrow uB$ vagy $A \rightarrow u$ alakú ($A, B \in N, u \in T^*$).

^aEzt "Korlátozott ε -szabály"-nak, röviden: KES-szabálynak hívjuk.

Az i -típusú nyelvtanok vagy grammatikák halmazát \mathcal{G}_i -vel jelöljük. A grammatikák alakjából következik, hogy

$$\mathcal{G}_i \subseteq \mathcal{G}_0 \quad (i = 1, 2, 3).$$

$$\mathcal{G}_3 \subseteq \mathcal{G}_2.$$

2.3.2. definíció. Egy L nyelvet i -típusúnak nevezünk ($i \in \{0, 1, 2, 3\}$), ha létezik olyan i -típusú grammatika, ami az L nyelvet generálja, azaz

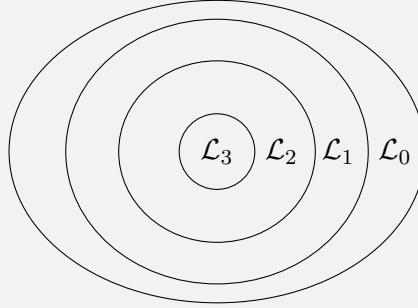
$$\exists G \in \mathcal{G}_i : L(G) = L.$$

Az i -típusú nyelvek halmazát – nyelvcsaládját, nyelvosztályát – jelölje \mathcal{L}_i , azaz

$$\mathcal{L}_i := \{L \text{ nyelv} \mid \exists G \in \mathcal{G}_i : L(G) = L\} \quad (i = 0, 1, 2, 3).$$

2.3.1. tétel (Chomsky-féle hierarchia).

$$\mathcal{L}_3 \subseteq \mathcal{L}_2 \subseteq \mathcal{L}_1 \subseteq \mathcal{L}_0.$$



Megjegyzés. A tartalmazásnak $\mathcal{L}_2 \subseteq \mathcal{L}_1$ része nem triviális az 1-es típusú nyelvek (nyelvtanok) kényszerű definíciója miatt.

A tételnek létezik az **erősebb változata**, mely valódi tartalmazást állít:

$$\mathcal{L}_3 \subset \mathcal{L}_2 \subset \mathcal{L}_1 \subset \mathcal{L}_0.$$

Figyeljük meg, hogy a Chomsky-féle hierarchia **nyelvcsaládokra** és **nem nyelvtanokra** vonatkozik. Emiatt

$$\mathcal{G}_3 \subseteq \mathcal{G}_2 \subsetneq \mathcal{G}_1 \subseteq \mathcal{G}_0.$$

Ha a 2-es típusú szabályoknál is kikötnénk, hogy $v \neq \varepsilon$, akkor igaz lenne a tartalmazás, és akkor triviálisan igaz lenne a nyelvcsaládokra is tartalmazás.

Ha ugyanazon nemterminálishoz több szabály tartozik, akkor tömörebben is felírhatjuk a rá vonatkozó szabályokat az alábbi módon:

$$S \longrightarrow \varepsilon \mid aSb \mid SS,$$

ahol a „ \mid ” jelek amolyan „vagy” jelentéssel bíró elválasztók. Ezt használja ki a **Backus–Naur-jelölés** (angolul **Backus–Naur-form**, röviden **BNF**), melyet a programozási nyelvek szintaxisának felírásához szoktak használni. Például ugyanez a nyelv BNF-fel felírva:

$$\langle \text{start} \rangle ::= \varepsilon \mid a \langle \text{start} \rangle b \mid \langle \text{start} \rangle \langle \text{start} \rangle,$$

ahol a „ $\langle \rangle$ ” jelek olyan *metaszimbólumok*, melyekkel meg lehet címkézni az egyes nemterminálisokat. A „ $::=$ ” felel meg a „ \longrightarrow ” jelölésnek.

Egy egyszerű példa, ami néhány magyar mondat generálására képes. Az egyszerűség kedvéért feltesszük, hogy egyetlen terminális jelből áll a **macska**, **kutya**, stb. szavunk.

$\langle \text{mondat} \rangle$	$::= \langle \text{alany} \rangle \langle \text{állítmány} \rangle .$
$\langle \text{alany} \rangle$	$::= \langle \text{névelő} \rangle \langle \text{főnév} \rangle \mid \langle \text{névelő} \rangle \langle \text{melléknév} \rangle \langle \text{főnév} \rangle$
$\langle \text{névelő} \rangle$	$::= A \mid \text{Egy}$
$\langle \text{főnév} \rangle$	$::= \text{macska} \mid \text{kutya}$
$\langle \text{melléknév} \rangle$	$::= \text{bozontos} \mid \text{kerge}$
$\langle \text{állítmány} \rangle$	$::= \text{eszik} \mid \text{iszik} \mid \text{alszik}$

2.4. Nyelvtani transzformációk

Ahogy korábban be lett vezetve, B szakirányon az elmélet leginkább a fordítóprogramok írásához legszükségesebb ismereteket adja át, emiatt a most következőket csak 3-as típusú nyelvtanokra, nyelvekre fogalmazzuk meg, ugyanis ezen nyelvek teszik lehetővé, hogy

2.4.1. definíció (Nyelvtani transzformáció). *A nyelvtani transzformáció olyan eljárás, amely egy G grammatikából egy másik G' grammatikát készít.*

2.4.2. definíció (Ekvivalens nyelvtani transzformáció). *Ekvivalens nyelvtani transzformációról beszélünk, ha minden G nyelvtanra és az ő G' transzformáltjára igaz, hogy $L(G) = L(G')$.*

2.4.1. Epsilon-mentesítés (ε -mentesítés)

A fordítóprogramok szempontjából fontos transzformációnk az ún. **ε -mentesítés**.

2.4.1. tétel (ε -mentesítés). *Minden $G = (N, T, P, S)$ környezetfüggetlen (2-es típusú) nyelvtanhoz megkonstruálható egy vele ekvivalens $G' = (N', T', P', S')$ környezetfüggetlen nyelvtan úgy, hogy P' -ben nincs $A \rightarrow \varepsilon$ alakú szabály, kivéve ha $\varepsilon \in L(G)$, mert akkor $S' \rightarrow \varepsilon \in P'$, de ekkor S' nem szerepelhet szabály jobb oldalán.*

Formális(abb)an:

$$\forall G = (N, T, P, S) \in \mathcal{G}_2, \exists G' = (N', T', P', S') \in \mathcal{G}_2, L(G') = L(G) :$$

- $\varepsilon \notin L(G) \implies$ nincs olyan szabály P' -ben, amely „ $A \rightarrow \varepsilon$ ” alakú lenne,
- $\varepsilon \in L(G) \implies S' \rightarrow \varepsilon \in P'$, de ekkor S' nem szerepelhet más szabály jobb oldalán.

Bizonyítás. A bizonyítás több lépésből áll.

1.) Határozzuk meg, hogy mely nemterminálisokból vezethető le az üres szó!

$$H := \left\{ A \in N \mid A \xrightarrow{*}_G \varepsilon \right\}.$$

Ehhez definiáljuk az alábbi H_i halmazokat ($i \geq 1$):

$$\begin{aligned} H_1 &:= \{ A \in N \mid \exists A \rightarrow \varepsilon \in P \}, \\ H_{i+1} &:= H_i \cup \{ A \in N \mid \exists A \rightarrow w \in P \wedge w \in H_i^* \}. \end{aligned}$$

Ebből nyilvánvalóan teljesül a következő összefüggés:

$$H_1 \subseteq H_2 \subseteq \dots \subseteq H_i \subseteq H_{i+1}.$$

Mivel $\forall i \geq 1 : H_i \subseteq N$ és N véges halmaz, ezért egy $k \in \mathbb{N}$ indextől kezdődően biztosan azonosak lesznek a halmazok, azaz

$$\exists k \in \mathbb{N}, \forall i \in \mathbb{N} : H_k = H_{k+i}.$$

Így legyen $H := H_k$.

(Megjegyzés. Ennek a részletesebb belátása az [1.] jegyzet 16. oldalán olvasható.)

Ekkor látható, hogy

$$A \in N \wedge A \xrightarrow[G]{*} \varepsilon \iff A \in H.$$

Ennek következménye, hogy

$$\varepsilon \in L(G) \iff S \in H.$$

2.) Alakítsuk át H ismeretében a grammatika szabályait a kellő alakúra.

- (a) $\boxed{S \notin H}$: $A \longrightarrow v' \in P'$ akkor és csak akkor, ha $v' \neq \varepsilon$ és $\exists A \longrightarrow v \in P$ úgy, hogy v' -t a v -ből úgy kapjuk meg, hogy elhagyunk nulla vagy több H -beli nemterminálist v -ből.
- (b) $\boxed{S \in H}$: A korábbi szabályhoz hozzávesszük még a következő két szabályt:

$$S' \longrightarrow \varepsilon \mid S$$

ahol $S' \notin N$ és S' a G' nyelvtan új startszimbóluma. \square

Megjegyzés. A tételt ugyan 2-es típusú nyelvtanokra mondtuk ki, de 3-as típusúakra is tökéletesen működik.

2.4.2. Nyelvek normálformája

A négy nyelvtani típusból háromnak létezik ún. **normálformája**. Ezek a normálformák olyan alakra hozzák az adott típusú nyelvtanok szabályait, melyek egyrészt könnyebben felismerhetővé, egyértelműbbé teszik a típusát, másrészt ez az alak nagy segítségünkre válik, amikor az automatákkal is elkezdünk foglalkozni. Eme ekvivalens transzformációkra gondolhatunk úgy, mint amikor egy egyenletet rendezünk át: a végeredmény nem változik, csupán az alakja. Hasonlóan, a normálformára hozott nyelvtanok az eredeti nyelvet generálják. A tantárgy keretein belül a 2-es és 3-as típusú grammatikák normálformáját fogjuk részletesebben tárgyalni, ugyanis ezeket tudjuk hasznosítani fordítóprogramok írásánál.

Az egyes **nyelvtani típusok normálformái** a következők.

1. típus. **Kuroda-normálforma** (*nem foglalkozunk vele*).

2. típus. **Chomsky-normálforma** és a **Greibach-normálforma**.

3. típus. **3-as típusú nyelvek normálformája**.

Az algoritmusokat a későbbi alfejezetekben részletezzük.

2.5. Automaták

Formális nyelvtannal nyelvet szabályrendszerrel, azaz **generatív módon** adhatunk meg. Ez a megközelítés abból a szemszögből közelíti meg a nyelv szavait, hogy milyen „törvényszerűségekkel” lehet őket levezetni.

Azonban a gyakorlatban számtalanszor van arra szükségünk, hogy *adott szóról kell eldöntünk, hogy a nyelvnek része-e*. Ezt eldönteni pusztán a produkciós szabályokkal nem mindig könnyű eldönteni. Jó lenne, ha úgymond „*automatizálhatnánk*” ezen kérdéskörnek a vizsgálatát. Egy olyan konstrukcióra, eszközre van szükségünk, ami egy „igen” vagy „nem” válasszal visszatérve eldönti, hogy a bemeneti sztring része-e a nyelvnek.

Pontosan erre a célra hozták létre az automatákat. Az **automaták** betűről betűre megvizsgálják, hogy valid-e a nyelv szabályrendszere szerint az *inputszalagon* beolvasott szó és az eredménnyel visszatérnek. Más szóval **akceptív módon** határozza meg a nyelv szavait, így beszélhetünk automata által generált nyelvről.

Mindegyik típushoz tartozik, tartoznak bizonyos típusú automaták. Róluk a megfelelő nyelvtani típusokat feldolgozó fejezetekben lesz bővebben szó. Emellett, mivel a grammatikák és az automaták annyira szorosan kapcsolódnak egymáshoz, bizonyos típusokra léteznek *algoritmusok*, melyekkel *grammatikát automatává lehet konvertálni és fordítva*.

2.6. Zártági tételek

Emlékeztető. Reguláris műveleteknek neveztük az alábbi, nyelvek felett értelmezett műveleteket: *unió, konkatenáció, lezárás*.

Legyen φ egy n -változós nyelvi művelet, azaz ha L_1, \dots, L_n nyelvek, akkor $\varphi(L_1, \dots, L_n)$ is nyelv.

2.6.1. definíció (Nyelvcsalád zártsága műveletre nézve). Az \mathcal{L} nyelvcsalád zárt a φ műveletre nézve, ha $L_1, \dots, L_n \in \mathcal{L}$ esetén $\varphi(L_1, \dots, L_n) \in \mathcal{L}$.

2.6.1. tétel. Az \mathcal{L}_i ($i = 0, 1, 2, 3$) nyelvcsaládok mindegyike zárt a reguláris műveletekre nézve.

Bizonyítás. Műveletenként. Az unió kivételével mindegyiknél csak $i = 3$ -ra látjuk be – a mi szempontunkból ennyi bőven elég.

(Megjegyzés. A többi nyelvcsaládra a régi jegyzet 1.9. fejezetében található részletesebb leírás (27. oldal).)

Legyen $G = (N, T, P, S)$ az L nyelvhez tartozó grammatika, $G' = (N', T, P', S')$ legyen az L' -hez tartozó grammatika, valamint teljesüljön, hogy $N \cap N' = \emptyset$ és G, G' azonos típusúak.

A) Unió: Vezessünk be egy új startszimbólumot! Az alapkonstrukciónk:

$$G_{\cup} := (N \cup N' \cup \{S_{\cup}\}, T, P \cup P' \cup \{S_{\cup} \rightarrow \text{új szabály jobb oldala}\}, S_{\cup}).$$

(a) $i = 0, 2, 3$: Legyen S_0 az új startszimbólum ($S_0 \notin (N \cup N')$). Az alábbi

alapkonstrukcióval fogunk dolgozni.

$$G_{\cup} := (N \cup N' \cup \{S_0\}, T, P \cup P' \cup \{S_0 \rightarrow S \mid S'\}, S_0).$$

Látható, hogy G_{\cup} típusa megegyezik G és G' típusával, és $L(G) \cup L(G') = L(G_{\cup})$. Röviden, nem kell attól tartanunk, hogy az ε -szabály elveszne.

- (b) $\boxed{i=1}$: Ebben az esetben már elveszhet az ε , ha $\varepsilon \in (L \cup L')$. Ekkor az előbbi módon elkészített grammatikában nem teljesül a KES.

Tekintsük az $L_1 := L \setminus L_{\varepsilon}$ és $L_2 := L' \setminus L_{\varepsilon}$ nyelveket, melyeket rendre G_1 és G_2 nyelvtanok generálnak (melyek 1-es típusúak).

Készítsük el G_{\cup} -t az előbbi módon, majd vezessünk be egy S_1 új kezdőszimbólumot és adjuk a szabályhalmazhoz az

$$S_1 \rightarrow \varepsilon \mid S_0$$

szabályokat (ez két szabály, tömörítve felírva).

- B) Konkatenáció: Csak $i = 3$ -ra.

A P szabályhalmazból megkonstruálunk egy P_1 szabályhalmazt úgy, hogy minden $A \rightarrow u$ alakú szabályt felcserélünk egy $A \rightarrow uS'$ alakú szabályra, a többi szabályt változatlanul hagyjuk.

Ekkor a

$$G_C := (N \cup N', T, P_1 \cup P', S)$$

grammatika 3-as típusú és generálja az $L(G)L(G')$ nyelvet.

- C) Lezárás: Csak $i = 3$ -ra.

Legyen S_0 új szimbólum, azaz $S_0 \notin N$. Definiáljuk a P_1 szabályhalmazt úgy, hogy minden $A \rightarrow u$ alakú szabályt felcserélünk egy $A \rightarrow uS_0$ alakú szabályra és ezek legyenek a P_1 elemei. Ekkor a

$$G_* := (N \cup \{S_0\}, T, P_1 \cup P \cup \{S_0 \rightarrow \varepsilon \mid S\}, S_0)$$

grammatika generálja az L^* nyelvet. \square

3. fejezet

Reguláris (3-as típusú) nyelvtanok

3.1. Reguláris nyelvek

A 3-as nyelvcsalád nyelveit az alábbi módokon írhatjuk le:

- 3-as típusú grammatikával,
- reguláris kifejezéssel,
- véges determinisztikus automatával (VDA),
- véges nemdeterminisztikus automatával (VNDA).

Megjegyzés. A programozási nyelvek lexikális egységei a 3-as nyelvcsaládba tartoznak.

3.1.1. állítás.

$$\mathcal{L}_3 = \mathcal{L}_{reg} = \mathcal{L}_{VDA} = \mathcal{L}_{VNDA}.$$

Bebizonyítható az állítás. A régi jegyzetben több tétel következményeként meggondolható. Emellett a későbbiekben be is fogjuk látni.

3.1.1. definíció (Reguláris nyelvek).

- az **elemi nyelvek**: \emptyset , $\{\varepsilon\}$, $\{a\}$, ahol $a \in U$, azaz egy tetszőleges betű
- azon nyelvek, melyek az elemi nyelvekből az **unió**, a **konkatenáció** és a **lezárás** műveletek **véges számú alkalmazásával** állnak elő;
- nincs más reguláris nyelv

Példa. $\{\{a\} \cup \{b\}\}^* \{b\} = \{ub \mid u \in \{a, b\}^*\}$.

3.1.1. tétel. Minden L reguláris nyelvhez megadható egy $G \in \mathcal{G}_3$ 3-as típusú grammatika, amelyre $L = L(G)$. ($\mathcal{L}_{reg} \subseteq \mathcal{L}_3$)

Bizonyítás. Az elemi nyelvekhez adhatunk 3-as típusú nyelvtanokat.

- $G = (\{S\}, \{a\}, \{S \rightarrow aS\}, S) \quad L(G) = \emptyset.$
- $G = (\{S\}, \{a\}, \{S \rightarrow \varepsilon\}, S) \quad L(G) = \{\varepsilon\}.$
- $G = (\{S\}, \{a\}, \{S \rightarrow a\}, S) \quad L(G) = \{a\}.$

Korábban láttuk, hogy az \mathcal{L}_3 nyelvcsalád zárt a reguláris műveletekre nézve. Az elemi nyelvek grammatikáiból kiindulva megkonstruálható a reguláris műveletekhez tartozó grammatika konstrukciókkal a megfelelő 3-as típusú grammatika bármely összetett reguláris nyelvhez. \square

3.1.2. definíció (Reguláris kifejezés).

- az *elemi reguláris kifejezések*: $\emptyset, \varepsilon, a \quad (a \in U)$
- ha R_1 és R_2 és R reguláris kifejezések akkor
 - i) $(R_1 | R_2)$;
 - ii) $(R_1 R_2)$;
 - iii) $(R)^*$ is *reguláris kifejezések*.
- a *reguláris kifejezések halmaza* a legszűkebb halmaz, melyre a fenti két pont teljesül.

Vigyázat! A reguláris kifejezések önmagukban nem reguláris nyelvek, azaz a reguláris kifejezés nem ugyanaz, mint a reguláris nyelv. Jelölésben az alábbi módon különböztetjük meg:

L_R jelöli az R reguláris kifejezéshez tartozó nyelvet.

Az elemi nyelvekre kiterjesztve:

$$\begin{aligned} L_{\emptyset} &= \emptyset, \\ L_{\varepsilon} &= \{\varepsilon\}, \\ L_a &= \{a\} \quad (a \in U). \end{aligned}$$

Valamint, ha Q és R reguláris kifejezések, akkor:

$$\begin{aligned} L_{(Q|R)} &= L_Q \cup L_R \\ L_{(QR)} &= L_Q L_R \\ L_{(R)^*} &= (L_R)^* \end{aligned}$$

A gyakorlatban sokszor nem számít ez a különbségtétel, ezért előfordulhat, hogy a jegyzetben a világosság érdekében, de a pontosság rovására ez a „szintaktikai cukormáz” fogja jelenteni a nyelvet.

A műveletek **prioritási sorrendje** növekvően:

unió < konkatenáció < lezárás.

A zárójelek elhagyhatók a reguláris kifejezésekből a prioritásoknak megfelelően.

3.2. 3-as típusú nyelvtanok normálformája

Ahogy korábban bevezettük, a nyelvtanok típusaihoz léteznek ún. **normálformák**, amelyekre gondolhatunk úgy, mint speciális formára hozott nyelvtanok, melyek ekvivalensek az eredetivel. Ezek sokszor megkönnyítik a nyelvtan vizsgálatát.

A 3-as típusú nyelvtanok normálformája az alábbi alakokkal rendelkeznek.

3.2.1. tétel. Minden 3-as típusú nyelv generálható olyan grammatikával, amelynek szabályai az alábbi alakokat ölthetik fel:

- $A \rightarrow aB$, ahol $A, B \in N$ és $a \in T$ (egyetlen szimbólum),
- $A \rightarrow \varepsilon$, ahol $A \in N$.

A normálformát a 3-as típusú nyelvtanok esetében azért szeretjük, mert **könnyű belőle automatát készíteni**. Az, hogy a normálformára hozott nyelvtanból hogyan tudunk automatát előállítani, azt a későbbiekben tárgyaljuk. Egyelőre megnézzük azt az **algoritmust**, mellyel **normálformára hozhatunk 3-as típusú nyelvtanokat**.

A 3-as normálformára hozás algoritmus 3 lépésből áll.

I. Hosszredukció

Elhagyjuk az $A \rightarrow a_1 \dots a_k B$ alakú szabályokat, ahol $k \geq 2$ és

$$\forall i \in [1..k] : a_i \in T,$$

valamint teljesül, hogy $A \in N$ és $B \in N \cup \{\varepsilon\}$. Tehát a jobb oldalon nem szükséges, hogy nemterminális szimbólum is szerepeljen.

Helyettesítsük a következő szabályokkal:

$$\begin{aligned} A &\rightarrow a_1 Z_1, & \text{ahol } Z_1 \notin N \rightarrow \text{új terminális} \\ Z_1 &\rightarrow a_2 Z_2, & \text{ahol } Z_2 \notin (N \cup \{Z_1\}) \\ Z_2 &\rightarrow a_3 Z_3, & \text{ahol } Z_3 \notin (N \cup \{Z_1, Z_2\}) \\ &\dots \\ Z_{k-1} &\rightarrow a_k B \end{aligned}$$

Vagyis minden szabályra új nemterminálisokat vezetünk be. Azért hívjuk hosszredukciónak ezt a lépést, mert a szabály jobb oldalának $a_1 \dots a_k \in T^k \subset T^*$ „szeletéből” olyan szabályokat hozunk létre, melyek már $a_i \in T$ ($i \in [1..k]$) terminálisokat tartalmaznak.

II. Befejező szabályok átalakítása

Elhagyjuk az $A \rightarrow a$ alakú szabályokat^a, ahol $a \in T$ és $A \in B$. Ehhez felvesszünk egy új nemterminálist (jelöljük E -vel), ami lehet közös minden befejező szabály esetén.

Innen az alábbi új szabályokat felvesszük a transzformált nyelvtanunkba:

$$A \rightarrow aE \quad \text{és} \quad E \rightarrow \varepsilon.$$

III. Lánementesítés

Elhagyjuk az $A \rightarrow B$ alakú szabályokat, ahol $A, B \in N$. Más szóval, csak nemterminális áll a szabály jobb oldalán.

Első lépésben meghatározzuk minden $A \in B$ esetén a

$$H(A) := \left\{ B \in N \mid A \xRightarrow[G]{*} B \right\}$$

halmazokat. Ehhez iteratívan definiáljuk a H_i halmazokat ($i \geq 1$):

$$\begin{aligned} H_1(A) &:= \{A\}, \\ H_{i+1}(A) &:= H_i(A) \cup \{B \in N \mid \exists C \in H_i(A) \wedge C \rightarrow B \in P\}. \end{aligned}$$

Ha elkészültünk a halmazokkal, azt mondhatjuk, hogy

$$\exists k \in \mathbb{N}^+ : H_1(A) \subseteq H_2(A) \subseteq \dots \subseteq H_k(A) = H_{k+1}(A).$$

Ekkor legyen $H(A) := H_k(A)$.

Ezután felvesszük a transzformált nyelvtenba az $A \rightarrow X$ szabályokat, ha

$\exists B \in H(A), B \rightarrow X \in P$, ahol $X \in (N \cup T)^*$ és X nem csak egyetlen terminális.

^aEzeket *befejező szabályoknak* nevezzük.

3.3. Véges automaták

3.3.1. definíció. *Véges determinisztikus automátának* nevezzük az

$$A = (Q, T, \delta, q_0, F)$$

rendezett ötöst, ahol

- Q az **állapotok halmaza** ($0 < |Q| < \infty$),
- T a **bemeneti szimbólumok ábécéje**,
- $\delta : Q \times T \rightarrow Q$ **leképezés az állapot-átmeneti függvény**,
- $q_0 \in Q$ a **kezdeti állapot**,
- $F \subseteq Q$ az **elfogadóállapotok halmaza** (vagy **végállapotok halmaza**)

Megjegyzés. Fontos, hogy *véges determinisztikus automata* esetén a δ függvény értelmezett minden $(q, a) \in Q \times T$ párra, azaz

$$\forall (q, a) \in Q \times T, \exists! p \in Q : \delta(q, a) = p.$$

Ha ez nem teljesül, azaz egy $(q, a) \in Q \times T$ párhoz több $p \in Q$ állapot is tartozhat, akkor véges **nem**determinisztikus automatáról beszélünk (*VNDA* vagy *NDA*).

3.3.2. definíció. *Véges nemdeterminisztikus automatának nevezzük az*

$$A = (Q, T, \delta, Q_0, F)$$

rendezett ötöst, ahol

- Q az **állapotok halmaza** ($0 < |Q| < \infty$),
- T a **bemeneti szimbólumok ábécéje**,
- $\delta : Q \times T \rightarrow \mathcal{P}(Q)$ leképezés az **állapot-átmeneti függvény**,
- $Q_0 \subseteq Q$ a **kezdőállapotok halmaza**,
- $F \subseteq Q$ az **elfogadóállapotok halmaza** (vagy **végállapotok halmaza**)

Felhívjuk a figyelmet arra a pár apró, ugyan lényeges különbségre, ami ebben a definícióban található.

- Egyrészt, a egyetlen kezdőállapot helyett kezdőállapotok halmazáról beszélünk.
- Az állapot-átmenetek függvénye a Q hatványhalmazába képez (amit $\mathcal{P}(Q)$ -val jelölünk). Ez engedi meg, hogy egy adott (q, a) párhoz több állapotot is hozzárendelhesünk.

A VNDA a VDA általánosításának tekinthető.

Hogy kövessük az eddig bevezetett konvenciókat, az állapot-átmeneteket is felírhatjuk olyan szintaxissal, amellyel a produkciós szabályokat írtuk fel.

$$\delta(q, a) = p \iff qa \longrightarrow p.$$

Az automaták témakörében is értelmezzük a *mondatformának* megfeleltethető fogalmat, amit **konfigurációnak** nevezünk.

3.3.3. definíció (Konfiguráció). *A $v \in QT^*$ egy konfigurációja egy VDA-nak, ha az aktuális állapotot és az inoput hátralévő részét tartalmazza, azaz $v = qu$.*

Hasonlóan, a *közvetlen és közvetett levezetésnek* is létezik megfelelője. Ezeket **közvetlen**, ill. **közvetett redukciónak** nevezzük. A redukció név arra utal, hogy az inputszalagról beolvasott szöveg hossza egyre csökken.

3.3.4. definíció (Közvetlen redukció). *Legyen $A = (Q, T, \delta, q_0, F)$ egy VDA és legyenek $u, v \in Q^*$ konfigurációk.*

*Azt mondjuk, hogy az A automata az u konfigurációt a v konfigurációra **redukálja közvetlenül**, ha*

$$\exists \delta(q, a) = p \text{ szabály} \wedge \exists w \in T^* : u = qaw \wedge v = pw.$$

Jele: $u \xRightarrow{A} v$.

Alternatív módon: van olyan $qa \longrightarrow p$ szabály és van olyan $w \in T^*$ szó, amelyre

$$u = \mathbf{q}aw \wedge v = \mathbf{p}w.$$

A vastag kijelölés nem azt jelenti, hogy vektorok lennének, hanem hogy szemléletesebben kiemeljem a „csere” helyét.

A **közvetett redukciót** gyakran csak egyszerűen **redukciónak** nevezzük.

3.3.5. definíció (Redukció vagy közvetett redukció). Az $A = (Q, T, \delta, q_0, F)$ véges automata az $u \in QT^*$ konfigurációt a $v \in QT^*$ konfigurációra **redukálja**, ha

- ha $u = v$, vagy
- ha $u \neq v$, akkor $\exists z \in QT^* : u \xrightarrow[A]{*} z \wedge z \xrightarrow[A]{*} v$.

Jele: $u \xrightarrow[A]{*} v$.

Ahogy a grammatikáknál is, itt is értelmezzük az automata által elfogadott nyelvet. Figyeljük meg a szóhasználatot: az automata továbbra sem generálja, hanem elfogadja a szavakat (akceptív módon közelíti meg a szóproblémát).

3.3.6. definíció (Automata által elfogadott nyelv). Az $A = (Q, T, \delta, q_0, F)$ véges automata által elfogadott nyelv alatt az

$$L(A) := \left\{ u \in T^* \mid q_0 u \xrightarrow[A]{*} p \wedge p \in F \right\}$$

szavak halmazát értjük.

A definíció azt jelenti, hogy az automata a kezdőállapotból (q_0 -ból) indulva végig olvasva az inputot elfogadóállapotba jut (azaz $p \in F$).

3.3.1. 3-as típusú nyelvek kapcsolata a véges automatákkal

3.3.1. tétel. Minden 3-as típusú L nyelvhez megadható egy véges nemdeterminisztikus automata, és fordítva; minden nemdeterminisztikus automata 3-as típusú nyelvet ismer fel.

$$\mathcal{L}_3 \subseteq \mathcal{L}_{VNDA} \quad \text{és} \quad \mathcal{L}_{VNDA} \subseteq \mathcal{L}_3$$

Bizonyítás. // Kidolgozni.

3.3.2. tétel. Minden $A = (Q, T, \delta, Q_0, F)$ nemdeterminisztikus automatához megadható egy $A' = (Q', T, \delta', q'_0, F')$ véges determinisztikus automata, hogy az általuk generált nyelvek ekvivalensek, azaz

$$\forall A = (Q, T, \delta, Q_0, F), \exists A' = (Q', T, \delta', q'_0, F') : L(A') = L(A).$$

$$\mathcal{L}_{VNDA} \subseteq \mathcal{L}_{VDA}$$

Bizonyítás. // Kidolgozni.

3.3.3. tétel (Kleene tétele).

$$\mathcal{L}_3 = \mathcal{L}_{reg}$$

Bizonyítás. // Kidolgozni.

3.3.2. Minimális véges determinisztikus automata

3.3.7. definíció (Minimális véges determinisztikus automata). Az A véges determinisztikus automata **minimális állapotszámú**, ha nincs olyan A' véges determinisztikus automata, amely ugyanazt a nyelvet ismeri fel, mint A , de A' állapotainak száma kisebb, mint A állapotainak száma.

$$\exists A' = (Q', T, \delta', q'_0, F') \text{ véges det. autom. : } L(A') = L(A) \wedge |Q'| < |Q|$$

3.3.4. tétel. Az L reguláris nyelvet felismerő **minimális** véges determinisztikus automata az izomorfizmus erejéig **egyértelmű**.

Bizonyítás. // Kidolgozni.

3.3.8. definíció (Elérhető állapot). Az $A = (Q, T, \delta, q_0, F)$ véges determinisztikus automata q **állapotát elérhetőnek** mondjuk, ha

$$\exists u \in T^* : q_0 u \xrightarrow[A]{*} q.$$

3.3.9. definíció (Összefüggő VDA). Az $A = (Q, T, \delta, q_0, F)$ véges determinisztikus automatát **összefüggőnek** mondjuk, ha minden állapota elérhető a kezdőállapotból.

3.3.10. definíció (Ekvivalens állapotok). Legyen $A = (Q, T, \delta, q_0, F)$ egy VDA és $q, p \in Q$ állapotok. Ekkor q és p **ekvivalens állapotok**, ha

$$\forall u \in T^* \text{ szóra teljesül, hogy } qu \xrightarrow[A]{*} r \text{ és } pu \xrightarrow[A]{*} r' \text{ esetén} \\ r \in F \text{ akkor és csak akkor, ha } r' \in F.$$

Jele: $\boxed{q \sim p}$.

3.3.1. állítás. Ha q és p ekvivalens, akkor $qa \rightarrow s$ és $pa \rightarrow t$ esetén s és t is ekvivalens állapotok $\forall a \in T$ betűre.

3.3.11. definíció (i -ekvivalens állapotok). Legyen $A = (Q, T, \delta, q_0, F)$ egy VDA és $q, p \in Q$ állapotok. Az mondjuk, hogy q és p **i -ekvivalens állapotok**, ha

$$\forall u \in T^* \text{ szóra, ahol } \ell(u) \leq i \text{ teljesül, hogy}$$

$$qu \xrightarrow[A]{*} r \text{ és } pu \xrightarrow[A]{*} r' \text{ esetén } r \in F \text{ akkor és csak akkor, ha } r' \in F.$$

Jele: $\boxed{q \sim^i p}$ vagy $\boxed{q \overset{i}{\sim} p}$.

3.3.1. lemma.

$$q \overset{i+1}{\sim} p \iff \forall a \in T, qa \longrightarrow s \wedge pa \longrightarrow t : s \overset{i}{\sim} t.$$

Szavakban: legfeljebb i hosszú szavak esetén a két állapot nem megkülönböztethető.

4. fejezet

A 2-es és 3-as nyelvcsalád viszonya

A következő tételek szükséges feltételeket fogalmaznak meg a 3-as típusú nyelvekre. Vannak nyelvek, amelyek bizonyíthatóan nem teljesítik a feltételeket, de 2-es típusú grammatikával generálhatók.

4.1. Szükséges feltétel 3-as típusú nyelvekre

4.1.1. lemma (Kis Bar-Hiller lemma). $\forall L \in \mathcal{L}_3$ nyelvhez $\exists n \in \mathbb{N}^+$ *nyelvfüggetlen konstans*, hogy $\forall u \in L$, ahol $\ell(u) \geq n$ szó esetén van u -nak olyan $u = xyz$ felbontása, amelyre

- $\ell(xy) \leq n$,
- $y \neq \varepsilon$,
- $\forall i \in \mathbb{N} : xy^i z \in L$.

Bizonyítás. // Kidolgozni.

4.2. Szükséges és elégséges feltétel 3-as típusú nyelvekre

4.2.1. definíció (Maradéknyelv). Legyen L egy T ábácé felett értelmezett nyelv ($L \subseteq T^*$). Az L nyelv egy $p \in T^*$ szóra értelmezett *maradéknyelve* a következő:

$$L_p := \{u \in T^* \mid pu \in L\}.$$

4.2.1. tétel (Myhill–Nerode-tétel). Egy L nyelv akkor és csak akkor 3-as típusú, ha a véges számú maradéknyelve van, azaz

$$L \in \mathcal{L}_3 \iff |\{L_p \mid p \in T^*\}| < \infty.$$

Megjegyzés. A szavakon egy osztályozást végzünk az adott nyelvtől függően.

Bizonyítás. // Kidolgozni.

5. fejezet

Környezetfüggetlen (2-es típusú) nyelvtanok

5.1. A szóprobléma kérdése

A *formális nyelvek* témakörében az egyik központi kérdésünk, hogy adott G grammatika és adott $u \in T^*$ szó esetén teljesül-e, hogy $u \in L(G)$. Vagyis, hogy a G nyelvtan által generált nyelvben benne van-e az u szó. Szerencsére, a 2-es típusú nyelvtanok esetében vannak eszközeink ezen kérdésnek eldöntésére.

Ez az úgynevezett **szintaxisfa**, vagy **levezetési fa**. Az elnevezés több értelmet nyer, ha felidézzük a Backus–Naur-jelölést.

5.1.1. definíció (Szintaxisfa). Legyen $G = (N, T, P, S) \in \mathcal{G}_2$ grammatika. A t nemüres fát G feletti **levezetési (szintaxis) fának** nevezzük, ha

- pontjai $T \cup N \cup \{\varepsilon\}$ elemeivel vannak címkézve;
- belső pontjai N elemeivel vannak címkézve;
- ha egy belső pont címkéje A , a közvetlen leszármazottjainak címkéi pedig balról jobbra olvasva X_1, X_2, \dots, X_k , akkor $A \rightarrow X_1 X_2 \dots X_k \in P$.
- az ε -nal címkézett pontoknak nincs testvére.

5.1.1. tétel. Ha adott G grammatika esetén $u \in L(G)$ akkor és csak akkor, ha u -hoz megadható egy szintaxisfa.

Megjegyzés. Az u -hoz tartozó szintaxisfa gyökere S és a leveleit balról jobbra összeolvasva az u szót kapjuk.

5.1.1. állítás. Minden szintaxisfához megadható egy levezetés és fordítva.

5.1.2. definíció (Egyértelmű nyelvtan). Egy $G \in \mathcal{G}_2$ nyelvtan **egyértelmű**, ha minden $u \in L(G)$ szóhoz egyetlen szintaxisfa tartozik.

5.2. 2-es típusú nyelvtanok normálformája

A Chomsky-hierarchia bevezetésénél kimondtuk az ε -mentesítés tételét, amit 2-es és 3-as típusú nyelvtanokon elvégezhető transzformáció.

5.2.1. definíció (Chomsky-normálforma). Egy $G = (N, T, P, S) \in \mathcal{G}_2$ nyelvtant *Chomsky-normálformájúnak* mondunk, ha szabályai

- $A \rightarrow a$, ahol $A \in N$ és $a \in T$ vagy
- $A \rightarrow BC$ alakúak, ahol $A, B, C \in N$.
- $S \rightarrow \varepsilon$, de ekkor S nem fordul elő egyetlen szabály jobboldalán sem.

5.2.1. tétel. Minden környezetfüggetlen grammatikához megkonstruálható egy vele ekvivalens *Chomsky normálformájú* grammatika.

Megjegyzések.

- A 2-es típusú grammatikák Chomsky normálformára hozásának algoritmusát nem a tananyag része.¹
- Chomsky normálformájú grammatikákhoz megadható olyan elemző program, amely $O(n^3)$ időben eldönti a szóproblémát (*Cocke–Younger–Kasami-algoritmus*).
- Bizonyos állítások bizonyítását elég elvégezni a normálformájú grammatikákra.

5.3. Nyelvtan redukálása

A grammatikák transzformálása közben keletkezhetnek olyan szabályok, amelyek egyetlen szó levezetésében sem használhatóak.

A grammatikában lehetnek olyan nemterminálisok, amelyekből

1. nem lehet csupa nem terminálisból álló sorozatot előállítani (*zsákutcák*);
2. nem érhető el a kezdőszimbólumból.

5.3.1. definíció (Aktív nemterminálisok). *Aktív nemterminálisok* halmaza egy adott $G = (N, T, P, S) \in \mathcal{G}_2$ grammatika esetén:

$$A := \left\{ X \in N \mid X \xRightarrow{*}_G u \wedge u \in T^* \right\}.$$

Inaktív (*zsákutca*) nemterminálisok: $N \setminus A$.

5.3.2. definíció (Elérhető nemterminálisok). *Elérhető nemterminálisok* halmaza egy adott $G = (N, T, P, S) \in \mathcal{G}_2$ grammatika esetén:

$$R := \left\{ X \in N \mid S \xRightarrow{*}_G uXw \wedge u, w \in (T \cup N)^* \right\}.$$

¹A jegyzet írása idején így szólt a tanterv (2023/2024/2. félév).

Nem elérhető nemterminálisok: $\boxed{N \setminus R}$.

5.3.3. definíció (Hasznos nemterminálisok). Egy nemterminálíst *hasznosnak* mondunk, ha *aktív és elérhető*:

$$X \in (A \cap R) \subseteq N.$$

5.3.4. definíció (Redukált nyelvtan). Egy környezetfüggetlen grammatika *redukált*, ha minden nemterminálisa *hasznos*, azaz a grammatika *zsákutcamentes és összefüggő*.

5.3.1. tétel. Minden $G \in \mathcal{G}_2$ nyelvtanhoz megkonstruálható egy vele ekvivalens *redukált grammatika*.

Bizonyítás. // Kidolgozni.

1. Zsákutcák meghatározása és minden olyan szabály elhagyása, amiben inaktív nemterminálisok szerepelnek.
2. Az S -ből nem elérhető nemterminálisokhoz tartozó szabályok elhagyása, azaz a grammatika összefüggővé tétele.

5.4. Veremautomaták

Elöljáróban elárultuk, hogy a szóprobléma eldönthető az összes környezetfüggetlen nyelvtan által generált nyelvre. Íme az erre vonatkozó tétel és bizonyítása.

5.4.1. tétel (Szóprobléma eldöntése). Minden $G = (N, T, P, S) \in \mathcal{G}_2$ grammatika esetében eldönthető, hogy egytetszőleges $u \in T^*$ szó benne van-e a G grammatika által generált nyelvben vagy sem.

Bizonyítás. // Kidolgozni.

A szóprobléma eldönthető a környezetfüggetlen grammatikákhoz párosuló **veremautomaták**kal, melyet az alábbi módon definiálunk.

5.4.1. definíció (Veremautomata). *Veremautomatának nevezzük az*

$$A = (Z, Q, T, \delta, z_0, q_0, F)$$

rendezett hetest, ahol

- Z a verem szimbólumainak ábécéje,
- Q az állapotok halmaza ($0 < |Q| < \infty$),
- T a bemeneti szimbólumok ábécéje,
- $\delta : Z \times Q \times (T \cup \{\varepsilon\}) \rightarrow \mathcal{P}(Z^* \times Q)$ leképezés az állapotátmeneti függvény, ahol δ végs részhalmozokba képez,
- $z_0 \in Z$ a kezdő veremszimbólum,
- $q_0 \in Q$ a kezdőállapot
- $F \subseteq Q$ az elfogadóállapotok halmaza.

Egy lépésben mindig kell egy jelet olvasni a verem tetejéről és csak egy jelet lehet elérni. Az input szalagról is egy jelet lehet olvasni, de nem kötelező.

Megváltoztatható az automata aktuális állapota, illetve a verem teteje. Egy lépésben egy egész sorozatot is beírhatunk a verembe.

Példák.

- $\delta(\#, q, a) = \{(\#a, q)\}$
Jelentése: Ha $\#$ van a verem tetején és a betű jön az inputon, akkor **tegyük be** a -t a verembe. Ne változtassunk az állapoton.
- $\delta(\#, q, a) = \{(\varepsilon, q)\}$
Jelentése: Ha $\#$ van a verem tetején és a betű jön az inputon, akkor **töröljük** $\#$ -t a veremből. Ne változtassunk az állapoton.
- $\delta(\#, q, a) = \{(\#, r)\}$
Jelentése: Ha $\#$ van a verem tetején és a betű jön az inputon, akkor **ne változtassuk a verem tartalmát**. Viszont váltsunk állapotot.
- $\delta(\#, q, \varepsilon) = \{(\#bb, q)\}$
Jelentése: Ha $\#$ van a verem tetején és nem olvasunk az inputról, akkor **tegyünk a verembe** két b betűt és váltsunk állapotot is.

Ahogy az a véges automatáknál is tapasztalhattuk, létezik az állapotátmeneteknek egy alternatív jelölése, mely követi azt a konvenciót, ahogyan a nyelvtani szabályokat írjuk fel.

- Ha $\delta(z, q, a) = \{(w_1, r_1), \dots, (w_k, r_k)\}$ ($k \in \mathbb{N}^+$), akkor ezt a leképezést a következő szabályhalmazzal is jelölhetjük:

$$zqa \longrightarrow w_i r_i \quad i \in [1 \dots k].$$

- Ha $\delta(z, q, \varepsilon) = \{(w_1, r_1), \dots, (w_k, r_k)\}$ ($k \in \mathbb{N}^+$), akkor ezt a leképezést a következő szabályhalmazzal is jelölhetjük:

$$zq \longrightarrow w_i r_i \quad i \in [1 \dots k].$$

Tehát a szabályok bal oldala ZQT vagy ZQ alakú és a jobboldala Z^*Q alakú.

Ahogy a véges automatáknál, itt is értelmezzük a **konfiguráció** fogalmát – természetesen a megfelelő módosításokkal.

5.4.2. definíció (Konfiguráció). Legyen $A = (Z, Q, T, \delta, z_0, q_0, F)$ egy veremautomata és legyen $\alpha \in Z^*QT^*$. Azt mondjuk α az A veremautomata egy **konfigurációja**.

Szavakban: a konfiguráció a veremautomata egy pillanatnyi állapotát írja le.

Hasonlóan, a redukciót fogalmát is értelmezzük. Ahogy azt a véges automatáknál is tapasztalhattuk, a „sima” redukció magába magába foglalja a közvetett redukciót, így a kettőt nem szoktuk megkülönböztetni.

5.4.3. definíció (Közvetlen redukció). Legyen $A = (Z, Q, T, \delta, z_0, q_0, F)$ egy veremautomata és legyen $\alpha, \beta \in Z^*QT^*$ konfigurációk. Azt mondjuk, hogy az A veremautomata az α konfigurációt a β konfigurációra **redukálja közvetlenül**, ha

$$\exists z \in Z, p, q \in Q, a \in T \cup \{\varepsilon\}, r, u \in Z^*, w \in T^*, \text{ hogy}$$

- $\boxed{zqa \rightarrow up}$ egy szabály δ -ban,
- $\alpha = rzqaw$ és $\beta = rupw$.

Jele: $\boxed{\alpha \xRightarrow[A]{}} \beta$.

A vastag betűk továbbra is arra szolgálnak, hogy szemléletesebbé váljon a helyettesítés és nem vektorokat jelentenek.

5.4.4. definíció (Redukció). Legyen $A = (Z, Q, T, \delta, z_0, q_0, F)$ egy veremautomata és legyenek $\alpha, \beta \in Z^*QT^*$ konfigurációk. Azt mondjuk, hogy az A veremautomata az α konfigurációt a β konfigurációra **redukálja közvetetten**,

- ha vagy $\alpha = \beta$,
- vagy ha $\alpha \neq \beta$, akkor $\exists \alpha_1 \dots \alpha_k$ ($k \in \mathbb{N}^+$) konfiguráció sorozat, hogy $\alpha_1 = \alpha$ és $\alpha_k = \beta$ és $\forall i \in [1 \dots k-1] : \alpha_i \xRightarrow[A]{*} \alpha_{i+1}$.

Jele: $\boxed{\alpha \xRightarrow[A]{*} \beta}$.

A véges automatákkal szemben itt kétféle „nyelvet” értelmezzünk attól függően, hogy megengedjük-e, hogy a verem lehet-e üres.

5.4.5. definíció (Elfogadó állapottal felismerhető nyelv).

$$L(A) := \left\{ u \in T^* \mid \exists z_0 q_0 u \xRightarrow[A]{*} wr \wedge r \in F \wedge w \in Z^* \right\}$$

Ez azt jelenti, hogy van olyan működése a veremautomatának, hogy kezdő konfigurációból

indulva végig olvasva az inputot elfogadóállapotba jut.

5.4.6. definíció (Üres veremmel felismerhető nyelv).

$$L(A) := \left\{ u \in T^* \mid \exists z_0 q_0 u \xrightarrow[A]{*} r \wedge r \in F \right\}$$

Ez azt jelenti, hogy van olyan működése a veremautomatának, hogy kezdő konfigurációból indulva végig olvasva az inputot teljesen kiüríti a vermet.

A veremautomaták **determinisztikusságát** is vizsgálhatjuk.

5.4.7. definíció (Determinisztikus veremautomata). Egy veremautomatát **determinisztikusnak** mondunk, ha $\forall \alpha \in Z^+QT^*$ konfiguráció esetén egyetlen konfiguráció vezethető le közvetlenül α -ból.

Ez azt jelenti, hogy nincs két olyan szabály, amelynek azonos a bal oldala, valamint, ha zq egy bal oldal, akkor nincs zqa bal oldal egyetlen terminálisra sem.

A determinisztikus veremautomatával felismerhető nyelvek családja szűkebb, mint a nemdeterminisztikussal felismerhető nyelvek családja. Például a szimmetrikus szavak nem ismerhetők fel determinisztikus veremautomatával.

5.4.1. lemma. Bármely A veremautomatához megadható A' veremautomata úgy, hogy $N(A') = L(A)$.

5.4.2. lemma. Bármely A veremautomatához megadható A' veremautomata úgy, hogy $L(A') = N(A)$.

Megjegyzés. Ez azt jelenti, hogy ha egy nyelvhez építhető elfogadó állapottal felismerő veremautomata, akkor építhető üres veremmel felismerhető veremautomata és fordítva.

5.4.2. tétel. Minden $L \in \mathcal{L}_2$ nyelvhez megadható egy A veremautomata úgy, hogy

$$L = N(A), \quad \text{azaz} \quad \mathcal{L}_2 \subseteq \mathcal{L}_{1V}.$$

Bizonyítás. // Kidolgozni.

5.4.3. tétel. Minden A veremautomatához megadható egy $G \in \mathcal{G}_2$ nyelvtan úgy, hogy

$$L(G) = N(A), \quad \text{azaz} \quad \mathcal{L}_{1V} \subseteq \mathcal{L}_2.$$

A fordított tételt nem bizonyítjuk.

6. fejezet

Fordítóprogramok

A jegyzet első felében részletezett elméleti háttérrel felvértezve már képesek vagyunk nyelveket definiálni. A most következő részben betekintést nyerhetünk abba, hogy miként lesz a nyelvünkben egy, a számítógép által értelmezhető és végrehajtható program.

Ennek megvalósításához szükségünk lesz egy **fordítóprogramra** (angolul *compilerre*). A fordítóprogram nem más, mint egy olyan eszköz, ami szöveges bemenetet fogad el (fájl, parancssori bemenet, stb.), ellenőrzi azt, majd

- ha helyes a szövegünk, létrehozza a futtatható programot,
- ha nem, hibát jelez (esetleg megmutatja, *mi a hiba* és az *hol található*).

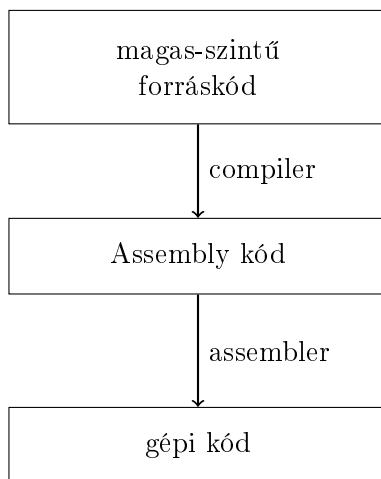
Azt a nyelvet, amit a számítógép beszél, **gépi kódnak** (*machine code*) nevezzük. Ez egy gépközel nyelv (numerikus utasításkódok, regiszterek, memóiahivatkozások, stb.), mely erősen platformfüggő, cserébe jól optimalizált.

Ezen tulajdonságai kényelmetlenné teszik a programírást a **magas(abb) szintű nyelvekkel** ellentétben (*high(er)-level languages*), melyekben könnyebb programozni, a benne megírt kód közelebb a megoldandó problémához, emellett platform-független.

<code>int sum = 0;</code>	B9 00 00 00 00
	B8 00 00 00 00
<code>for (int i = 0; i < len; ++i)</code>	81 F9 0A 00 00 00
<code>{</code>	7D 06
<code>sum += t[i];</code>	03 04 8B
<code>}</code>	41
	EB F2

6.1. ábra. Magas szintű nyelv és gépi kódja

Megelőlegezzük, hogy a magas szintű nyelvek és a gépi kód között helyezkedik el az **Assembly**, ami egy *ember számára olvashatóbb* változatát nyújtja a *gépi kódnak*. Kriptikus hexadecimális számok helyett rendkívül egyszerű műveletek, regiszterek, címkék, ugróutasítások állnak a programozó rendelkezésére. Azt a programot, ami egy Assembly-forráskódból gépi kódot generál, **assemblernek** nevezzük. Bővebben a róla szóló fejezetben les szó.



6.2. ábra. A forráskód állapotának szakaszai

6.1. Fajtái

A programozási nyelveket háromféle csoportba oszthatjuk attól függően, milyen stratégiát követ az adott nyelv fordítóprogramja.

6.1.1. Fordított programozási nyelvek

Léteznek az ún. **fordított programozási nyelvek** (*compiled programming languages*, 6.3. ábra), melyeknek fordítóprogramja generál egy, a gépen közvetlenül futtatható állományt. A fordítás folyamata lassabb, azonban a létrejött program végrehajtása gyors. Ha hiba merül fel, két időszakaszban jelentkezhetnek ezek.

- **Fordítási időben** történik (*compile time*), ha a compiler veszi észre, jelez róla vissza – ekkor megszakítja a fordítást. Az ilyen hibát fordítási idejű hibának hívjuk.
- **Futási időben** történik (*run time* vagy *runtime*), ha a fordítás sikeres volt, létrejött a futtatható gépi kód, de működés közben elszáll. Ezek a futási idejű hibák.

Az ilyen nyelvekben a forráskód alaposabb ellenőrzése erősen javasolt.

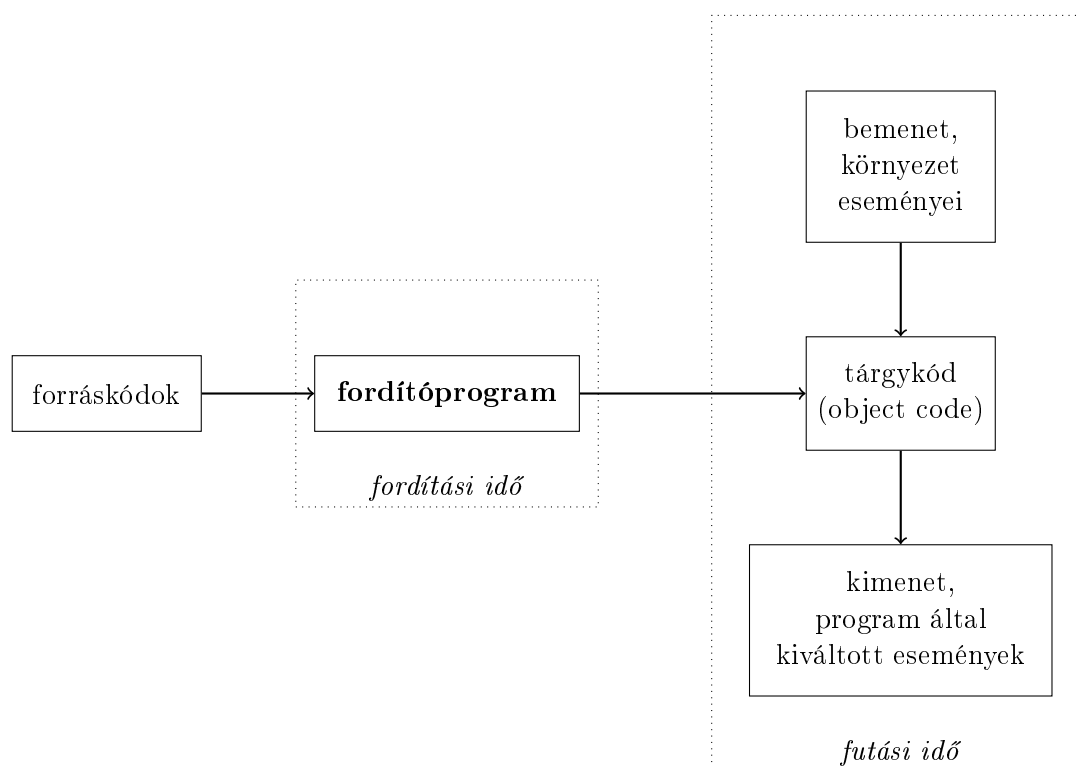
További előnye a fordított nyelveknek, hogy a fordító képes a **tárgykódot optimalizálni**, akár az adott platformra specifikusan. Hátránya sajnos ebben is rejlik, hiszen **minden platformra külön-külön le kell fordítanunk**.

Tipikusan ilyen nyelvek: C, C++, Haskell, Ada, ...

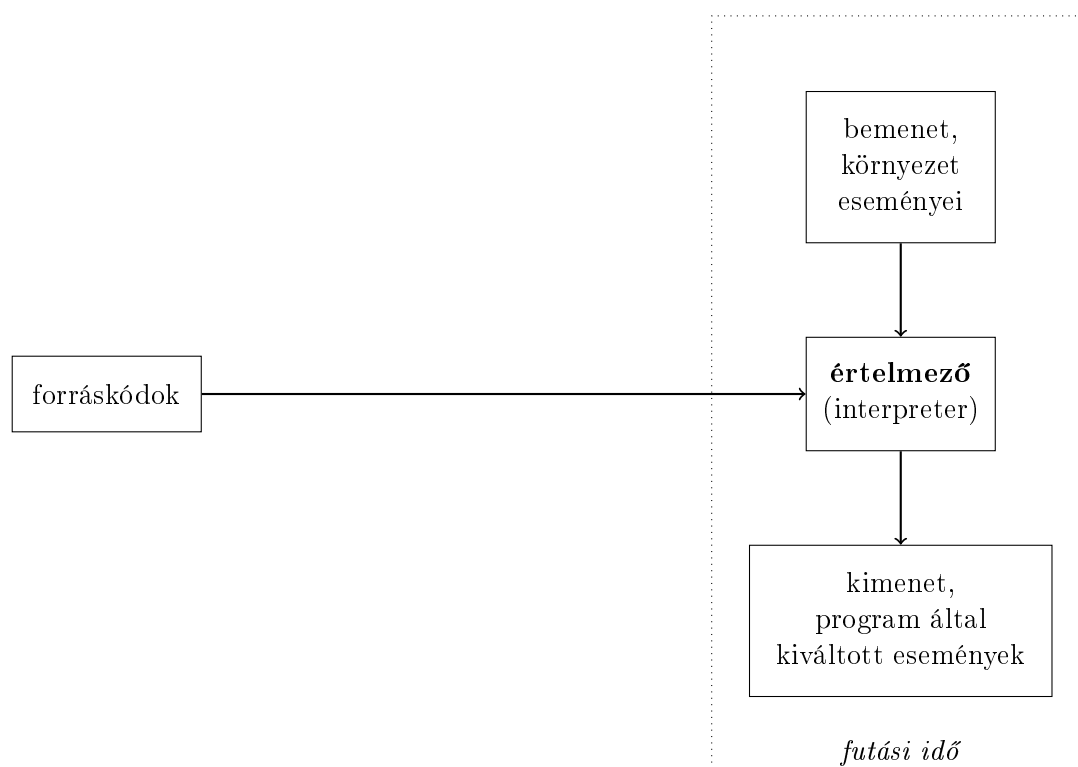
6.1.2. Értelmezett programozási nyelvek

A másik nagy csoportot képezik az **értelmezett** vagy **interpretált programozási nyelvek** (6.4. ábra). Ez némi rugalmasságot enged meg a fordított nyelvekkel szemben. Itt a fordító sorrol sorra hajtja végre az utasításokat és ott áll meg, ahol a hiba jelentkezik – azaz, **csak futási idő van**. További következménye, hogy jellemzően **jelentősen lassabb a végrehajtás**. Cserébe **minden platformon azonnal futtatható**, ahol az interpreter rendelkezésre áll.

Tipikusan ilyen nyelvek: Python, Perl, PHP, JavaScript, ...



6.3. ábra. Fordítás és végrehajtás



6.4. ábra. Értelmezés

6.1.3. Fordítás végrehajtás közben

Létezik a két stratégiának az ötvözete, a **fordítás végrehajtás közben** (angolul *just-in-time (JIT) compilation*).

Hasonlóan a fordított programozási nyelvekhez, a fordítási és futási idő elkülönül. A fordítóprogram gépi kód helyett **bájt kódra** (*bytecode*) fordítja le a forráskódot. Ezután a **virtuális gép** (*virtual machine*) végrehajtja a bájt kód utasításait. Azonban felmerülnek bizonyos problémák.

1. Ha a virtuális gép *futási időben értelmezi* a bájt kódot, az ugyanolyan lassan történne, mint egy hagyományos értelmezett nyelv esetében.
2. Ha *végrehajtás előtt fordítjuk le* teljesen a bájt kódot gépi kódra, az túl nagy kezdeti lassulást eredményezne.

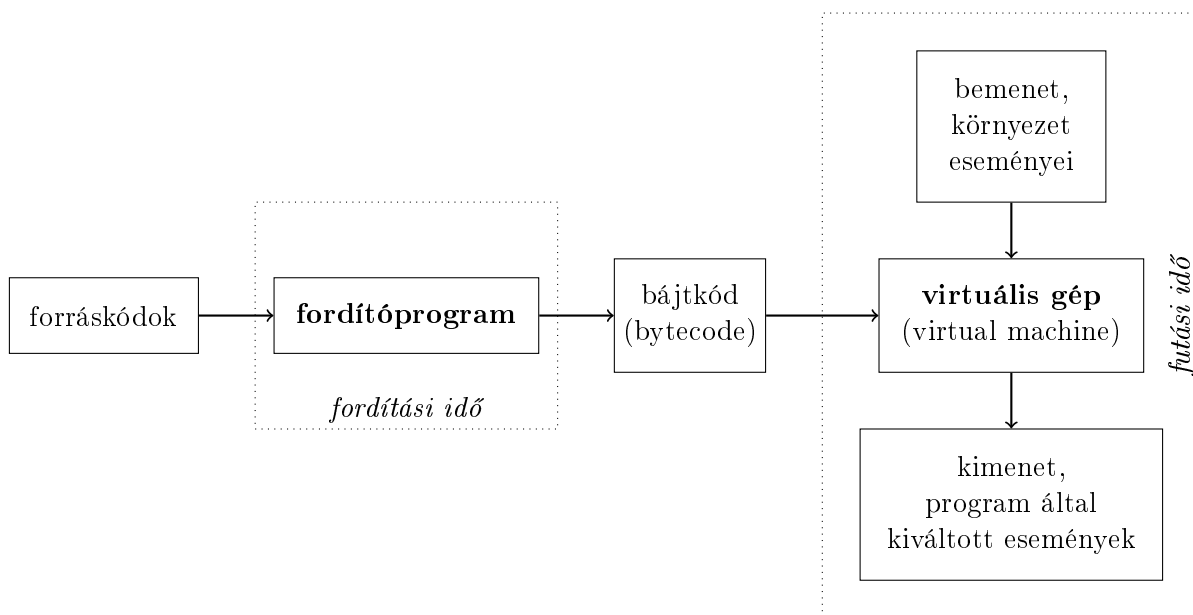
Éppen ezért a következő stratégiát követi a virtuális gép.

1. Kezdetben értelmezi, interpretálja a bájt kódot.
2. Futási időben statisztikákat gyűjt a leggyakrabban lefutó kódrészletekről. Ezeket „*hot spot*”-oknak nevezzük.
3. Ezeket lefordítja gépi kódra.
4. Így a következő alkalommal a lefordított kódrészlet fut az értelmezés helyett.

További előnye, hogy a JIT fordító futási időben gyűjtött információkat is figyelembe vehet a kódoptimalizálásnál. Ilyenekhez a klasszikus fordítóprogram nem fér hozzá!

Ugyanakkor, a bájt kódok végrehajtása jellemzően még így is lassabb a gépi kódhoz képest, de ez speciális alkalmazási területeket leszámítva nem baj.

Tipikusan ilyen nyelvek: C#, Java.



6.5. ábra. JIT-fordítás

Nyelv	Bájtkód	Virtuális gép
C#	Common Intermediate Language (CLI)	Common Language Runtime (CLR)
Java	Java bytecode	Java Virtual Machine (JVM)

6.2. Fordítóprogramok fejlődése

- 1957: Első **Fortran compiler** – 18 emberévnyi munka
- Azóta fejlődött a **formális nyelvek és automaták elmélete**.
- Ma: A fordítóprogramok létrehozásának egy része **automatizálható** elemzőgenerátorokkal.
 - A programszöveg elemi egységekre (tokenekre) bontása
 - A programszöveg formai helyességének vizsgálata
- A további ellenőrzések és a kódgenerálás nem automatizálható, de az implemetációt **keretrendszerek** segíthetik.
- A **kódoptimalizálás** (és a hozzá szükséges elemzések) komoly kihívás.

6.3. Logikai felépítése

Két fázisból áll a fordítás: az **analízis**ből és **szintézis**ből. Az egyes részeket és alrészletet önálló fejezetek is részletezik, itt csak egy rövid áttekintést nyújtunk.

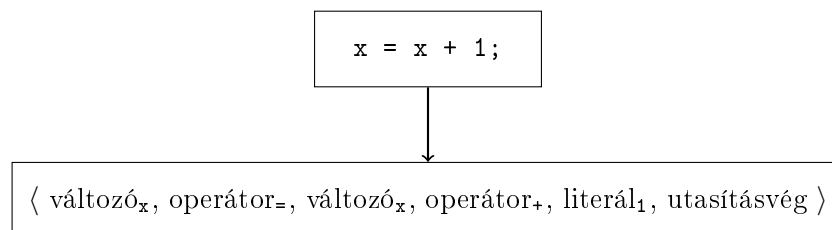
A vizuális összefoglaló a 6.10. ábrán található.

6.3.1. Analízis

Az analízis *előfeldolgozást* hajt végre a bemeneti forráskódon. Ellenőrzi, hogy lexikálisan, szintaktikusan, illetve szemantikusan helyes-e a kódunk. Ha ez nincs így, a megfelelő helyen hibát dob.

Lexikális elemzés

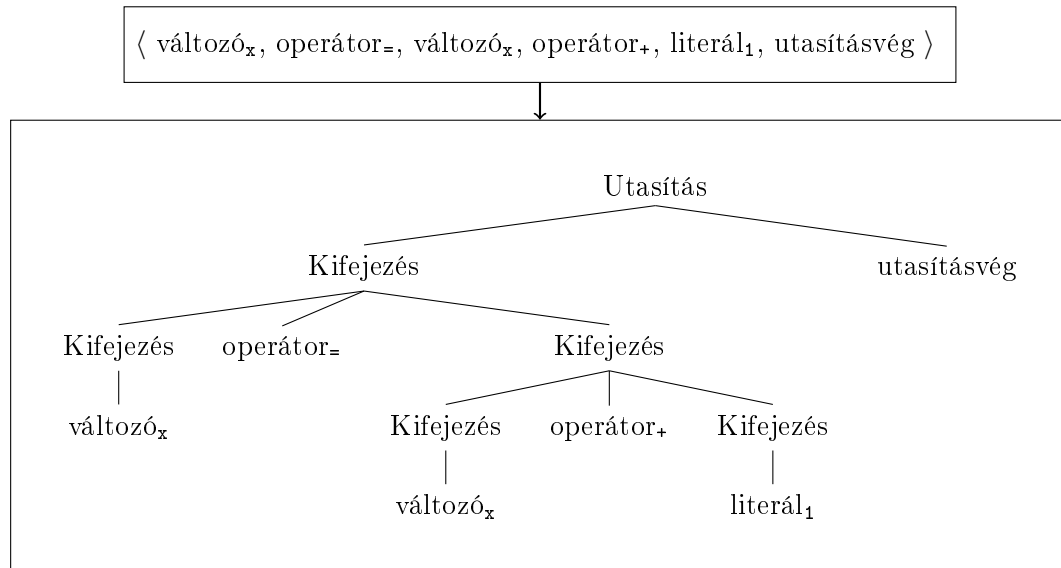
- *Feladat*: A forrásszöveg elemi egységekre, ún. **token**ekre bontása. Idegen szóval ez a **tokenizáció**.
- *Bemenet*: karaktersorozat
- *Kimenet*: tokenek sorozata + lexikális hibák
- *Eszközök*: reguláris kifejezések, véges determinisztikus automaták



6.6. ábra. Helyes lexikális elemzés eredménye

Szintaktikus elemzés

- *Feladat*: A forrásszöveg szerkezetének felderítése, formai ellenőrzése.
- *Bemenet*: tokenek sorozata
- *Kimenet*: szintaxisfa + szintaktikus hibák
- *Eszközök*: környezetfüggetlen nyelvtanok, veremautomaták



6.7. ábra. Helyes szintaktikus elemzés eredménye

Az alábbi környezetfüggetlen nyelv határozza meg a szintaxist. Ezen nyelvnek a terminális szimbólumai a tokenek (lexikális elemek).

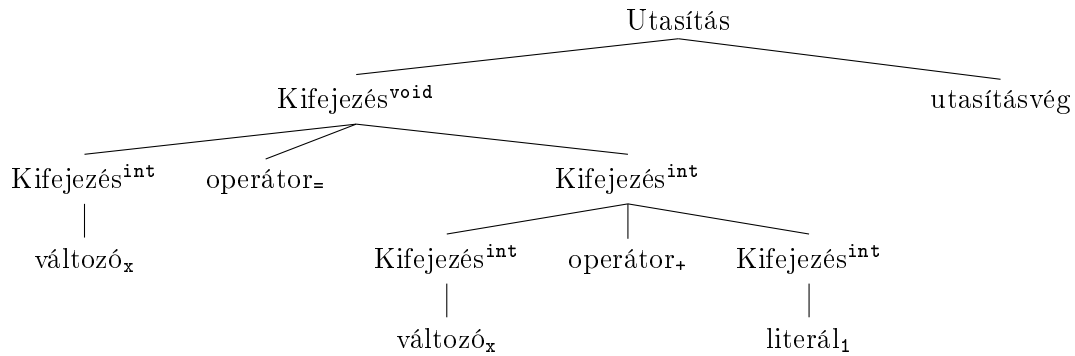
$\langle \text{Utasítás} \rangle ::= \langle \text{Kifejezés} \rangle \text{ utasításvég}$
 $\langle \text{Kifejezés} \rangle ::= \text{változó} \mid \text{literál} \mid \langle \text{Kifejezés} \rangle \text{ operátor } \langle \text{Kifejezés} \rangle$

Szemantikus elemzés

- *Feladat*: A statikus szemantika (pl. változók deklaráltsága, típushelyesség stb.) ellenőrzése
- *Bemenet*: szintaxisfa
- *Kimenet*: szintaxisfa attribútumokkal, szimbólumtábla + szemantikus hibák
- *Eszközök*: attribútumnyelvtanok

Név	Típus
x	int

6.8. ábra. Szimbólumtáblázat. Általában több információt is tartalmaz.



6.9. ábra. Szintaxisfa attribútumokkal

6.3.2. Szintézis

Kódgenerálás

- *Feladat*: Alacsonyabb szintű belső reprezentációkra, végül **tárgykóddá** alakítja a programot
- *Bemenet*: szintaxisfa attribútumokkal, szimbólumtábla
- *Kimenet (az utolsó menetben)*: **tárgykód**
- *Eszközök*: **kódgenerálási sémák**

Közvetlenül gépi kódot csak nagyon indokolt esetben érdemes generálni. Helyette Assembly kód (pl. valamely platform Assembly nyelve vagy LLVM) generálható, amit assemblerekkel fordítunk tovább.

Megemlíthetjük az ún. **transzlációt** is. Ez magas szintű nyelvek közti fordítást jelent. Ez lehet végcél (pl. projektek portolása esetén egyik nyelvről a másikra), ugyanakkor elterjedt nyelvekre való fordítás esetén használhatjuk azok fordítóit a gépi kód / bájtkód előállításához.

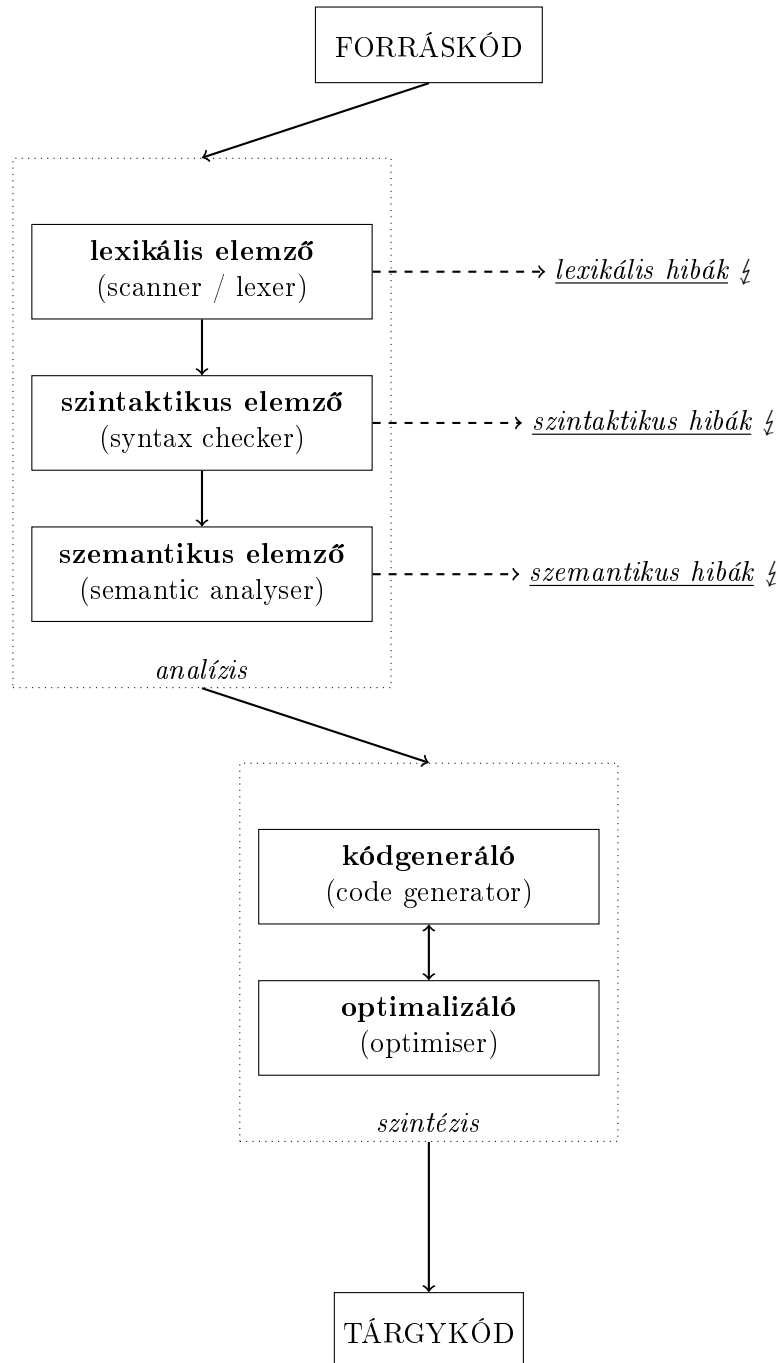
Optimalizáció

- *Feladat*: Kód átalakítása hatékonyságnövelés céljából (pl. sebességnövelés, memóriaigény csökkentés)
- *Bemenet*: belső reprezentáció / tárgykód
- *Kimenet (az utolsó menetben)*: belső reprezentáció / tárgykód
- *Eszközök*: **Statikus elemzés, transzformációs keretrendszerek**

Egyes compilerek több lépésben is optimalizálhatják a kódot.

Ahogy megtárgyaltuk, a szintézis fázisa úgy kezdődik, hogy rendelkezésünkre áll a szintaxisfa. Ezen ún. **magas szintű optimalizációt** hajtanak végre, így kapunk egy optimalizált szintaxisfát¹. Ez alapján megtörténik a kódgenerálás, létrejön az Assembly kód. Végül ezt az Assembly kódot optimalizáljuk (**alacsony szintű optimalizálás**).

¹El tudjuk képzelni, hogy milyen matematikai vonatkozásai lehetnek ennek: egy fagráfot kevesebb úttal vagy csúccsal „írjunk fel” úgy, hogy az ezen gráf által felírt „program” ekvivalens maradjon az eredetivel. (A megfogalmazás természetesen matematikailag pontatlan, csupán a szemléltetés céljából raktam ide.)



6.10. ábra. A fordítóprogramok logikai felépítése

6.4. Szerkesztés és végrehajtás

Általában amikor programot írunk, modularizálva írjuk meg azt, azaz több, kisebb részekre bontjuk fel – legtöbbször **könyvtárak**, **csomagok** formájában. Ezen összetevők gyakran hivatkoznak egymásra, emiatt elengedhetetlen, hogy el is ériék egymást. Ezt oldja meg a(z) **(össze)szerkesztés** vagy **linkelés**.

A mai rendszereken kétféle stratégia létezik a könyvtárak összeszerkesztéséhez.

1. Statikus szerkesztés

Nagy vonalakban azt jelenti, hogy mindazon **könyvtárakat**, **csomagokat**, melyeket felhasználunk a programunkban, „beleégetjük” a **gépi kódba**. Tipikusan ez történik, amikor C-ben include-oljuk az `stdio.h` könyvtárat. Hiába csak a `printf` függvényt használjuk fel, minden más is bekerül a binárisba.

Ez előnyös lehet, mivel csökkenti a külső függőségeket (akár használhatjuk a programunkat olyan rendszeren, amin nincs telepítve a `glibc`). Hátránya, hogy jelentősen megnövelheti a futtatható fájl méretét.

A statikus könyvtárak tipikus kiterjesztései: `.a`, `.lib`.

2. Dinamikus szerkesztés

Futási időben éri el a hivatkozott függvényeket, osztályokat, stb. Előnye, hogy kisebb lesz a futtatandó fájl mérete. Hátránya pedig, hogy meg kell győződnünk futtatás előtt, hogy telepítve vannak-e a szükséges **függőségek**.

A dinamikus könyvtárak tipikus kiterjesztései: `.so` (*shared object*), `.dll` (*dynamically linked library*).

Ha visszaemlékszünk az *Objektumelvű programozás* c. tárgyból tanultakra, a gyakorlatokon használtuk a `TextFileReader.dll` könyvtárat, ami (a mostani tudásunkkal összevetve) egy dinamikusan linkelt könyvtár.

A programunk futtatása, végrehajtása esetén a **teljes futtatható állományt betöltjük a fájlrendszerből**. Ha dinamikus könyvtárakat is használunk, ezek is betöltésre kerülnek.

7. fejezet

Lexikális elemzés

7.1. A tokenizáció

Adott az alábbi karakterlánc:

$$x = (x + 2) * 3;$$

A feladat, hogy hogyan állapíthatjuk meg a benne lévő tokeneket?

Számunkra ránézésre nyilvánvaló, hogy a helyes tokenizáció a következő:

$$\langle x, =, (, x, +, 2,), *, 3, ; \rangle.$$

Ezt kell valahogy a „számítógép nyelvén” kifejeznünk. Megállapítunk bizonyos tulajdonságokat, melyekkel kizárásos alapon kiválaszthatjuk a tokeneket.

- **Aminek a belső szerkezete fontos, nem lehet token!**

Például az értékadásnak van bal és jobb oldala, mindkettőnek megvannak a rá vonatkozó szabályai.

$$\langle x, =, (x + 2) * 3, ; \rangle \not\prec$$

Mivel az értékadás önmaga is egy utasítás, amire szintén vonatkoznak szabályok, emiatt az alábbi tokenizáció sem helyes.

$$\langle x = (x + 2) * 3, ; \rangle \not\prec$$

- **Aminek a formája nem írható le reguláris kifejezéssel, nem lehet token!**

Tipikus példája ennek a helyes zárójelezések nyelve, ami környezetfüggetlen grammatikával írható le.

$$\langle x, =, (x + 2), *, 3, ; \rangle \not\prec$$

Következő probléma: a **fehérélválasztók** (*whitespaces*). A legtöbb programozási nyelvénél a szóközők, tabulátorok és újsorok **nem alkotnak tokeneket**, csak más tokenek elválasztására valók. A lexikális elemzőnek fel kell ismernie ezeket, de nem kell továbbítania a szintaktikus elemző felé.

Ezalól kivételt képeznek a **behúzásra** (vagy indentációra) **érzékeny nyelvek**, mint a Python vagy a Haskell. Az elemzőnek a **sorok behúzását számon kell tartania**. Növekvő behúzás jelenti a blokknyitó token (C-ben {), a csökkenő behúzás meg a blokkzáró token (C-ben }).

Érdekességgként megemlítjük a *Whitespace* nyelvet, amiben kizárólag a fehérélválasztóknak van jelentése.

Ahogy korábban megállapítottuk, token csak az lehet, amit leírhatunk reguláris kifejezéssel. Bizonyos reguláris kifejezések elsőbbséget élveznek a többivel szemben – nevezetesen azok, melyekkel kulcsszavakat írunk le. Tehát a konkrétabbak előrébb, az általánosabbak hátrébb kerülnek a felsorolásban.

Reguláris kifejezés	Példák	Token típus
<code>while</code>	<code>while</code>	kulcsszó a While nyelvben
<code>[a-zA-Z][a-zA-Z0-9_]*</code>	<code>x, apple123, list_length</code>	azonosító
<code>[+-]?[0-9]+</code>	<code>0, 123, -2, +100</code>	egész számliterál
<code>[\t\n]+</code>	(fehérélválasztók nemüres sorozata)	–
<code>"//".*</code>	<code>// Ez egy megjegyzés</code>	–

7.1. ábra. Definíció reguláris kifejezésekkel

7.2. A lexikális elemzés elvei

- **Leghosszabb illeszkedés elve**

A leghosszabban illeszkedő karaktersorozatból képzünk tokenet.

Pl. `w|hile, wh|ile, ..., whil|e` → Hiába helyes azonosító szimbólum a `w`, `wh`, ..., `whil`, mégis folytatni kell a keresést.

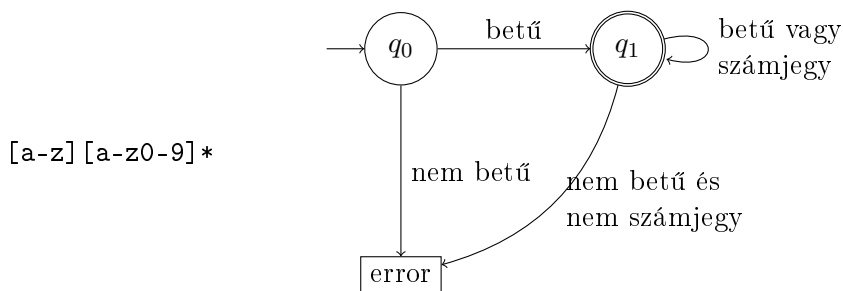
- **Prioritás elve**

Ha a leghosszabban illeszkedő karaktersorozat több reguláris kifejezésre is illeszkedhet, a sorrendben korábban álló „nyer”.

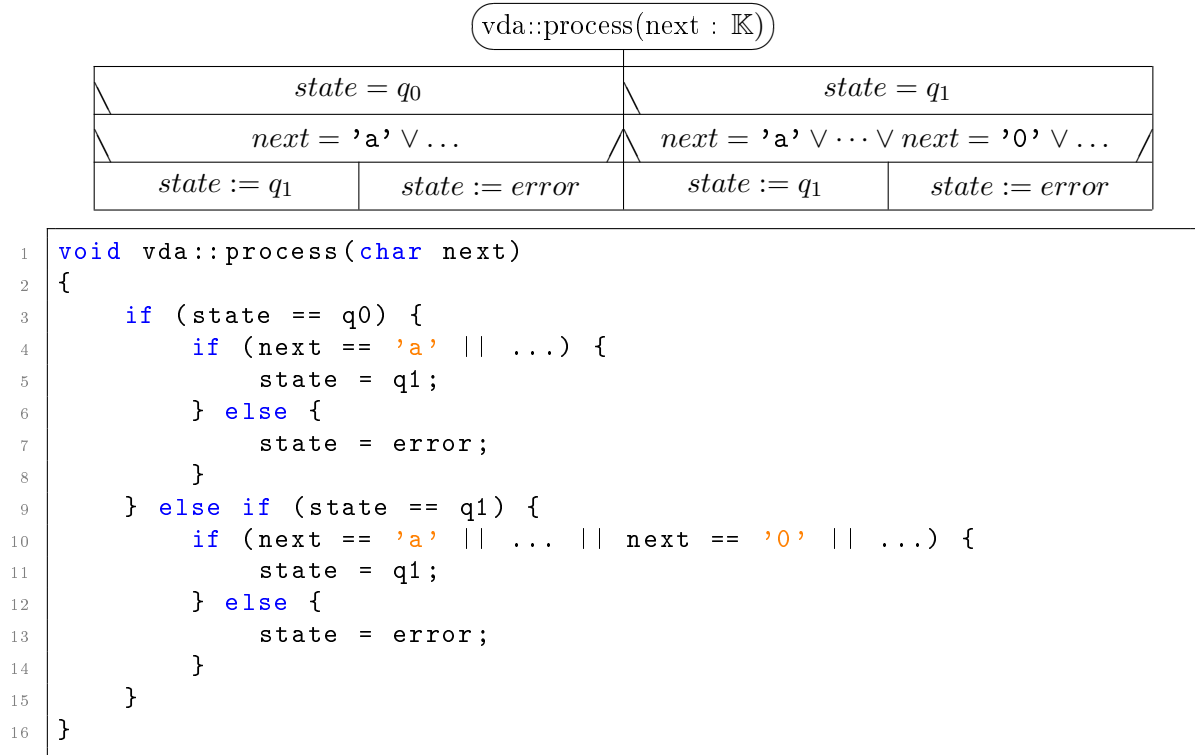
Pl. `while|` → Lehet kulcsszó is, lehet azonosító is. Mivel kulcsszóként korábban definiáltuk, így ez élvez elsőbbséget.

7.3. Implementációja

A reguláris kifejezések átalakíthatók **véges determinisztikus automatává**.



A VDA implementációja történhet **elágazásokkal**, amelynek a struktogramja itt látható (a `ℕ` jelöli a `char` adattípust).

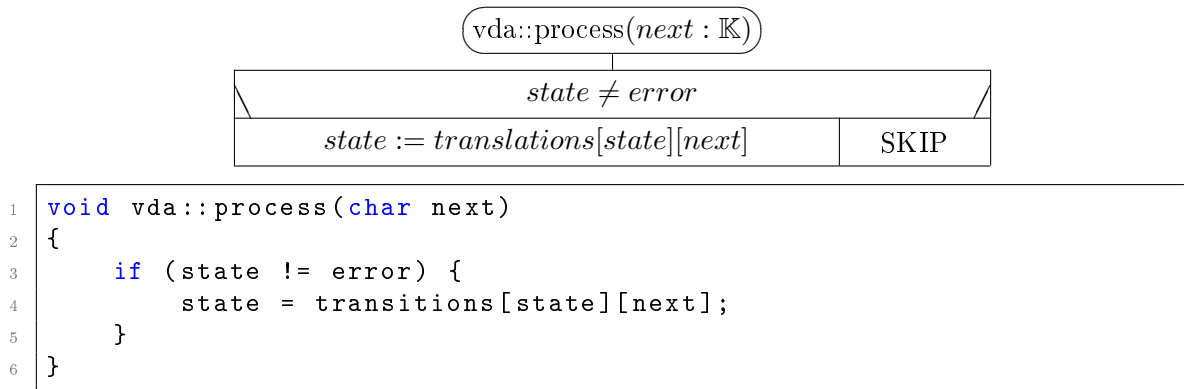


VDA implementációja elágazásokkal

Megoldható ugyanakkor **táblázattal** is.

	q_0	q_1
a	q_1	q_1
\vdots		\vdots
0	error	q_1
\vdots		\vdots
other	error	error

7.2. ábra. A VDA táblázata



VDA implementációja táblázattal

7.4. Tokenhez csatolt információk

A felismert tokenekhez a lexikális elemző kiegészítő információkat csatol. Ezeket nevezzük **kitüntetett szintetizált attribútumoknak**. A jelentőségük a szemantikus elemzésnél fog megjelenni.

- **Minden tokenhez:** a token pozícióját (első karakter sor- és oszlop-, utolsó karakter sor- és oszlopszáma)
- **Azonosítókhöz:** az azonosító szövegét (ez szükséges a szemantikus elemzéshez)
- **Literálokhoz:** a literál értékét (kódgeneráláshoz, kódoptimalizációhoz szükséges)

7.5. Lexikális hibák

Lexikális hiba esetén hibajelzést ad a fordító, és *folytatja az elemzést*. A leggyakrabban előforduló hibák:

- Illegális karakter: A nyelv ábécéjébe nem tartozó karakter az inputszövegben.
Az addig felépített token kiadja, ha volt illeszkedés. Az illegális karaktert követő karakterrel folytatódik az elemzés.
- Lezáratlan sztring
A sor végén derül ki; az őt követő sorban folytatódik az elemzés.
- Lezáratlan többsoros megjegyzés
A fájl végén derül ki; nincs további elemzés.

8. fejezet

Szintaktikus elemzés

8.1. Grammatikai előfeltételek

A lexikális elemzés kinyerte a tokenek sorozatát a forrásfájlból. Ebben a lépésben az a feladatunk, hogy ezen tokenekből a „nyelvtani hierarchiát”, a **szintaxisfát** állítsuk fel. Ehhez szükségünk vannak a **környezetfüggetlen nyelvtanokra**, valamint az ezek elfogadására szolgáló **veremautomatákra**.

Szintaktikus elemzőt manapság nagyon egyszerűen hozhatunk létre különböző generátorok segítségével. Ilyen például a Bison, amit a gyakorlaton is használunk. Ennek a forrásfájljában (pl. `while.y`) megadjuk a lehetséges tokeneket és definiáljuk a szabályainkat.

Ahhoz, hogy elemezhető nyelvet tudjunk készíteni, a nyelvtanunknak szüksége van arra, hogy bizonyos előfeltételeket teljesítsen.

1. Redukáltság: Nincsenek „felesleges” nemterminálisok.
Mindegyik nemterminálishoz adható olyan levezetés, amiben szerepel, és nem üres terminális sorozatot vezetünk le belőle.
2. Ciklusmentesség: Nincs $A \rightarrow^+ A$ levezetés.
Ciklusos nyelvtan olyan, aminek az egyik bal oldalának levezetéséből visszajuthatunk önmagába. Példa ciklusos (tehát nem jó) nyelvtanra:

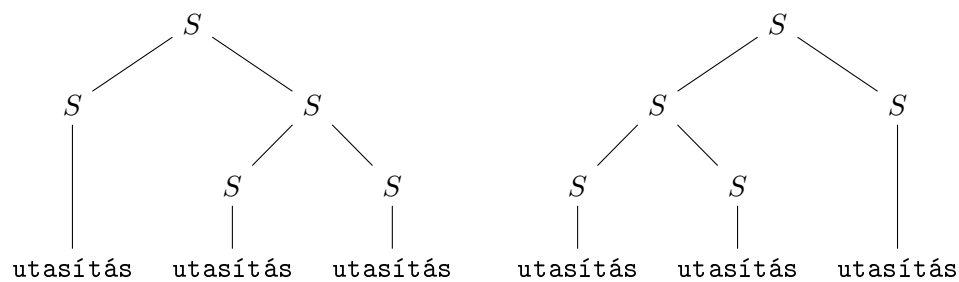
$$\begin{aligned} S &\rightarrow A \\ A &\rightarrow a \mid B \\ B &\rightarrow A. \end{aligned}$$

3. Egyértelműség: Minden szóhoz **pontosan egy szintaxisfa** tartozik.

- Több levezetés tartozhat egy szóhoz, de a szintaxisfáik legyenek identikusak!

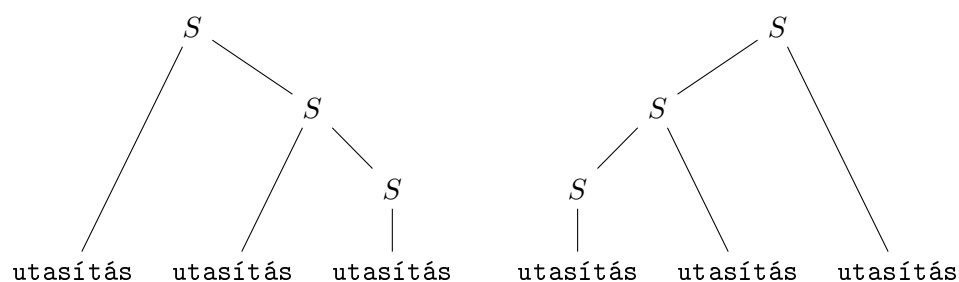
$$\begin{array}{l} S \Rightarrow AB \Rightarrow aB \Rightarrow ab \\ S \Rightarrow AB \Rightarrow Ab \Rightarrow ab \end{array} \quad \begin{array}{c} S \\ \swarrow \quad \searrow \\ A \quad B \\ | \quad | \\ a \quad b \end{array}$$

- Példa nem egyértelmű nyelvtanra: $S \rightarrow \text{utasítás} \mid SS$.



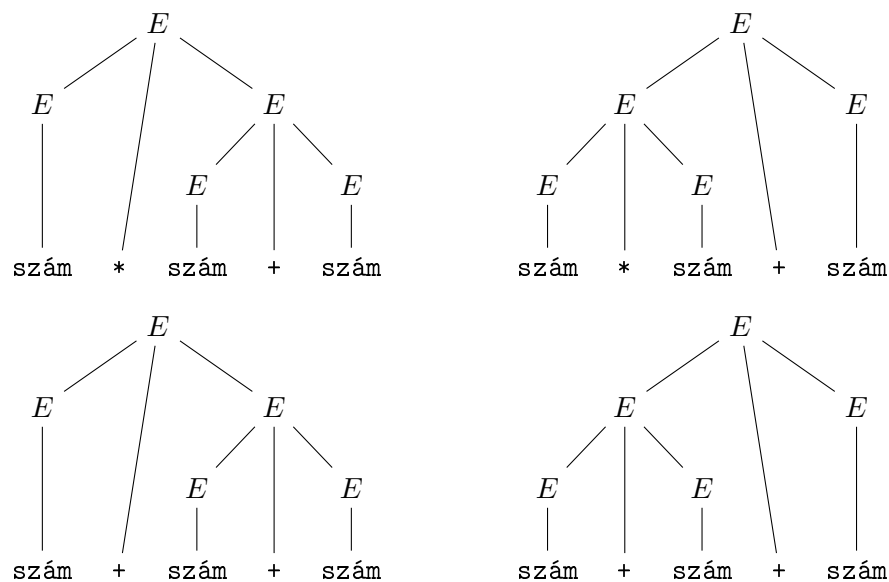
Ez a nemegyértelműség feloldható a nyelvtan átalakításával:

$$S \rightarrow \text{utasítás } S \mid \text{utasítás} \quad \text{vagy} \quad S \rightarrow S \text{ utasítás} \mid \text{utasítás}.$$



- A nem-egyértelműség feloldható, ha megadjuk az operátorok precedenciáját és asszociativitását. Az alábbi nyelvtan nem egyértelmű:

$$E \rightarrow \text{szám} \mid E + E \mid E * E.$$



Átalakítva:

- A $*$ magasabb precedenciájú, mint a $+$.
- Mindkét operátor balasszociatív.

$$E \rightarrow F \mid E + F$$

$$F \rightarrow \text{szám} \mid F * \text{szám}.$$

8.2. Felülről lefele elemzés

A felülről lefele elemzés az egyik lehetséges stratégiája a szintaktikus elemzésnek. A startszimbólumból indulva a terminálisok felé építjük a szintaxisfát. A bemenet feldolgozása balról jobbra történik, így ezáltal legbaloldalibb levezetést állít elő – ami azt jelenti, hogy több terminális esetén a legbaloldalibbat helyettesíti.

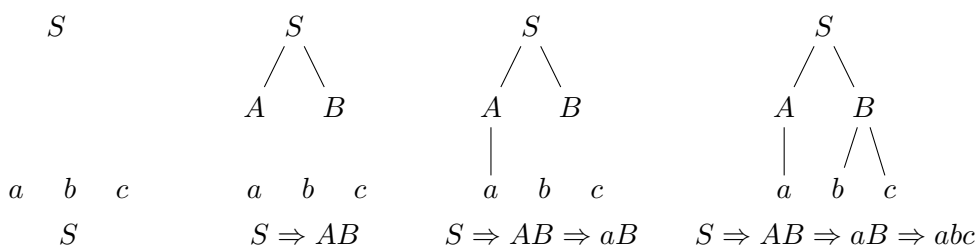
Szemléltessük az alábbi nyelvtanon:

$$S \longrightarrow AB$$

$$A \longrightarrow a$$

$$B \longrightarrow bc.$$

Legyen a bemeneti szövegünk: *abc*. A szó szintaxisfáját így kapjuk meg:



8.1. ábra. Felülről lefele elemzés lépései

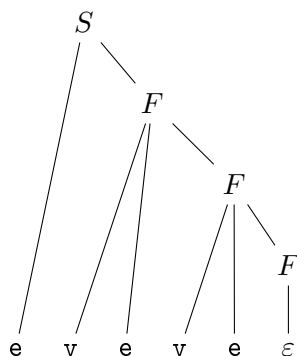
Felmerül a kérdés: **mi alapján választjuk ki a használandó szabályt?** A probléma egyszerűen feloldható előreolvasással. Vegyük a következő példát!

Legyen a nyelvtanunk, ami vesszővel (v) elválasztott elemek (e) listáját írja le. Megengedjük az üres listát is (ε).

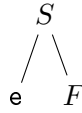
$$S \longrightarrow \varepsilon \mid \mathbf{e}F$$

$$F \longrightarrow \varepsilon \mid \mathbf{ve}F$$

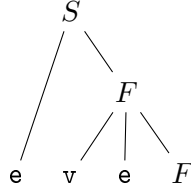
A példaszövegünk legyen „apple, banana, pear”. A lexikális elemzővel megkapjuk a tokenek sorozatát, ami „eveve”. Megelőlegezzük, hogy a szintaxisfának így kell kinéznie.



Szemléltessük a szintaktikus elemzést! Egyelőre nem olvastunk egy karaktert sem, emellett a szintaxisfánk is kizárólag az S startszimbólumból áll. Beolvastunk egyet, a szövegünk e lesz. Megnézzük, hogy erre melyik szabály passzol. Mivel az $S \rightarrow eF$ jobb oldala ugyanezzel a karakterrel kezdődik, így ezt választjuk. Így a fánk már 3 csúcsból áll.



Folytatjuk az elemzést, előreolvassunk ismét egy karaktert. A szövegünk állapota ev , hisz v -t olvastunk be. Ezt kihasználva kiválasztjuk az $F \rightarrow veF$ szabályt. A szintaxisfánk állapota:



Ezt az eljárást addig folytatjuk, ameddig fel nem dolgoztuk a teljes szöveget. Ha nem marad már beolvasandó karakter, akkor az ε -t kapjuk, ami biztosítja, hogy befejezhessük az elemzést. Ha felidézük a végleges fát, a legvégén láthattunk egy elsőre feleslegesnek tűnő üres szót. Valójában pont emiatt került a végére.

Következő kérdés: **hány tokent kell előreolvasnunk?** Szerencsére, ezt is megválaszolhatjuk, ugyanis ez a nyelvtan tulajdonságaitól függ. Az előző nyelvtan esetében elegendő volt 1-et előre olvasnunk, míg más nyelvtanok esetében más lehet ez a konstans.

Ennek jellemzéséhez bevezetjük az $LL(k)$ nyelvtan fogalmát.

8.2.1. definíció ($LL(k)$ nyelvtan). Egy környezetfüggetlen nyelvtan $LL(k)$ tulajdonságú valamely $k \in \mathbb{N}^+$ számra, ha felülről lefelé elemzés esetén a legbaloldali feldolgozatlan nemterminálishoz egyértelműen meghatározható a rá alkalmazandó nyelvtani szabály legfeljebb k token előreolvasásával.

Az elnevezés a „*left to right using leftmost derivation*” elnevezés angol rövidítéséből származik. A legutóbbi példánk $LL(1)$ tulajdonságú.

Megjegyzések.

- Azt mondtuk, hogy a megfelelő szabály kiválasztása előreolvasással oldható meg. Ez azt feltételezi, hogy nulla karaktert nem olvashatunk előre, ezért $k \in \mathbb{N}^+$.
- Nem minden nyelvtanhoz adható meg ilyen konstans. A következő tétel ezt mondja ki (nem bizonyítjuk).

8.2.1. tétel. Van olyan nyelvtan, ami semmilyen $k \in \mathbb{N}^+$ -re sem $LL(k)$.

Például az alábbi nyelvtanhoz nem létezik megfelelő $k \in \mathbb{N}^+$ szám.

$$\begin{aligned}
 S &\rightarrow A|B \\
 A &\rightarrow a|aA \\
 B &\rightarrow ab|aBb
 \end{aligned}$$

A tanulmányaink során az $LL(1)$ nyelvtanokkal foglalkozunk részletesebben. Ennek egy implementációja az ún. **rekurzív leszállás**.

A rekurzív leszállást azért kedveljük, mivel rendkívül kényelmessé teszi az elemző lekódolását. Gyakorlatilag arra van szükségünk, hogy a **nyelvtani szabályokat** közvetlenül **átírjuk függvényekké** egy tetszőleges programozási nyelvben.

1. **Mindegyik nemterminálishoz írunk egy-egy függvényt.**

A példák a korábban definiált „listás nyelv” nemterminálisait illusztrálják.

2. **Minden szabályalternatívát egy-egy elágazás ágaként fejezünk ki.**

Például a $S \rightarrow \varepsilon \mid eF$ szabály két ágból fog állni; egy az üres szó esetéért felel, a másik meg a lista fejeleméért.

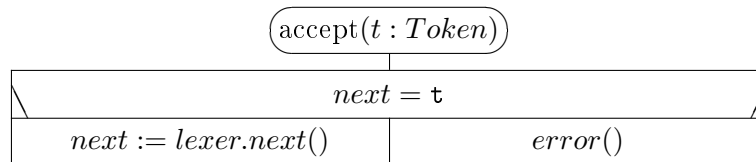
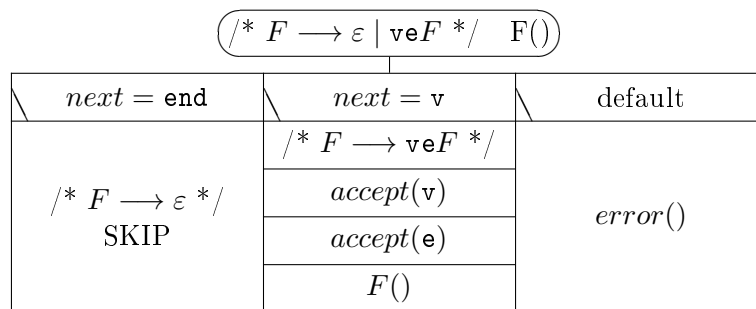
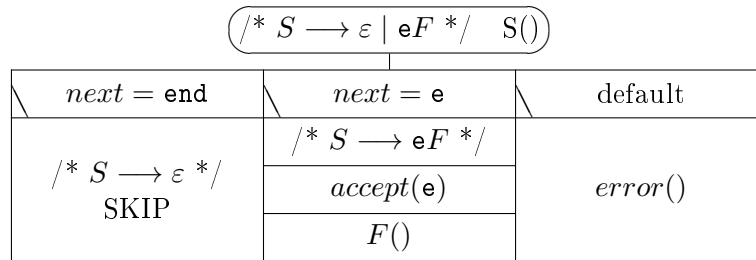
Gondoskodnunk kell a hibakezelésről is, emiatt egy további ágot fentartunk erre a célra. Így végső soron egy 3-ágú elágazásunk lesz.

3. **Az ágak belsejében a szabály jobboldalának minden szimbólumához egy-egy utasítást rendelünk.**

A terminálisokhoz egy-egy *accept()* függvényhívás fog tartozni, míg a nemterminálisokhoz a hozzájuk eljárás kerül meghívásra($S()$, $F()$).

4. Az *accept()* eljárás feladatai:

- Ha a várt token következik, akkor új token kérést a lexikális elemzőtől.
- Egyébként hibát jelez.



8.2. ábra. A példanyelvtan elemzőjének függvényeinek struktogramjai

Megjegyzések.

- Ha a nyelvtan rekurzív, akkor rekurzív vagy kölcsönösen rekurzív függvényeket kapunk, innen a módszer neve.
- Valójában ez az elemző is veremautomata: a függvényhívásokat kezelő futási idejű verem az automata verme.
- A levezetés legbaloldalibb redukálható nemterminálisát **nyél**nek is nevezik.

Az egyes függvények felépítését elég alaposan körül tudjuk írni. Azonban felmerülhet a kérdés, hogy az **elágazások feltételeit miképpen tudjuk meghatározni?** A válasz a *FIRST* halmaz és *FOLLOW* halmaz fogalmában rejlik, amiket be is vezetünk.

8.2.2. definíció (*FIRST*₁ halmaz). Adott nyelvtan esetén egy α szimbólumsorozatra a $FIRST_1(\alpha)$ halmaz azokat a **terminálisokat** tartalmazza, amelyek az α -ból levezethető szimbólumsorozatok elején állnak.
Ha α -ból levezethető az üres szó (ε), akkor ε is eleme a halmaznak.

A *FIRST* halmaz általánosan n hosszú eredménysorozatokra is definiálható: $FIRST_n(\alpha)$.

$$\begin{aligned} FIRST_1(\varepsilon) &= \{\varepsilon\} & \underline{\varepsilon} \\ FIRST_1(\mathbf{e}F) &= \{\mathbf{e}\} & \underline{\mathbf{e}F} \\ FIRST_1(\mathbf{v}F) &= \{\mathbf{v}\} & \underline{\mathbf{v}F} \\ FIRST_1(F) &= \{\varepsilon, \mathbf{v}\} & \underline{\varepsilon} \text{ és } \underline{\mathbf{v}F} \end{aligned}$$

8.2.3. definíció (*FOLLOW*₁ halmaz). Adott nyelvtan esetén egy α szimbólumsorozatra a $FOLLOW_1(\alpha)$ halmaz azokat a **terminálisokat** tartalmazza, amelyek az α után állhatnak a kezdőszimbólumból induló levezetésekben.
Ha α után nem áll semmi, akkor $\#$ (szöveg vége jel) eleme a halmaznak.

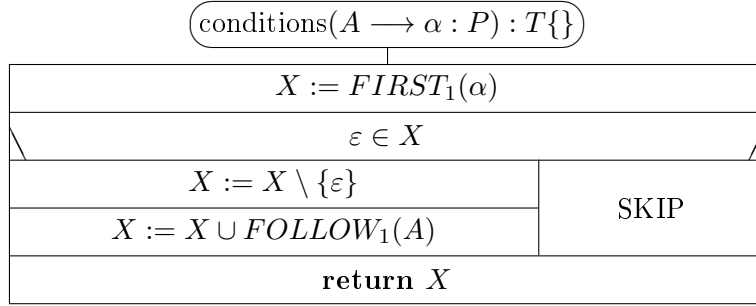
A *FOLLOW* halmaz általánosan n hosszú eredménysorozatokra is definiálható: $FOLLOW_n(\alpha)$.

$$\begin{aligned} FOLLOW_1(S) &= \{\#\} & S_ \\ FOLLOW_1(F) &= \{\#\} & S \Rightarrow \mathbf{e}F_ \Rightarrow \mathbf{v}F_ \Rightarrow \dots \\ FOLLOW_1(\mathbf{e}) &= \{\#, \mathbf{v}\} & S \Rightarrow \mathbf{e}F \Rightarrow \mathbf{e}_ \text{ és } S \Rightarrow \mathbf{e}F \Rightarrow \mathbf{e}\underline{\mathbf{v}F} \end{aligned}$$

Az elágazások feltételeinek meghatározását a következőképp tehetjük meg.

- Az $A \rightarrow \alpha$ szabályhoz meghatározzuk a $FIRST_1(\alpha)$ halmazt.
- Ha ebben van ε , akkor kivesszük ε -t és helyette hozzávesszük a halmazhoz $FOLLOW_1(A)$ elemeit.
- Az így kapott halmaz elemeiből (pl. x_1, x_2, \dots, x_n) képezzük az elágazás feltételét:

$$next = x_1 \vee next = x_2 \vee \dots \vee next = x_n.$$



8.3. ábra. Az elágazás feltételének meghatározásának algoritmus

Ellenőrizhető a struktogram segítségével, hogy valóban ezen eredmények jönnek ki.

$$\begin{aligned}
 conditions(S \rightarrow \varepsilon) &= \{\#\} \\
 conditions(S \rightarrow \mathbf{e}F) &= \{\mathbf{e}\} \\
 conditions(F \rightarrow \varepsilon) &= \{\#\} \\
 conditions(F \rightarrow \mathbf{v}F) &= \{\mathbf{v}\}
 \end{aligned}$$

8.2.2. tétel ($LL(1)$ tulajdonság ellenőrzése). *Egy környezetfüggetlen nyelvtan pontosan akkor $LL(1)$ tulajdonságú, ha bármely két $A \rightarrow \alpha$, $A \rightarrow \beta$ (a két A megegyezik!) szabályokhoz a fenti módon meghatározott halmazok diszjunktak.*

- Példa $LL(1)$ tulajdonságú nyelvtan.

$$\begin{aligned}
 conditions(S \rightarrow \varepsilon) &= \{\#\} \\
 conditions(S \rightarrow \mathbf{e}F) &= \{\mathbf{e}\} \\
 \{\#\} \cup \{\mathbf{e}\} &= \emptyset \quad \checkmark
 \end{aligned}$$

$$\begin{aligned}
 conditions(F \rightarrow \varepsilon) &= \{\#\} \\
 conditions(F \rightarrow \mathbf{v}F) &= \{\mathbf{v}\} \\
 \{\#\} \cup \{\mathbf{v}\} &= \emptyset \quad \checkmark
 \end{aligned}$$

- Példa nem $LL(1)$ tulajdonságú nyelvtanra.

$$\begin{aligned}
 conditions(S \rightarrow \varepsilon) &= \{\#\} \\
 conditions(S \rightarrow N) &= \{\mathbf{e}\} \\
 \{\#\} \cup \{\varepsilon\} &= \emptyset \quad \checkmark
 \end{aligned}$$

$$\begin{aligned}
 conditions(N \rightarrow \mathbf{e}) &= \{\mathbf{e}\} \\
 conditions(N \rightarrow N\mathbf{v}F) &= \{\mathbf{e}\} \\
 \{\mathbf{e}\} \cup \{\mathbf{e}\} &\neq \emptyset \quad \nless
 \end{aligned}$$

8.3. Alulról felfele elemzés

Az **alulról felfele elemzés** a másik lehetséges stratégiája a szintaktikus elemzésnek. A **terminálisokból a startszimbólum felé** építjük a szintaxisfát. A **bemenet feldolgozása** továbbra is **balról jobbra** történik, azonban az elemzés a **legjobboldalibb levezetés inverzét** állítja elő – a legjobboldalibb levezetés azt jelenti, hogy több terminális esetén a legjobboldalibbat helyettesíti.

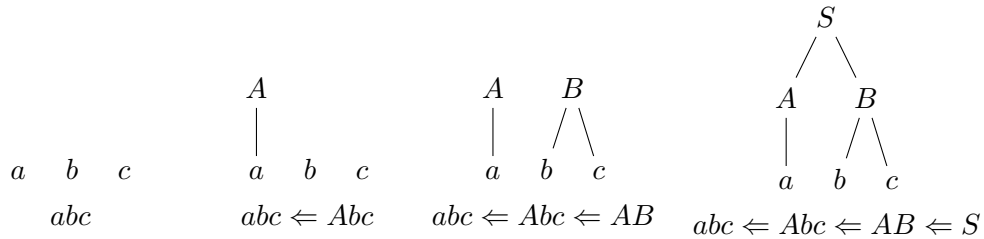
Szemléltessük az korábbi nyelvtanon:

$$S \longrightarrow AB$$

$$A \longrightarrow a$$

$$B \longrightarrow bc.$$

Legyen a bemeneti szövegünk továbbra is: *abc*. A szó szintaxisfáját így kapjuk meg:



8.4. ábra. Alulról felfele elemzés lépései

Az alulról felfele elemzők egyik gyakori változatával, az ún. LR elemzőkkel fogunk megismerkedni. A pontos definícióját a későbbiekben kimondjuk.

Hasonlóan az *LL*-hez, az *LR*-elemzés is **verem alapú**, ám a verem nem futás idejű – tehát a kódban valóban példányosítanunk kell egyet. **Ebben gyűjtjük a szimbólumokat** (terminálisokat és nemterminálisokat egyaránt) egészen addig, amíg a megfelelő szabályjobboldal megjelenik benne.

Tartozik hozzá **két művelet**.

1. Léptetés (*shift* vagy *push*): A következő token elhelyezése a verem tetején.
2. Redukció (*reduce* vagy *pop*): A szabályjobboldal helyettesítése szabálybaloldallal a veremben, közben a szintaxisfa bővítése.

Szemléltessük a műveleteket a következő *balrekurzív* nyelvtanon (ami szintén a vesszővel elválasztott listák nyelét fejezi ki):

$$S \longrightarrow \varepsilon \mid N$$

$$N \longrightarrow e \mid Nve$$

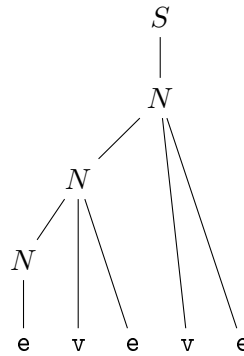
A példaszavunk továbbra is legyen az „eveve”.

Kezdetben a vermünk üres, így előreolvasunk egy karaktert. Betesszük a verembe (**e**) – azaz léptetünk. Ekkor megjelent egy szabályjobboldal ($N \longrightarrow e$), amit kicserélhetünk a bal oldalával (N).

$$\# \xrightarrow{\text{shift}} e \xrightarrow{\text{reduce}} N.$$

Folytatjuk a léptetést. A verem állapota így Nv lesz. Nincs ilyen alakú szabályjobboldal, így újból léptetünk. A veremben Nve lesz. Ez már helyettesíthető szabálybaloldalra ($N \rightarrow Nve$), így redukálunk (verem: N). Kettőt léptetünk (verem: Nv , Nve), majd redukálunk (verem: N). Mivel elfogytak a beolvasandó karaktereink, így tovább redukálhatjuk a verem tartalmát. Az elemzés akkor sikeres, ha csupán az S marad benne a legvégén.

$$\# \xRightarrow{\text{shift}} e \xRightarrow{\text{red.}} N \xRightarrow{\text{shift}} Nv \xRightarrow{\text{shift}} Nve \xRightarrow{\text{red.}} N \xRightarrow{\text{shift}} Nv \xRightarrow{\text{shift}} Nve \xRightarrow{\text{red.}} N \xRightarrow{\text{red.}} S.$$



8.5. ábra. Az LR elemzés által létrehozott szintaxisfa

Ha visszapillantunk a korábbi ábrára, ahol egy léptetés és redukció után az N szerepelt a veremben, észrevehetjük, hogy akár rögtön abban a lépésben is redukálhattuk volna S -re a tartalmát – ezzel kihagyva a szövegünk hátralévő 80%-át.

A korábbiakhoz hasonlóan, felmerülhet a kérdés: **hogyan döntjük el, hogy mikor melyik műveletet végezzük el?** Ennek az eldöntéséhez figyelembe kell vennünk a *következő valahány token* (ami **előreolvasást** jelent), valamint az **elemző állapotát** (ami a verembe bekerülő szimbólumoktól függően változik).

Ezzel el is érkeztünk ahhoz, hogy kimondjuk az $LR(k)$ nyelvtan pontos definícióját.

8.3.1. definíció ($LR(k)$ nyelvtan). Egy környezetfüggetlen nyelvtan $LR(k)$ tulajdonságú valamely $k \in \mathbb{N}$ számra, ha az elemzés pillanatnyi állapotából és legfeljebb k token előreolvasásával egyértelműen meghatározható, hogy léptetés vagy redukció következik, és redukció esetén az alkalmazandó nyelvtani szabály is kiderül.

Az elnevezés a „*left to right using rightmost derivation*” elnevezés angol rövidítéséből származik.

Az $LR(1)$ elemzést fogjuk részletesen megvizsgálni – egyetlen szimbólum előreolvasása elegendő.

A korábbi megállapításaink alapján létre kell hoznunk egy **elemző táblázat**ot, ami meghatározza a lépéseket és az állapotátmeneteket. Mivel az LR elemzés verem alapú, így ez is egy **veremautomata** lesz. Négy **akció** szerepel egy ilyen táblázatban: *léptetés*, *redukció*, *elfogadás* és *hiba*.

	e	v	input vége	N
0	léptetés : 2	hiba	elfogadás	1
1	hiba	léptetés : 3	elfogadás	hiba
2	hiba	redukció : $N \rightarrow e$	redukció : $N \rightarrow e$	hiba
3	léptetés : 4	hiba	hiba	hiba
4	hiba	redukció : $N \rightarrow Nve$	redukció : $N \rightarrow Nve$	hiba

8.6. ábra. A nyelvtanunk $LR(1)$ elemző táblázata

Talán nem meglepő, a nyelvtannak ezen tulajdonságának ellenőrzésére is létezik tétel.

8.3.1. tétel ($LR(1)$ tulajdonság ellenőrzése). *Egy környezetfüggetlen nyelvtan pontosan akkor $LR(1)$ tulajdonságú, ha az elemző táblázatot kitöltő algoritmus konfliktusmentesen kitölti a táblázatot.*

Megjegyzés. A táblázat a nyelvtanból algoritmikusan létrehozható, de nem része a tananyagnak. Állítólag elég bonyolult.

9. fejezet

Szemantikus elemzés

A szintaktikus elemzés létrehozza a szintaxisfát. A szemantikus ellenőrzés azt állapítja meg róla, hogy „van-e értelme” annak, amit kifejez – mindezt fordítási időben.

Nyelvtől függően jelentősen eltérhetnek a specifikus feladatai, így csak általánosságokban fogunk róluk értekezni. A szemantikus elemzés két eszközt használ: ezek a **szimbólum-tábla** és az **attribútumnyelvtan**.

9.1. Szimbólumtábla

A szimbólumtábla a deklarációkat tárolja. A fordítóban gyakran globális változó. Segítségével a szemantikus elemzés

- feldolgozza a deklarációkat,
- az azonosítószimbólumokat deklarációhoz köti,
- ellenőrzi a hatókörrel és láthatósággal kapcsolatos szabályokat.

A következő, **tipikus hibákat** képes kiszűrni:

- deklarálatlan változókat,
- újradeklarált változókat,
- változó hatókörön kívüli használatát,
- privát adattagok elérését (pontosabban azoknak a korlátozását),
- elfedésből adódó típushibákat.

Ahogy azt megtárgyaltuk, a lexikális elemző a forráskód karaktersorozatából tokenek sorozatát állítja elő.

$$x = (x + 2) * 3;$$
$$\downarrow$$
$$\langle x, =, (, x, +, 2,), *, 3, ; \rangle.$$

Arról is beszéltünk, hogy egyes tokenek rendelkezhetnek kiegészítő információkkal (az azonosító a szövegüket, a literálok az értéküket, stb.). Megelőlegezzük, hogy kitüntetett szintetizált attribútumoknak hívjuk őket.

type <i>Kind</i> is {function, parameter, local variable, stb.}
type <i>Type</i> is {int, int→void, stb.}
SymbolTable
+ <i>name</i> : <i>S</i>
+ <i>kind</i> : <i>Kind</i>
+ <i>type</i> : <i>Type</i>
+ <i>declaration</i> : $\mathbb{N} \times \mathbb{N}$ /* (row, column) */
+ <i>used_here</i> : $\mathbb{N} \times \mathbb{N}()$ /* list of coordinates */
/* additional fields may come here */
+ SymbolTable(...)
/* additional methods may come here */

9.1. ábra. A szimbólumtábla egy lehetséges megvalósítása

Egy egyszerűbb nyelv esetében a szimbólumtáblát implementálhatjuk egy **hasító táblával**. Ez olyan összetett típusú objektumokat tartalmaz, amelyek az adott deklarált „egységről” (legyen az változó, függvény, stb.) információkat tárolnak, mint például

- a nevét,
- fajtáját (függvény, függvényparaméter, lokális vagy globális változó, ciklus, elágazás, névtelen blokk, stb.),
- típusát (egész szám, int→void típusú függvény, stb.)
- a deklarációja helyét (a nevének első karaktere az eredeti szövegben melyik sor melyik oszlopában történik),
- mikor használtuk a program során.

Két művelettel is rendelkezik.

1. Beszúrás

- deklaráció esetén
- az új szimbólum és adatai bekerülnek a szimbólumtáblába
- a beszúrás *mindig egy kereséssel kezdődik*, hogy kiderüljön az újradeklarálás

2. Keresés

- szimbólum használatakor
- a szimbólum *neve a kulcs* a kereséshez
- a szimbólum használatát érdemes feljegyezni (pl. refaktoráláshoz)

1 void f(int p) {	Név	Fajta	Típus	Deklaráció	Használat
2 int x;	f	függvény	int→void	(1, 6)	⟨⟩
3 cin >> x;	p	paraméter	int	(1, 12)	⟨(4, 13)⟩
4 cout << x+p+y;	x	lokális változó	int	(2, 7)	⟨(3, 10), (4, 11)⟩
5 }					

9.2. ábra. A kódrészlet és a hozzá tartozó szimbólumtábla

Megjegyzés. Ha a korábbi kódrészletben a függvény végére beillesztenénk a(z)

```
int x;
```

sort, akkor a változót nem tudná beszúrni a táblába, hiszen szerepel már egy ilyen nevű és típusú változó. Ilyenkor a compiler újradeklarálás hibájával fog visszajelezni.

Jobban járunk, ha hasító tábla helyett **verem adatszerkezetet** használunk a szimbólumtáblához. Ez sokkal jobb megoldás, ugyanis neki köszönhetően képesek vagyunk kezelni a **blokkszerkezeteket**, a változók **hatókörét**, **láthatóságát**, valamint az **elfedést** is!

A beszúrás lecserélődik egy klasszikus *push()* műveletre. Keresékor fentről lefelé keresünk a veremben, és az *első találatnál* megállunk.

Továbbá felvesszünk egy ún. **blokk-index vektort**, ami a nevével ellentétben szintén egy **verem**. Ennek az a feladata, hogy **amikor új blokk kezdődik, a blokk-index vektorban megjelöljük a szimbólumtábla-verem tetejét** – magyarul egy olyan pointert push-olunk bele, ami az adott „blokkelemre” mutat a szimbólumtáblában¹. Függvényhívás esetén megfeleltethető a függvény aktivációs rekordjának. **Minden blokk megkezdésekor új mutató kerül a vektor tetejére.** A blokk végén eltávolítjuk a szimbólumokat a blokk-index vektor legfelső jelöléséig (néhány *pop()*). Végül a jelölést is eltávolítjuk a blokk-index vektorból.

```
1 int x = 1;
2 void f(int p) { // <- semantic analysis in progress
3     cout << x;
4     int x = 2;
5     cout << x+p;
6 }
7 int p = x;
```

⋮	⋮	⋮	⋮	
p	paraméter	int	(2, 12)	
f	függvény	int→void	(2, 6)	
x	globális változó	int	(1, 5)	&(f)
Név	Fajta	Típus	Deklaráció	Blokk-index vektor

9.3. ábra. Verem szerkezetű szimbólumtábla

Deklaráció feldolgozásakor ellenőrizni kell, hogy nem újradeklarált változóról van-e szó. Csak ez után szabad beszúrni a szimbólumot és adatait a táblázatba. A verem szerkezetű szimbólumtáblában **csak a blokk-index vektor legfelső bejegyzése által mutatott rekord fölött keresünk**, azaz az aktuális blokk szimbólumai között. Ha a blokk-index vektor üres, akkor az egész szimbólumtáblában keresünk. Ha nincs hiba, a szimbólum beszúrható a táblába.

9.2. Attribútumnyelvtan

A típusrendszerek a programhibák felderítésének legfontosabb eszközei. Ezek jelölik ki, milyen műveletek végezhetők az adatokkal. Vannak a jól ismert alaptípusaink (bool, int,

¹A 9.3. ábrában a & jel jelöli az f nevű függvényhez tartozó szimbólumobjektum memóriacímét.

char, stb.), de a nyelv támogathat összetett típusokat is (tömb, rekord, mutató, referencia, osztály, interfész, unió, algebrai típusok, stb.). Ugyanakkor léteznek típus nélküli nyelvek is – ilyen a legtöbb Assembly nyelv.

Az **ellenőrzés két fázisban** történik.

1. Statikus típusozás – fordítási időben

- Futás közben már **csak az értékeket kell tárolni**, típusinformációt nem
- Ha a program lefordul, típusokkal kapcsolatos hiba nem történhet futás közben: **biztonságosabb** megoldás
- Ada, C++, Haskell, stb.

2. Dinamikus típusozás – futási időben

- Futási időben az értéket mellett **típusokat is kell tárolni**
- Az utasítások **végrehajtása előtt kell ellenőrizni** a típusokat
- Futás közben derülnek ki a típushibák, cserébe **hajlékonyabbak** az ilyen nyelvek
- Lisp, Erlang, stb.

A statikusan típusos nyelvek is használnak dinamikus technikákat: dinamikus kötés, Java `instanceof` operátora.

A típusokat is kétféleképpen adhatjuk meg.

1. Programozó adja meg \rightarrow típusellenőrzés

- A deklarációk típusozottak
- A kifejezések egyszerű szabályok alapján típusozhatók
- Egyszerűbb fordítóprogram, gyorsabb fordítás

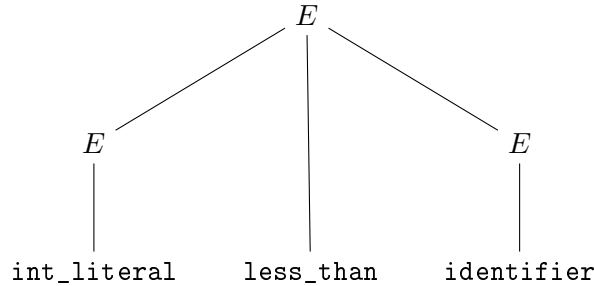
2. Fordítóprogram találja ki \rightarrow típuslevezetés, típuskikövetkeztetés

- A deklarációkhoz (általában) nem kell típust megadni
- A kifejezések típusát a fordítóprogram „találja ki” a műveletek alapján
- Kényelmesebb a programozónak – azonban ajánlott típusozni a deklarációkat, hogy olvashatóbb legyen a kód

A **típuskonverzió** a kifejezés típusának megváltoztatását jelenti. Ez történhet automatikusan vagy explicit konverzióval is (C/C++-ban ez a kasztolás). A fordítóprogramnak ügyelnie kell az osztályhierarchiához kapcsolódó típuskonverziókra – azaz a *Liskov-féle helyettesítési elvre*. A típuskonverziókkal a kódgenerátornak is törődnie kell: adatkonverzióra is szükség lehet.

A szintaxist leíró nyelvtan szimbólumaihoz **attribútumokat** rendelünk, melyek a szemantikus elemzés vagy a kódgenerálás, kódoptimalizálás számára fontos, kiegészítő információk. Az ilyen nyelvtant nevezzük **attribútumnyelvtannak**. A szabályokhoz **akciókat** (programkód részleteket) rendelünk. Ezek a meglévő attribútumértékekből újabb attribútumok értékeit számolják ki, valamint ellenőrzéseket végeznek, szemantikus hibákat jeleznek.

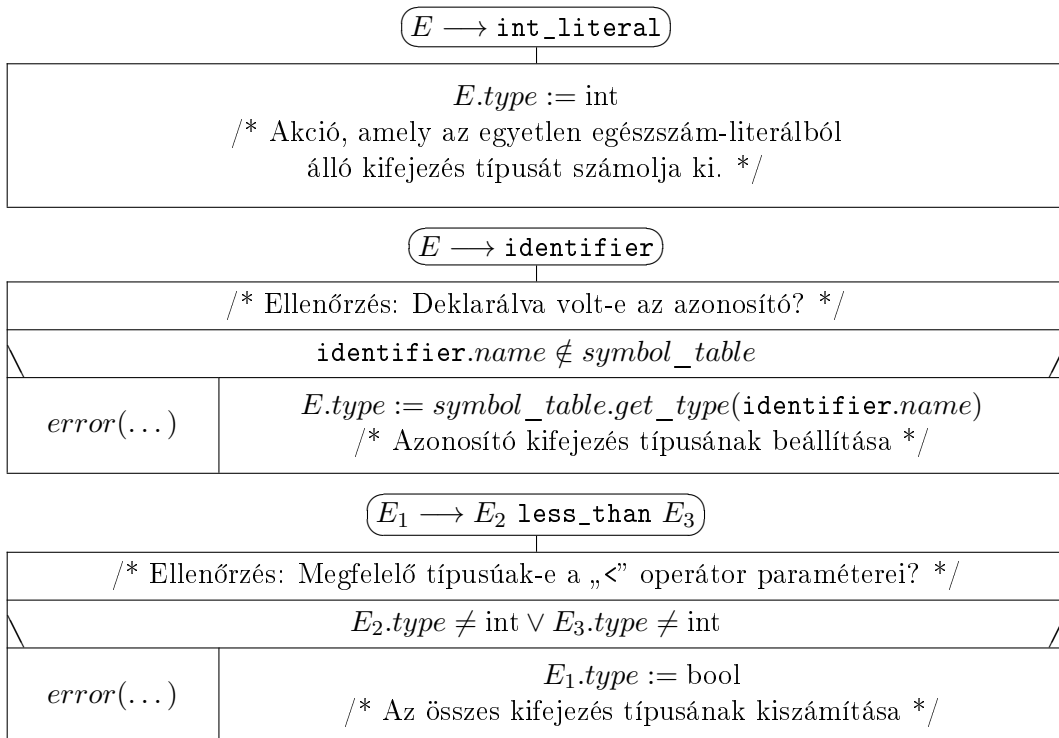
Legyen az elemzendő szövegünk a(z) `10 < x` az alábbi szintaxisfával
(a nyelvtan: $E \rightarrow \text{int_literal} \mid \text{identifier} \mid E \text{ less_than } E$).



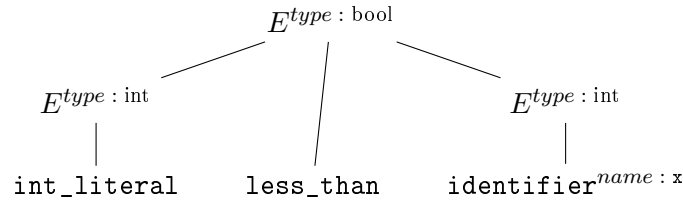
A következő attribútumokra lesz szükségünk.

- Az azonosítóknak a nevére, amit `identifier.name` formában érhetünk el (lásd a 9.1. ábrát). Ezek a **kitüntetett szintetizált attribútumok**.
- A kifejezésekhez is hozzárendelünk (egész pontosan a *szabály bal oldalához*) egy `type : Type` attribútumot, amit `E.type` formában érhetünk el. Az ilyen attribútumokat **szintetizált attribútumok**nak nevezzük. Az *LR* elemzőkhöz nagyon jól illeszkednek.

Az elemző **alulról felfele** haladva meghatározza először a literálok típusát. A felsőbb szintekhez érve a korábbiak alapján meghatározza az egész kifejezés típusát is az **akciók** segítségével.



Az újonnan megállapított szintetikus attribútumok így bekerülnek a szintaxisfába.



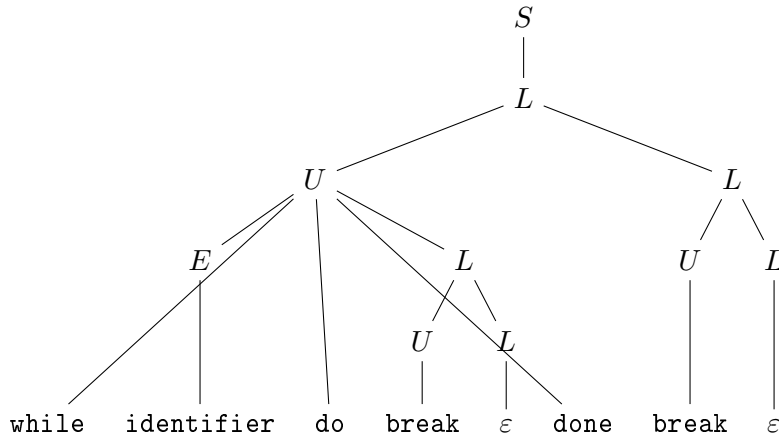
Vegyünk egy másik példát, amiben egy *újabb attribútumfajtáról* lesz szó. A **ciklus** működését, valamint az abból való kiugrást (**break** utasítás) szemlélteti. Legyen a mondatunk

while b do break done break,

aminek a nyelvtana

$$\begin{aligned}
 S &\longrightarrow L \\
 L &\longrightarrow \varepsilon \mid UL \\
 U &\longrightarrow \text{break} \mid \text{while } E \text{ do } L \text{ done} \\
 E &\longrightarrow \text{identifier}.
 \end{aligned}$$

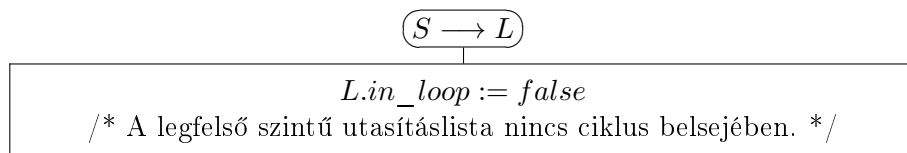
A szintaxisfája nyilvánvalóan

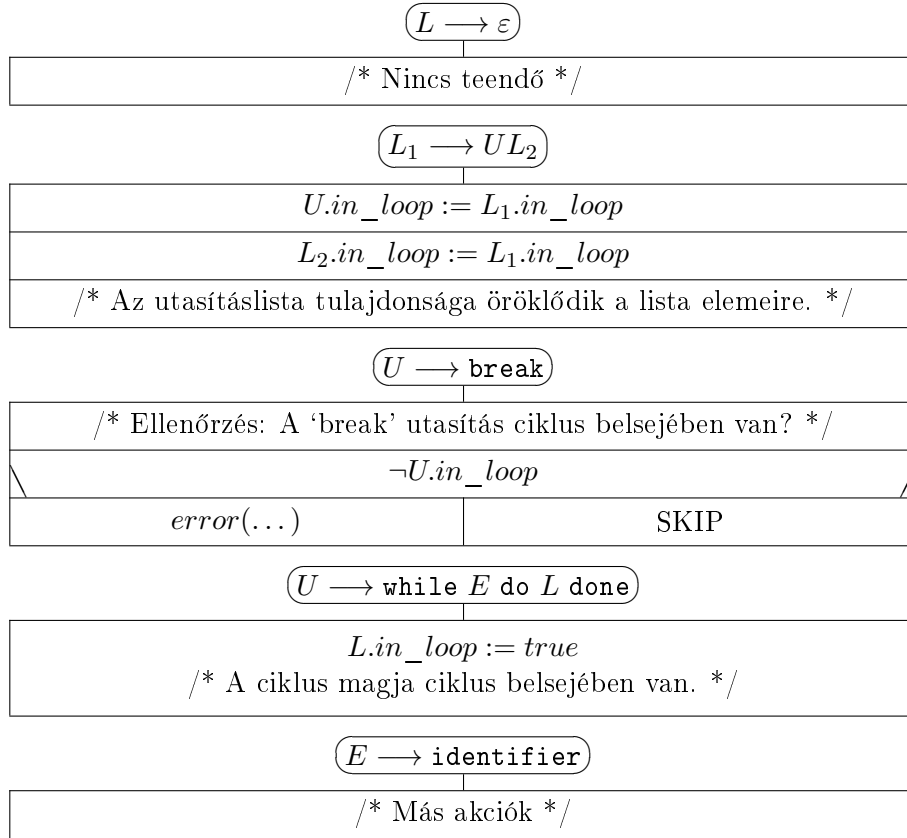


Felvesszük a következő attribútumokat.

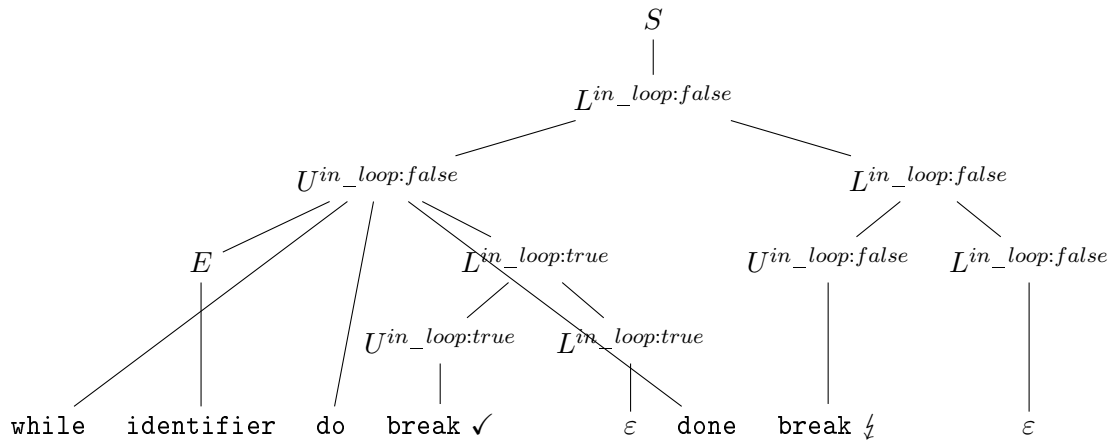
- Az L nemterminális szimbólumoknak (mint *loop*) egy $in_loop : \mathbb{B}$ attribútumot.
- Az U szimbólumoknak (mint *utasításlista*) szintén egy $in_loop : \mathbb{B}$ attribútumot.

Mindkettő esetben a *szabály jobb oldalán* állnak, amikor kiszámítjuk őket. Ugyanakkor fontos különbség, hogy **felülről lefelé** közvetít információt a szintaxisfában. Az ilyen attribútumokat **örökölt attribútumoknak** nevezzük, mivel a gyökerénél meghatározzuk ezt a tulajdonságot, ami az elemzés során „leszivárog” az alsóbb szintekre. Az egyes nyelvtani szabályainkhoz az alábbi **akciókat** rendeljük hozzá.





Az attribútumokkal ellátott fa pedig:



Ahogy láthatjuk, az első **break** utasítást helyesen használtuk, hiszen cikluson belül helyezkedik el, míg a második nem. Emiatt szemantikai hibát fog jelezni az elemző.

```

1  /*
2     Nonterminal 'expression' has type 'type'.
3  */
4  %type <type> expression
5
6  expression:
7  expression LESS_THAN expression
8  {
9      /*
10       $1, $2, $3, ... refer to
11       the given attributes of the RHS of the rule.
12      */
13      if ($1 != int || $3 != int)
14          error(...);
15      else
16          /* $$: the attribute of the LHS of the rule. */
17          $$ = bool;
18  }

```

9.4. ábra. Példa a Bison szemantikus elemzőjére

A gyakorlatokon a Bison fordítógenerátort használjuk, aminek szemantikus elemzője ki-tüntetett szintetizált és szintetizált attribútumokat támogat, azonban örökölteket nem². Emiatt jól illeszkedik az *LR* elemzéshez, hiszen egy lépésben elvézni a szintaktikus és szemantikus elemzést.

²Az ilyen nyelvtanokat *S*-attribútumnyelvtanoknak nevezzük.

10. fejezet

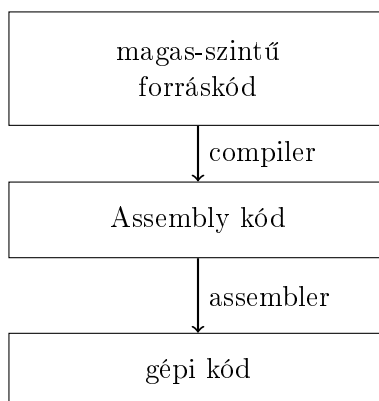
Az Assembly alapjai

Az **Assembly** egy *alacsony szintű programozási nyelv*, amely segít áthidalni a magas szintű nyelvek és a gépi kód közti hatalmas „szakadékot”. Valójában a gépi kódnak egy ember számára olvashatóbb változatáról van szó, ami a rejtélyes hexadecimális számok helyett korlátozott számú rövid, tömör nevű **műveleteket**, valamint **regisztereket** használ. Megengedi a programozó számára, hogy a memóriacímek azonosítására **címkéket** használjunk.

Az Assembly nem egy egységes nyelv, hanem inkább egy nyelvcsalád, aminek a nyelvei architektúránként eltér. Azonban vannak közös jellemzői. Fordítóprogramját **Assemblernek** nevezzük. A tanulmányai folyamán a **32-bites, x86-os architektúrájú**, NASM szintaxisú Assemblyvel fogunk foglalkozni.

1	<code>int sum = 0;</code>	1	<code>mov ecx,0</code>	1	<code>B9 00 00 00 00</code>
2	<code>for (int i = 0;</code>	2	<code>mov eax,0</code>	2	<code>B8 00 00 00 00</code>
3	<code> i < len;</code>	3	<code>eleje:</code>	3	<code>81 F9 0A 00 00 00</code>
4	<code> ++i)</code>	4	<code>cmp ecx,10</code>	4	<code>7D 06</code>
5	<code>{</code>	5	<code>jge vege</code>	5	<code>03 04 8B</code>
6	<code> sum += t[i];</code>	6	<code>add eax,[ebx+4*ecx]</code>	6	<code>41</code>
7	<code>}</code>	7	<code>inc ecx</code>	7	<code>EB F2</code>
		8	<code>jmp eleje</code>		
		9	<code>vege:</code>		

10.1. ábra. C++ kód, Assembly kód és gépi kód



10.2. ábra. A forráskód állapotának szakaszai

10.1. Az Assembly forrásfájl

As Assembly forrásfájl jellemző kiterjesztése a(z) `*.asm`, gyakran használt fordítója a `nasm`. Adott a következő Assembly kódrészlet.

```

1 global main
2 extern write_natural
3 extern read_natural
4
5 section .text
6 main:
7 call read_natural
8 % increment
9 inc eax
10 push eax
11 call write_natural
12 add esp,4
13 % move
14 mov eax,0
15 % return
16 ret

```

addone.asm

- `global main` : Definiáljuk a `main` címkét, ami a C/C++-ban a `main`-függvénynek feleltethető meg. Egy globálisan látható címkéről van szó.
- `extern <címke>` : Olyan címkét határozunk meg vele, melyek máshol vannak definiálva, de itt szeretnénk őket használni.
Összevethető mondjuk egy C-s `include`-olással, bár itt nincsenek `include guard`-ok, és magunknak kell gondoskodnunk fordításnál, hogy valóban ne hiányozzanak a definíciójuk.
- `section .text` : Itt kezdődik a programkód.
- `<címke>:` : A címke által jelölt programrésznek a kezdetét jelzi. Jelen esetben ez a `main`-függvény lesz.
- `call <címke>` : „Meghívjuk a `<címke>` függvényt”, valójában **ugróutasítás**, ami arra a kódrészre ugrik, ahova a címke mutat.
 - Eltárolja a **stacken** (futásidejű verem típusú tárhely), hogy honnan ugrottunk és paraméterül azt adjuk meg, hova ugrunk.
 - Egyes kódoptimalizálók csinálhatják azt, hogy ugrás helyett bemásolják a szóban forgó kódrészletet.
- `inc <regiszter>` : Növeli, inkrementálja az adott regiszteren lévő értéket.
A kód esetében az `eax`-ben tároljuk el a korábbi függvényhívás visszatérési értékét.
- `push <regiszter>` : A stack tetejére rakja a regiszter értékét: így lehet paraméterként átadni.
- Meghívjuk a `write_natural` függvényt, ami kiírja a paraméterként kapott számot.

- `add <mihez>, <mit>` : Összeadja a két értéket. Az eredményt a `<mihez>` regiszterben tárolja el. Részletesebben később.
Itt: az imént a verembe tett paramétert kitöröljük onnan.
- `mov <hová>, <honnan>` : Átmozgatjuk a `<honnan>` tartalmát a `<hová>`-ba. Részletesebben később.
Itt: beállítjuk a main-függvény visszatérési értékét (0).
- `ret` : Visszaugrik oda, ahonnan a `call`-ból érkeztünk. Ez úgy történik, hogy a verem tetejéről leveszi a pointert és arra ugrik vissza.
 - Honnan tudjuk, hogy a veremben melyik aktivációs rekord, melyik lokális változó?
 - Sehonnan, számon kell tartanunk, hány változónk van és annyit kivenni, vagy esetleg **offset**tel mozgatni.

11. fejezet

Kódgenerálás

