



**EÖTVÖS LORÁND TUDOMÁNYEGYETEM**

**INFORMATIKAI KAR**

**Algoritmusok és Alkalmazásaik Tanszék**

---

# **Interpreter készítése strukturált programozott nyelvtanokhoz**

**BSc szakdolgozat**

**Sebestyén Balázs**

**Programtervező Informatikus hallgató**

**Témavezető:**

**Dr. Hunyadvári László**

**Tanszékvezető egyetemi docens**

**Budapest, 2010**

## Tartalom

1	Bevezető.....	5
1.1	Programozott nyelvtanok.....	6
2	Felhasználói dokumentáció.....	10
2.1	Munkalapok .....	10
2.1.1	Táblázat - Nyelvtan.....	10
2.1.2	Táblázat - Programozott nyelvtan.....	12
2.1.3	Program – Programozott nyelvtan.....	12
2.1.4	Program – Turing-gép, Veremautomata .....	13
2.2	Beállítások .....	14
2.2.1	Általános.....	14
2.2.2	Programozott nyelvtan beállításai .....	15
2.2.3	Turing-gép, Veremautomata.....	15
2.2.4	Könyvtárak .....	16
2.2.5	Stílusok .....	16
2.3	A Prolan nyelv .....	16
2.3.1	Hello World! .....	16
2.3.2	Fordítási direktívák, kommentek .....	18
2.3.3	Típusok .....	19
2.3.4	Deklarációk.....	19
2.3.5	Beépített konstansok .....	21
2.3.6	Logikai kifejezések.....	21
2.3.7	Utasítások.....	22
2.3.8	Blokkok.....	23
2.3.9	Makrók .....	24
2.4	Speciális nyelvtanok, automaták programozása.....	25
2.4.1	Közönséges nyelvtanok .....	25
2.4.2	Mátrixnyelvtanok .....	26
2.4.3	Lindenmayer-rendszerek .....	27
2.4.4	Automaták.....	28
2.4.5	Turing-gép .....	30
2.4.6	Véges automata .....	32
2.4.7	Veremautomata .....	33
3	Fejlesztői dokumentáció .....	35

3.1	Fogalmak, algoritmusok .....	35
3.1.1	Programozott automata.....	35
3.1.2	Programozott automata redukálása .....	39
3.1.3	Turing gép fordítása .....	44
3.1.4	Veremautomaták fordítása .....	48
3.1.5	Futtatás .....	48
3.2	Implementáció .....	49
3.2.1	Felhasznált technológiák.....	49
3.2.2	Fájlok .....	50
3.2.3	Telepítés .....	51
3.2.4	Előfordítás .....	51
3.2.5	Lexikális, szintaktikus elemző.....	52
3.3	Osztályhierarchia.....	52
3.3.1	A szintaxisfa osztályai.....	53
3.3.2	Programozott automata.....	57
3.3.3	Konfigurációk .....	62
3.3.4	Programozott nyelvtan .....	64
3.3.5	Turing-gép és veremautomata.....	65
3.3.6	A felhasználói felület.....	69
3.4	Tesztelés.....	77
4	Irodalomjegyzék.....	79

## 1 Bevezető

A szakdolgozat eredeti célja egy programozott nyelvtanokat demonstráló felhasználói program létrehozása volt, egy készülő formális nyelvek és automaták jegyzet számára, annak mellékleteként.

A programozott nyelvtanok (továbbiakban PNY) a közönséges generatív nyelvtanok fogalmának egy természetes kiterjesztése, mely a nyelvtan levezetéseit a programok futásához teszi hasonlatossá (ld. 1.1 fejezet). A témakör egyik érdekes területe a strukturált PNY, melynek lényege, hogy az egyes rutinfeladatok, pl. aritmetikai műveletek számára makrókat készítünk, majd ezek ismételt felhasználásával konstruáljuk meg a végleges nyelvtant. Ez az eljárás tekinthető egyfajta strukturált programozásnak, melyben a különböző nyelvtanok létrehozása mint programozási feladat jelenik meg.

A PNY makrók használata közben tapasztalhatjuk, hogy fáradtságos velük dolgozni, a makrók nehezen áttekinthetőek, nehezen módosíthatóak. Ezek a problémák mind emlékeztetnek az assemblyben való programozás nehézségeire, aminek oka a leíró nyelv minimalizmusában rejlik. A megoldás kézenfekvő: használjunk valamilyen magasabb szintű nyelvet, programszerkezetekkel, típusokkal, operátorokkal. A felhasználó ezen a nyelven (Prolan) definiálja a nyelvtant, melynek végső részleteit a fordítóprogram határozza meg. A program egymás mellett jeleníti meg a forráskódot, a belőle lefordított nyelvtant, valamint a nyelvtan által generált levezetéseket ill. nyelvet.

Miután elkészült a fenti futtatási környezet, felmerült, alkalmas-e Turing-gép modellezésére, így jóval megnövelve a program használati értékét az eredeti célkitűzésekhez képest. A válasz igen, hiszen másodrendű programozott nyelvtanokkal elvileg tetszőleges egyéb speciális nyelvtan vagy automata szimulálható (mivel ez a nyelvtanosztály Turing-teljes). Azonban ezek a szimulációk általában csak elméleti jelentőségűek, a gyakorlatban főlegesen sok nyelvtani szabályt, és áttekinthetetlenül bonyolult működést eredményeznek (egy egyszerű Turing-gépet is csak egy bonyolult PNY-al lehet leírni). Ehelyett, mint kiderült: a nyelv és az interpreter könnyen kiterjeszthető úgy, hogy „natív módon” támogassa a nem determinisztikus Turing-gépeket és veremautomatákat is, vagyis a felhasználó közvetlenül dolgozhat Turing-gépekkel, anélkül hogy a PNY-okat ismernie kellene. Ezek segítségével pedig már lefedhető a formális nyelvek és a számításelmélet több területe az említett bonyolultság-növekedés nélkül. A program tehát a következőket támogatja:

- Programozott nyelvtanok
- Nemdeterminisztikus Turing-gépek
- Veremautomaták
- Közönséges nyelvtanok (PNY segítségével)
- Mátrixnyelvtanok (PNY segítségével)
- Párhuzamos nyelvtanok (PNY segítségével)
- Végtes automaták (T-gép segítségével)

Egy további érv a feladat fenti, általánosabb megoldása mellett: A felhasználói célcsoport tagjai (formális nyelveket tanuló hallgatók) esetleg írhatnak maguknak determinisztikus Turing-gépet valamilyen általános célú programnyelven, ehhez főlegesen számukra egy külön nyelv

megtanulása. A nemdeterminisztikus változatok esetében azonban egészen más a helyzet, ezekhez már érdekesebb egy céleszközt választani, valamint ez a terület igényli leginkább, hogy a megértést valamilyen demonstrációs eszközzel segítsük.

Az informatika oldaláról megközelítve: a Prolan funkcióját tekintve a Logohoz hasonló, egyszerű, oktatási célra szánt nyelv, szintaktikai szempontból pedig a C nyelv az alapja. Sosem produkál futásidejű hibát, maga a futás eredménye és a „nyomkövetés” a fordítás után azonnal megjelenik, megkönnyítve ezzel a hibajavítást és a program működésének áttekintését. Az elkészült programok lehetnek nemdeterminisztikusak is, melyekkel amúgy egy kezdő programozó ritkán találkozna. Ezen tulajdonságai miatt esetleg hasznos lehet a programozás-módszertan oktatásában is, különösen, ha az párhuzamosan történik a számításelmélet és formális nyelvekével.

## 1.1 Programozott nyelvtanok

A közönséges nyelvtanok esetén a startszimbólumból kiindulva tetszőleges sorrendben alkalmazhatjuk a nyelvtan szabályait. Programozott nyelvtanok esetén erre a sorrendre bizonyos megkötéseket teszünk. A pontos matematikai definíciók előtt nézzünk egy egyszerű példát programozott nyelvtanra, mely az  $\{a^{2^n} | n \in \mathbb{N}\}$  nyelvet generálja. A megadás általában táblázatos formában történik:

### Példa

címke	szabály	$\sigma$	$\varphi$
$f_1$	$S \rightarrow ZZ$	$f_1$	$f_2$
$f_2$	$Z \rightarrow S$	$f_2$	$f_1, f_3$
$f_3$	$S \rightarrow a$	$f_3$	$exit$

A levezetés mondatformája kezdetben az  $S$  startszimbólumot tartalmazza. Az első lépésben a nyelvtan tetszőleges szabályát alkalmazhatjuk, hasonlóan a közönséges nyelvtanokhoz. Viszont ettől kezdve minden szabály végrehajtása után, attól függően hogy a szabály illeszthető volt-e az aktuális mondatformára, egy címkét választunk a  $\sigma$  vagy pedig a  $\varphi$  oszlopból, és a kiválasztott címkéhez tartozó szabállyal kell folytatnunk a végrehajtást. Egy lehetséges levezetés (a konfigurációk alakja:  $[mondatforma, címke]$ ):

$$[S, f_1] \rightarrow [ZZ, f_1] \rightarrow [ZZ, f_2] \rightarrow [SZ, f_2] \rightarrow [SS, f_3] \rightarrow [Sa, f_3] \rightarrow [aa, f_3]$$

### Magyarázat

1.  $[S, f_1]$ . Kezdetben a mondatforma  $S$ . Válasszunk egy tetszőleges címkét, legyen ez  $f_1$ . Az  $f_1$  címkéjű  $S \rightarrow ZZ$  szabály illeszthető  $S$ -re, eredményül kapjuk a  $ZZ$  mondatformát. Mivel a szabály alkalmazható volt, a következő lépés címkéjét a  $\sigma$  oszlopból választjuk, vagyis ismét  $f_1$ -et, mivel az oszlop egyedül ezt tartalmazta.
2.  $[ZZ, f_1]$ . A mondatforma  $ZZ$ , amire  $f_1$  szabályát kell megpróbálnunk illeszteni. Ez nem lehetséges, ezért most a  $\varphi$  oszlopból kell választanunk, amire az egyetlen lehetőség az  $f_2$  címke.
3.  $[ZZ, f_2]$ . A mondatforma maradt  $ZZ$ , amire illeszthető az  $f_2$  címkéjű  $Z \rightarrow S$  szabály. Az új mondatforma  $SZ$  (választhattuk volna  $ZS$ -t is), címkét a  $\sigma$  oszlopból kell választanunk:  $f_3$ . S í.t..

4.  $[aa, f_3]$ . Az utolsó lépésben a mondatforma csupa terminális jelből áll, ezért ekkor kaptunk egy a nyelvtan által levezethető szót, a levezetést nem folytatjuk.  
Általánosabban: a levezetés akkor ér véget, ha a mondatforma kizárólag terminális jeleket tartalmaz, vagy akkor, ha a speciális *exit* címkét választottuk.

A fent bemutatott példa valójában egy előfordulásellenőrzéses PNY, ami annyit jelent, hogy a nyelvtan felkészül arra az esetre is, amikor a szabály nem illeszthető (erre szolgál a  $\varphi$  oszlop). Az előfordulásellenőrzés nélküli változat csak annyiban különbözik ettől, hogy a táblázat nem tartalmaz  $\varphi$  oszlopot, ezért ha a levezetésben az aktuális szabály nem illeszthető a mondatformára, akkor a levezetés egyszerűen véget ér, nem folytatható. Ez megfelel annak, mintha egy előfordulásellenőrzéses PNY  $\varphi$  oszlopába mindenhol az *exit* címkét írnánk. Az egyszerűség kedvéért a dolgozat többi részében, valamint a programban nem különböztetem meg e két változatot, hanem egységesen az általánosabb, előfordulásellenőrzéses változatot nevezem PNY-nak. Az alábbiakban ismertetem a PNY-ok és az általuk generált nyelv definícióját (1).

**Definíció:** Egy  $G = \langle T, N, P, S, F, lab, \sigma \rangle$  hetest *programozott nyelvtannak* nevezünk, ha a következők teljesülnek.  $\langle T, N, P, S \rangle$  egy nyelvtan, ekkor  $T$ -t,  $N$ -et,  $P$ -t és  $S$ -et rendre a programozott nyelvtan ábécéjének, a nyelvtani jelek ábécéjének, szabályhalmazának illetve kezdőszimbólumának nevezzük.  $F$  ( $exit \notin F$ ) egy véges halmaz, a címkék halmaza.  $lab: F \rightarrow P$  mindenütt értelmezett címkéző függvény. Továbbá  $\sigma: F \rightarrow 2^{F \cup \{exit\}}$  mindenütt értelmezett rákövetkezési függvény.

**Definíció:** Legyen  $\alpha \in (T \cup N)^*$  és  $f \in F \cup \{exit\}$ , ekkor  $[\alpha, f]$ -et a programozott nyelvtan egy konfigurációjának nevezzük.

Legyen  $P = p \rightarrow q$  egy szabály, ekkor vezessük be a következő jelöléseket a szabály bal illetve jobboldalára:  $bo(P) := p$ ,  $jo(P) := q$ .

**Definíció:** Azt mondjuk, hogy az  $[\alpha, f]$  konfigurációból közvetlen (1-lépéses) konfigurációátmenettel levezethető a  $[\beta, f']$  konfiguráció ( $\alpha, \beta \in (T \cup N)^*$ ,  $f \in F$ ,  $f' \in F \cup \{exit\}$ ), ha létezik olyan  $\gamma_1, \gamma_2 \in (T \cup N)^*$ , melyre  $\alpha = \gamma_1 bo(lab(f))\gamma_2$ ,  $\beta = \gamma_1 jo(lab(f))\gamma_2$  és  $f' \in \sigma(f)$ . A közvetlen konfigurációátmenet jelölése:  $[\alpha, f] \rightarrow [\beta, f']$ .

**Definíció:** A közvetett konfigurációátmenet a közvetlen konfigurációátmenet reflexív, tranzitív lezártja, jelölése  $\rightarrow_G^*$ . A levezetés (azaz a közvetlen konfigurációátmenet) hosszán a levezetés során alkalmazott közvetlen konfigurációátmenetek  $n \in \mathbb{N}$  számát értjük.  $n$  lépéses levezetés jelölése  $\rightarrow_G^n$ .

**Definíció:** A  $G$  programozott nyelvtan által generált nyelv:

$$L(G) = \{u \in T^* \mid \exists f \in F, f' \in F \cup \{exit\}, [S, f] \rightarrow_G^* [u, f']\}$$

**Definíció:** Egy  $G = \langle T, N, P, S, F, lab, \sigma \rangle$  programozott nyelvtan  $i$ . típusú, ha  $\langle T, N, P, S \rangle$  a lehető legáltalánosabb értelemben vett  $i$ . típusú nyelvtan.

**Példa:** Az alábbi PNY generált nyelv:  $L(G) = \{uu|u \in T^*\}$

$f_0$	$S \rightarrow ABC$	$f_1, f_4$
$f_1$	$A \rightarrow aA$	$f_2$
$f_2$	$B \rightarrow bB$	$f_3$
$f_3$	$C \rightarrow cC$	$f_1, f_4$
$f_4$	$A \rightarrow a$	$f_5$
$f_5$	$B \rightarrow b$	$f_6$
$f_6$	$C \rightarrow c$	$exit$

**Definíció:** Egy  $G = \langle T, N, P, S, F, lab, \sigma, \varphi \rangle$  nyolcast előfordulásellenőrzés programozott nyelvtannak nevezünk, ha a következők teljesülnek.  $\langle T, N, P, S, F, lab, \sigma \rangle$  egy programozott nyelvtan, ekkor  $T$ -t,  $N$ -et,  $P$ -t és  $S$ -et,  $F$ -et,  $\sigma$ -t rendre az előfordulásellenőrzés programozott nyelvtan ábécéjének, a nyelvtani jelek ábécéjének, szabályhalmazának, kezdőszimbólumának, címkéalmazának, címkéző függvényének illetve pozitív rákövetkezési függvényének nevezzük. Továbbá  $\varphi: F \rightarrow 2^{F \cup \{exit\}}$  mindenütt értelmezett függvény, melyet negatív rákövetkezési függvénynek hívunk.

**Definíció:** Legyen  $\alpha \in (T \cup N)^*$  és  $f \in F \cup \{exit\}$ , ekkor  $[\alpha, f]$ -et az előfordulásellenőrzés programozott nyelvtan egy konfigurációjának nevezzük.

**Definíció:** Azt mondjuk, hogy az  $[\alpha, f]$  konfigurációból közvetlenül (1-lépéses) konfigurációátmenettel levezethető a  $[\beta, f']$  konfiguráció, jelölése:  $[\alpha, f] \xrightarrow{G} [\beta, f']$  ( $\alpha, \beta \in (T \cup N)^*, f \in F, f' \in F \cup \{exit\}$ ), ha  $[\alpha, f] \xrightarrow{G'} [\beta, f']$ , ahol  $G' = \langle T, N, P, S, F, lab, \sigma \rangle$  az előfordulásellenőrzés nélküli programozott nyelvtan.  $[\alpha, f]$ -ből közvetlen (1-lépéses) konfigurációátmenettel előfordulásellenőrzéssel levezethető  $[\beta, f']$ , jelölése  $[\alpha, f] \xrightarrow{G, ac} [\beta, f']$ , ha  $[\alpha, f] \xrightarrow{G} [\beta, f']$  vagy  $bo(lab(f)) \not\subseteq \alpha$ ,  $\alpha = \beta$  és  $g \in \varphi(f)$ .

**Definíció:** A(z előfordulásellenőrzés) közvetett konfigurációátmenet a közvetlen konfigurációátmenet reflexív, tranzitív lezártja, jelölése  $\rightarrow_{G, (ac)}^*$ . A levezetés (azaz a közvetlen konfigurációátmenet) hossza a levezetés során alkalmazott közvetlen konfigurációátmenetek  $n \in \mathbb{N}$  számát értjük.  $n$  lépéses levezetés jelölése  $\rightarrow_{G, (ac)}^n$ .

**Definíció:** A  $G$  programozott nyelvtan által generált nyelv:

$$L^{(ac)}(G) = \{u \in T^* | \exists f \in F, f' \in F \cup \{exit\}, [S, f] \xrightarrow{G, (ac)}^* [u, f']\}$$

**Definíció:** Egy  $G = \langle T, N, P, S, F, lab, \sigma, \varphi \rangle$  előfordulásellenőrzés programozott nyelvtan i. típusú, ha  $\langle T, N, P, S \rangle$  a lehető legáltalánosabb értelemben vett i. típusú nyelvtan.

Az előfordulásellenőrzés változatra példát a fejezet elején láthattunk. A programozott nyelvtanok számítási képességeinek illusztrálására álljon itt két tételt bizonyítás nélkül:

Jelölje  $\mathcal{G}_i^{P, ac}$  az i. típusú előfordulásellenőrzés programozott nyelvtanok összességét. Legyen továbbá  $P_i^{ac} := \{L | \exists G \in \mathcal{G}_i^{P, ac} \text{ és } L^{(ac)}(G) = L\}$

**Tétel:**  $\mathcal{P}_3^{ac} = \mathcal{L}_3$ . Vagyis a harmadrendű előfordulásellenőrzéses programozott nyelvtanok által generált nyelvek halmaza megegyezik a harmadrendű nyelvekével. Tehát ebben az esetben a „programozottság” nem jelent semmilyen előnyt.

**Tétel:**  $\mathcal{P}_2^{ac} = \mathcal{L}_0$ . Vagyis a másodrendű előfordulásellenőrzéses programozott nyelvtanok által generált nyelvek halmaza megegyezik a nulladrendű nyelvekével, tehát ez a nyelvtanosztály már Turing-teljes.



## 2 Felhasználói dokumentáció

A program egy 32 bites Windows alkalmazás, mely Delphi 2007 fejlesztői környezetben készült. Windows 7 operációs rendszer és 1280\*1024-es képernyőfelbontás ajánlott (a felhasználói felület erre lett optimalizálva), de működik Windows XP rendszeren is. Általában a program memóriaigénye 10-20 MB, bizonyos ritka esetekben azonban felmehet több száz MB-ra is, ilyenkor ajánlott egyszerre csak kevés munkalap használata.

A program külön telepítést nem igényel.

### 2.1 Munkalapok

A felhasználói felület munkalapokra osztott. Új munkalapot a Fájl/Új menüvel adhatunk hozzá. Az aktuális munkalapot bezárhatjuk Ctrl+F4 billentyűkombinációval, vagy a Fájl/Bezárás menüponttal. A Fájl/Mind bezárása értelemszerűen mindegyik munkalapot bezárja.

A munkalapok alapvetően az alábbi két típusba tartoznak:


**Táblázatok:** A közönséges és programozott nyelvtan különböző paramétereit (ábécé, szabályok stb.) egy táblázat kitöltésével adhatjuk meg. Ennek a típusnak előnye, hogy használata egyszerű, a formális nyelvtanok ismeretén kívül semmilyen előismeretet nem igényel.

**Program:** Programozott nyelvtant, Turing-gépet ill. veremautomatát adhatunk meg egy erre szolgáló programnyelv segítségével. Előnye, hogy a nyelv kifejezőereje lehetővé teszi sokkal nagyobb és bonyolultabb nyelvtanok létrehozását, ill. ugyanaz a nyelvtan rövidebben fogalmazható meg programként, mint táblázattal. Könnyen modellezhetünk pl. mátrixnyelvtanokat vagy Lindenmayer rendszert is. Az automatákat (T-gép, VA) csak programként adhatjuk meg, táblázattal nem.

Az alábbi fejezetekben áttekintjük a különböző típusú munkalapokat.

#### 2.1.1 Táblázat - Nyelvtan

Ezen a munkalapon közönséges formális nyelvtanokat adhatunk meg táblázatos formában. Használat:

1. A bal oldali táblázatokban adjuk meg a nyelvtan szimbólumait és szabályait. A táblázat alján megjegyzést fűzhetünk a nyelvtanhoz, pl. a generált nyelv leírását, vagy utasításokat a használatra.
2. Futtassuk le a nyelvtant a  gombra kattintva.
3. A futtatás gyakorlatilag azonnal lezajlik, és az ablak jobb oldalán böngészhetjük ennek eredményét, vagyis a generált levezetéseket.

Start szimbólum	Nemterminálisok	Terminálisok
S	A B L R	a
Bal oldal	Jobb oldal	
S	A L a B	
A L	A R	
R B	L B	
a L	L a a	
R a	a a R	
A L	R	
R B	L	
A L	eps	
R B	eps	

Generált nyelv:  $a^*(2^n)$   
Az A L -> eps szabályt csak akkor szabad használni, ha már nincs B.  
Hasonlóan R B -> L csak akkor, ha már nem szerepel A.

1. ábra: Közönséges nyelvtan megadása táblázattal

A szimbólumok megadására vonatkozó szabályok:

- A szimbólum az angol ábécé betűiből és számjegyekből állhat. Kezdődhet számjeggyel, és állhat csak számjegyből is.
- Lista esetén (pl. terminális jelek felsorolása) a szimbólumokat szóközzel kell elválasztani.
- A görög  $\epsilon$  használható rövidítése: **eps**.

### 2.1.1.1 Lista nézet

A lista nézet célja egyetlen levezetés megjelenítése. Az elemeket automatikusan generálja a program bizonyos szabályok szerint (ld. Beállítások 2.2), de az elemekre kattintva módosíthatjuk is a levezetés menetét. A listának két oszlopa van:

**Mondatforma:** Az adott lépéshez tartozó mondatforma. Abban az esetben, ha a megelőző lépés során a szabály több pozíción is végrehajtható volt (pl. az  $AAB$  mondatformára az  $A \rightarrow a$  szabály két helyen is illeszthető), akkor a mondatformára kattintva megválaszthatjuk a kívánt pozíciót.

**Szabály:** Ez az oszlop tartalmazza a nyelvtani szabályt, amit

alkalmazunk az aktuális mondatformára. Amennyiben több szabály is alkalmazható, az elemre kattintva megválaszthatjuk a nekünk megfelelőt.



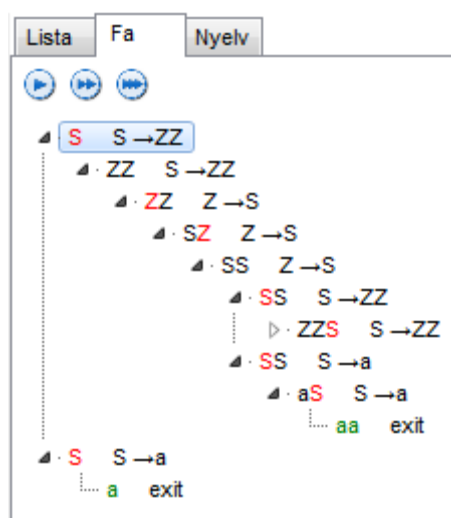
2. ábra: Lista nézet

A lista nézet tetején található egy egyszerű animációs rész, amivel lejátszhatjuk a levezetést lépésről lépésre. A lejátszást a szokásos gombokkal irányíthatjuk.

### 2.1.1.2 Fa nézet

A fa nézet célja az összes lehetséges levezetés megjelenítése kibontható fa struktúrában. A nyelvtan lefuttatása után a fa egyedül a kezdő konfigurációt tartalmazza, melyet ha a mellette lévő jellel „kibontunk”, megjelennek a belőle közvetlenül levezethető konfigurációk.

A panel tetején lévő gombokkal 4, 16 vagy 64 lépés mélységben, rekurzív módon bonthatjuk ki a kijelölt részét.



3. ábra: Fa nézet

### 2.1.1.3 Színkódok

A program piros színnel jelöli azokat a részzavakat, melyekre az aktuális szabály illesztése történt. Ha Fa nézetben egyszerre több illesztés is tartozik egy elemhez, ekkor ezek közül az első számít. Ha egy szó csupa terminális jelből áll, vagyis a generált nyelv részét képezi, akkor a színe zöld (ld. 3. ábra **aa** és **a** szavai). Ha egy szóra nem folytatható a levezetés (ezek „sikertelen” levezetés-ágaknak tekinthetők), akkor a színe piros (ld. **Hiba! A hivatkozási forrás em található.** utolsó eleme: **aaaaB**).

### 2.1.1.4 Nyelv nézet

A nyelv nézet célja a nyelvtan által generált nyelv néhány elemének megjelenítése. Mivel ez a számítás időigényes lehet (a hatékonyság nagyban függ a beállításoktól (2.2)), csak szükség esetén indul el: akkor, ha a felhasználó a Nyelv fülre kattint. Egy idő után megjelenik egy felugró ablak mely az eltelt időt jelzi, valamint a Cancel-gomb megnyomásával lehetőséget ad a számítás leállítására. A generált elemek ekkor sem vesznek el, csak nem keletkeznek továbbiak. A számítás befejeztével az eredményeket hosszúságuk alapján rendezi sorba, másodlagos rendezési szempont szerint pedig lexikografikusan.

### 2.1.2 Táblázat - Programozott nyelvtan

Ezen a munkalapon programozott nyelvtanokat modellezhetünk. Használata megegyezik a közönséges nyelvtanokéval, azzal a különbséggel, hogy most meg kell adnunk minden szabályhoz egy címkét is (ID oszlop), valamint a  $\sigma$ ,  $\varphi$  halmazokat. Ezen halmazok jelentése: ha a levezetés egy lépésében pl. az alábbi ábrán a 3. címkéjű szabályt hajtottuk végre, és a végrehajtás sikeres volt (vagyis a mondatformában szerepelt B szimbólum, amit lecserélhettünk Bb-re), akkor a következő lépésben a 4 vagy a 2 címkéjű szabállyal kell folytatni a levezetést. Ha a végrehajtás sikertelen volt (nem szerepelt B a mondatformában), akkor a  $\varphi$  oszlopból kell címkét választanunk. Ez az oszlop a példában most egyedül az **exit**-et tartalmazza, ami azt jelenti, hogy ekkor a levezetés véget ér.


Start szimbólum		Nemterminálisok		Terminálisok
S		A B		a b
ID	Bal oldal	Jobb oldal	$\sigma$	$\varphi$
1	S	A B	2 4	exit
2	A	A a	3	exit
3	B	B b	4 2	exit
4	A	eps	5	exit
5	B	eps	exit	exit

Generált nyelv:  $a^n b^n$ , ahol  $n \geq 0$

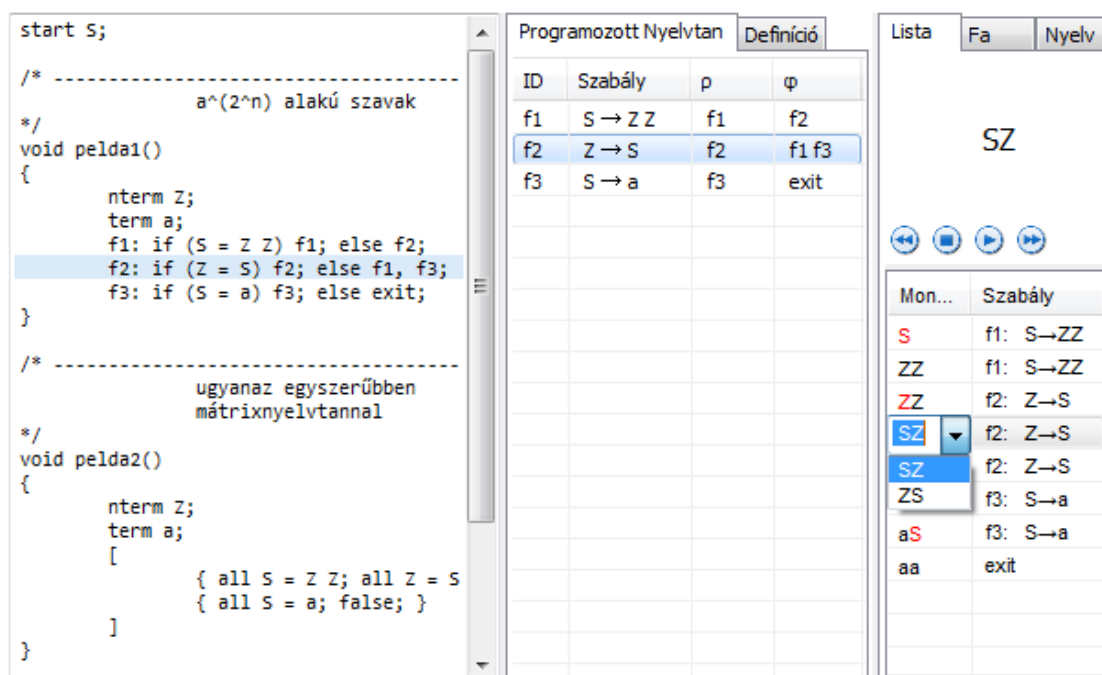
4. ábra: Programozott nyelvtan megadása táblázattal

### 2.1.3 Program – Programozott nyelvtan

Ezen a munkalapon programozott nyelvtanokat adhatunk meg egy erre szolgáló programnyelv segítségével. Használat:

1. A bal oldali kód-szerkesztő ablakrészben írjuk meg a nyelvtan megadására szolgáló programkódot. A nyelv részletes leírását ld. a 2.3 fejezetben. Üres munkalap létrehozásakor egy előre megírt vázat kapunk, amely a PNY-ok használatához van igazítva.
2. Futtassuk le a programot a  gombra kattintva. A futtatás két lépésben történik:
  - 2.1. A munkalap középső részén megjelennek a kész nyelvtan szabályai. Ha a programkód hibás, hibaüzenetet kapunk, és a nyelvtan nem jön létre. Szintén a középső részen találunk egy „Definíció” fület, amelyre kattintva megtekinthetjük a PNY matematikai definíciónak megfelelő leírását (ennek jelentősége leginkább majd a T-gép és a VA esetén lesz).
  - 2.2. Az előállított nyelvtan generálja a levezetések és a nyelvet, melyeket a jobb oldali ablakrészben böngészhetünk a táblázatos munkalapokéval megegyező módon.

Ha középső ablakrészben a táblázat egy sorára kattintunk (vagyis egy nyelvtani szabályra), az ablak bal oldalán a forráskódban kijelölésre kerül az a programsor, mely az adott szabály leírását tartalmazza. Ha a jobb oldalon a levezetés böngészőben egy konfigurációra kattintunk, az imént említett két ablakrészben szintén kijelölésre kerül a konfigurációhoz tartozó megfelelő sor.



The screenshot displays the macro editor interface with three main panels:

- Left Panel (Code Editor):** Contains two C-like functions. The first, `pelda1()`, uses conditional logic to derive strings based on grammar rules. The second, `pelda2()`, uses a matrix-based approach for the same purpose.
- Middle Panel (Grammar Rules Table):** A table with columns ID, Szabály, p, and φ. It lists three rules: f1 ( $S \rightarrow ZZ$ ), f2 ( $Z \rightarrow S$ ), and f3 ( $S \rightarrow a$ ).
- Right Panel (Derivations and List):** Includes a 'Lista' tab showing a list of strings (SZ, ZZ, ZZ, SZ, SZ, ZS, aS, aa) and a 'Szabály' tab showing the corresponding grammar rules for each string.

5. ábra: A makrószerkesztő munkalap.

#### 2.1.4 Program – Turing-gép, Veremautomata

Turing-gépek programozása és futtatása megegyezik a programozott nyelvtanokéval.

Különbség a levezetések megjelenítésében van:

- A középső ablakrészben a T-gép állapotai és állapotátmenetei jelennek meg. Minden T-gépnek van egy elutasító és egy elfogadó állapota (melyekre **exit**-tel ill. **accept**-tel hivatkozhatunk), ezeknek nincsenek átmenetei.
- A lista nézetben a mondatforma helyén a szalag állapota látható, rajta pirossal megjelölve a fej pozíciója. Ez nem módosítható, mivel T-gép esetén itt nincsenek

alternatív választási lehetőségek. A szabályra kattintva viszont választhatunk a konfigurációra alkalmazható állapotátmenetek közül.

- A fa nézetben is a szalag állapota látható.
- A nyelv nézetben a T-gép által felismert nyelv (tehát nem az eldöntött nyelv!) néhány eleme látható.
- A levezetés-böngészőben egy elemre kattintva a program kijelöl egy sort a középső ablakrészben, ami egy állapotátmenetet ír le: az aktuális konfigurációból a következő konfigurációba vezető állapotátmenet. Egy átmenetet egyszerre több utasítás is leírhat a forráskódban (olvasás, írás, léptetés), ezek közül mindig az utolsó számít, vagyis ezt jelöli ki a program. A Fa nézetben egy konfigurációnak több gyerek-konfigurációja is van, vagyis egyszerre több átmenetet is ki kellene jelölni, a program ezek közül csak az elsőt jelöli ki.

Különbségek **veremautomaták** esetén:

- A középső ablakrészben láthatóak a veremautomata állapotai és átmenetei. Ezúttal is hivatkozhatunk az **exit** és **accept** állapotokra, most azonban ezek virtuális állapotok, nem tartoznak hozzá az automata matematikai definíciójához, csak a használat megkönnyítését (és egységességét) szolgálják. Ezzel függ össze, hogy az **accept**-re mutató átmenetek verembe és veremki mezője üres, vagyis ezek nem valódi átmenetek.
- Lista és Fa nézetben a szalag helyett az automata bemenete (pirossal megjelölve az aktuálisan olvasott szimbólum) és verme látható, melyek nem módosíthatóak, csak az alkalmazható átmenetek.
- A nyelv nézetben az automata által felismert nyelv néhány eleme látható.

A T-gépek és veremautomaták programozásáról bővebben ld. a 2.3 fejezetben.

## 2.2 Beállítások

### 2.2.1 Általános

**Determinisztikus lépések elrejtése:** Ha ez az opció be van kapcsolva, akkor lista nézetben csak az első és az utolsó állapot jelenik meg, valamint azok, melyekben a levezetés elágazáshoz érkezik, vagyis több irányban is folytatható. Ez az opció hasznos lehet olyan esetekben, amikor áttekinthetetlenül sok lépésből csak azokat akarjuk meghagyni, melyek interakciót tesznek lehetővé.

**Animáció, egy lépés időtartama:** Megadhatjuk, hogy a lista nézetben lévő animáció egy lépése hány ms alatt játszódjon le.

**Mondatforma, verem, szalag maximális hossza:** Ha egy levezetésben a mondatforma, verem vagy szalag hossza eléri ezt a küszöböt, a levetés megáll és nem folytatható. T-gép esetén a szalag hossza természetesen (a végtelen) szalagon lévő használt (nem \_ jelet tartalmazó) rész hosszát jelenti.

**Generálás maximális ideje:** ms-ban megadhatjuk mennyi ideig tarthat a megoldások keresése.

**Találatok:** A nyelv nézetben a találatok maximális száma. Ezt elérve a program nem keres több megoldást.

**Lista nézet, lépésszám:** Egy futtatás alkalmával hány lépést tegyen meg a lista nézetben látható levezetés (ez nem feltétlenül egyezik meg a megjelenített elemek számával, annál több is lehet). Ha a levezetés még folytatható, a lista alján megjelenik egy Tovább gomb.

**Fa nézet, elemszám:** Ha a Fa nézet elemszáma elérte ezt a küszöböt, nem folytatódik a rekurzív kibontás (viszont manuálisan, egyesével továbbra is lehet folytatni).

### 2.2.2 Programozott nyelvtan beállításai

A különböző nézetekben (Lista, Fa, Nyelv) megadhatjuk, hogy a program milyen szabályok szerint generálja ill. szűrje a levezetéseket. A következő lehetőségek adottak:

**Pozíció választás nyelvtanok esetén:** Egy nyelvtani szabály alkalmazásánál dönteni kell arról, hogy a szabály bal oldalát - amennyiben többször is előfordul az aktuális mondatformában - melyik pozícióra illesszük. Ettől a választástól függően lehet a levezetés legbal, legjobb, véletlen valamint teljes (minden lehetőségre kiterjedő).

**Szabály választás:** Amennyiben valamely konfigurációnak több gyereke is van, a gép működése nem determinisztikus. Bizonyos esetekben teljesítmény okokból szükség lehet a determinisztikus működés kikényszerítésére. A pozíció választáshoz hasonlóan itt is dönthetünk arról, hogy az első, az utolsó, véletlenül választott szabály kerüljön alkalmazásra, vagy pedig mindegyik. Mivel a kezdő felhasználó számára ez az opció zavaró lehet, ill. ritkán van rá szükség, a felhasználói felületből nem érhető el, csak fordítási direktívával (ld. 2.3.2, **rule** direktíva).

**Zsákcák elrejtése:** Ha programozott nyelvtanokkal közönséges nyelvtanokat akarunk modellezni, gyakran szembesülünk azzal a problémával, hogy néhány lépés után a program terminál, mivel olyan szabályt próbál végrehajtani ami nem végrehajtható, miközben más szabályok még végrehajthatóak volnának (a definíciók alapján ilyenkor a közönséges nyelvtan „megkeresi” az alkalmazható szabályt, a PNY viszont erre nem képes). A levezethető szavak halmazán ez nem változtat, de némely levezetések használhatóságát erősen lerontja. Ezt elkerülhetjük azzal, ha elrejtjük a zsákcákat, vagyis azokat a konfigurációkat, melyek egy lépésben biztosan az **exit** állapotba vezetnek (pontosabban: csak akkor választ a program ilyen utat, ha nincs más lehetőség). Az elrejtés bekapcsolása ugyanakkor zavaró is lehet, mivel nem a definíciónak megfelelő működést eredményez.

### 2.2.3 Turing-gép, Veremautomata

**Bemenetek száma:** Turing-gép és veremautomata esetén a program különböző bemenetekre futtatja le az automatát, és vizsgálja hogy terminál-e. Ezek maximális számát adhatjuk itt meg.

**Konfiguráció/Bemenet:** Ha egy adott bemenetre az automata nem terminál, akkor egy bizonyos mennyiségű konfiguráció után abba kell hagynia a vizsgálatot, és a következő bemenetre kell térnie. Ennek a küszöbnek az értékét adhatjuk meg itt. Természetesen előfordulhat, hogy egy nagyobb küszöb beállítása esetén ugyanaz a bemenet már terminálni fog.

### 2.2.4 Könyvtárak

Itt megadhatjuk, mely könyvtárakban keresse a fordítóprogram az **#include** utasításban szereplő fájlokat (ha nem használunk **#include**-ot, ez a beállítás lényegtelen). Alapértelmezés szerint a lib alkönyvtár szerepel a listában.

### 2.2.5 Stílusok

Ezen a panelen beállíthatjuk a programban használt betűtípusokat. Mivel nem minden karakterkészlet tartalmazza a szükséges karaktereket ( $\rightarrow$ ,  $\epsilon$ ), nem megfelelő betűtípus beállítása esetén hibás lehet a megjelenítés. Külön adható meg három típus:

**Kódszerkesztő:** A bal oldalon elhelyezkedő ablakrész betűtípusa, amibe a programkódot írhatjuk. Ennek a betűtípusnak érdemes valamilyen fix szélességű (monospaced) típust választani, pl.: consolas, courier new.

**Animáció:** A Lista nézet tetején megjelenő animációs rész. Ezt a betűt érdemes nagy méretűnek választani.

**Ablak:** A program összes többi része (menük, listák stb.).

## 2.3 A Prolan nyelv

Szintaktikai szempontból a nyelv alapja a C programozási nyelv, amellyel sok hasonlóságot mutat. Ilyen hasonlóság pl. hogy a kis és nagybetűt megkülönbözteti, a deklarációk alakja, a blokkok, függvények használata stb.. Egyéb szempontból a Prolan sokkal egyszerűbb és könnyebben tanulható mint a C, mivel nem általános célú programnyelv. Nincsenek változói, pointerei. Az egyetlen érték típusa a logikai típus, a szokásos alaptípusok (számok, string, tömb stb.) mind hiányoznak, és nem hozhatóak létre új típusok, vagyis nincs típuskonstrukció.

Az imperatív nyelvek lényege, hogy változókon végzett műveleteken keresztül módosítjuk a számítógép belső állapotát. Programozott nyelvtanok esetén ez úgy értelmezhető, hogy a programnak egyetlen változója van, az aktuális mondatforma, a műveletek pedig amikkel módosítjuk, a nyelvtani szabályok. Ilyen értelemben tehát a Prolan egy imperatív nyelv.

### 2.3.1 Hello World!

```
start S;  
term Hello, World;  
void main()  
{  
    S = Hello World;  
}
```

A fenti program deklarál egy startszimbólumot: S, és két terminális jelet: Hello és World. Valamint definiál egy main függvényt, melynek egyetlen sora az S jelet kicseréli a Hello World szóra. Ez a program tehát egy olyan generatív nyelvtant ír le, mely egyetlen szót generál: Hello World. (felkiáltójel nem szerepelhet nyelvtani jelben, ezért erről most le kellett mondanunk)

Általában a program törzse tartalmazhat:

- Makródefiníciókat.

- Globálisan látható szimbólumok deklarációit.
- Tartalmaznia kell pontosan egy **void main()** makródefiníciót.
- Fordítási direktívákat, kommenteket.

Nézzünk egy tanulságosabb példakódot, ami a 3 faktoriálisát „számolja ki”.

#### **Példa**

```
#pragma leftmost
#include "matek.pla"
start S;
void fact(nterm X, nterm N)
{
    X = N;
    all X = eps;
    N = X;
    while (N = eps) Szor(X, N);
}

void main()
{
    nterm X, N;
    S = X X N N N;
    fact(X, N);
}
```

A fenti program bemenetét az  $S = X X N N N$ ; utasítással adjuk meg. Ebben 3db  $N$  szerepel, amit a **fact** eljárás úgy fog értelmezni, hogy az  $X$  szimbólum előfordulási darabszámát a mondatformában  $3!$ -ra kell beállítania, miközben  $N$  darabszámát nullára csökkenti (vagyis a várt eredmény:  $X X X X X X$ ).  $N$  kezdeti darabszáma zérus is lehet, hiszen  $0! = 1$  értelmezhető (ez indokolja az eljárás feleslegesnek tűnő első három sorát).

Néhány szó az utasításokról: Az  $X = N$  logikai kifejezés egy  $X \rightarrow N$  nyelvtani szabály végrehajtásának felel meg. A kifejezés értéke **true**, ha szerepelt  $X$  az aktuális mondatformában, különben **false** (ebben az esetben most a program terminálna). Az **all** utasítás addig értékeli ki a mögötte álló logikai kifejezést, amíg annak értéke igaz, vagyis eredményét tekintve most kitörli a mondatformából az összes  $X$ -et. Az **eps** jelentése a görög  $\varepsilon$  (üres szó). A **while** utasítás a jól megszokott elől tesztelési ciklus. A **Szor** eljárás definíciója a `lib/math.pla` fájlban található, amit a kód elején az **#include** direktívával beszerkesztettünk. A két paraméterként átadott szimbólum darabszámát összeszorozza, ennyi darabot tesz vissza az első paraméterben szereplő szimbólumból, valamint változatlanul hagyja a második paraméter darabszámát. A paraméterek típusa **nterm**, ami a nemterminális jelek típusa.

Megfigyelhetjük, hogy a program futása szempontjából csak a szimbólumok előfordulási számának van jelentősége, az elhelyezkedésüknek nincs (vagyis gyakorlatilag olyan, mintha egy zsak struktúrán végeznénk műveleteket). A PNY programok jelentős része ebbe a típusba tartozik. Ilyen esetekben teljesítmény-szempontból érdemes bekapcsolni a **#pragma leftmost** direktívát, hogy legbal levezetéseket gyártson a program.



### 2.3.2 Fordítási direktívák, kommentek

A program egy C előfordítót használ, ezért a kommentezés szabályai és a használható fordítási direktívák megegyeznek a C-ben megszokottakkal (bővebben ld. (2)), kiegészítve néhány új direktívával:

- Egysoros komment: `//` jeltől a sor végéig.
- Többsoros komment: `/* szöveg.... */`
- **#include** <filename>: A filename fájl tartalmát beszerkeszti az utasítás helyére.
- **#define** <NAME>: Definiálja a NAME tokenet. Ld. még **#ifdef**, **#endif**. (2)
- **#pragma**: Minden Prolan-specifikus (vagyis nem C) fordítási direktíva **#pragma**-val kezdődik  
(A pragma általában implementációfüggő fordítási direktívát jelöl. Ezeket az utasításokat a C előfordító változatlanul hagyja, és átadja őket a fordítóprogramnak.)
- **#pragma pgrammar|grammar|lindenmayer|pushdown|epushdown|turing**  
Közi a fordítóprogrammal, hogy a forráskód milyen típusú nyelvtant/automatát ír le:
  - **pgrammar**: Programozott nyelvtan. Ez az alapértelmezett, elhagyható.
  - **grammar**: Közöséges nyelvtan. Hatása megegyezik **pgrammar**-ral, azzal a különbséggel, hogy bekapcsolja a „Zsákutcák elkerülése” funkciót. (ld. 2.2.2)
  - **lindenmayer**: Lindenmayer-rendszer. Hatása alapvetően megegyezik **pgrammar**-ral, tőle annyiban különbözik, hogy bekapcsolja a determinisztikus lépések elrejtését, a színek elrejtését Lista nézetben, valamint legbal levezetést állít be az összes nézet számára.
  - **pushdown**: Végállapottal felismerő veremautomata.
  - **epushdown**: Üres veremmel felismerő veremautomata.
  - **turing**: Turing-gép
- **#pragma leftmost**: Legbal levezetést állít be mindegyik nézet számára (Lista, Fa, Nyelv). Hatása megegyezik a Beállítások ablakban a „Nyelvtani szabály bal oldalát melyik pozícióra illessze?” opcióval, viszont annál magasabb prioritású. Vagyis ha a kódban szerepel ez a direktíva, a Beállításokban szereplő értéket ignorálja a program. Ugyanez vonatkozik az összes alábbi direktívára.
- **#pragma rightmost**: Legjobb levezetést állít be mindegyik nézet számára.
- **#pragma hide(determ)**: bekapcsolja a „Determinisztikus lépések elrejtése” funkciót. Ez az utasítás megegyezik az azonos nevű funkcióval a Beállítások panelen.  
**determ** helyett állhat:
  - **deadend**: Bekapcsolja a „Zsákutcák elkerülése” funkciót.
  - **all**: Minden lépést elrejt.
  - **color**: Elrejt a színeket a Lista és Fa nézetben.  
A **determ**, **deadend** és a **color** hatása a kód egészére kihat, a **#pragma hide(all)** viszont kombinálva a **#pragma show(all)** utasítással kódrészletekre is használható (így csak az adott utasítások által generált lépéseket rejti el).
- **#pragma show(determ)**: A fenti utasítás ellentettje.
- **#pragma printmode**: Aktiválja a **print** utasításokat (ld.: 2.3.7).

- **#pragma position(list, left)**: Legbal levezetést állít be a Lista nézet számára. Megegyezik a Beállítások panel „Nyelvtani szabály bal oldalát melyik pozícióra illessze?” funkcióval.
  - **list** helyett állhat **tree** (vagyis Fa) vagy **lang** (Nyelv).
  - **left** helyett állhat **right** (jobb) és **all** (mind, ami a lista nézet esetén a véletlennek felel meg).
- **#pragma rule(list, left)**: Megadja, hogy a Lista nézetben, ha több szabály közül is lehet választani egy lépésben, akkor mindig az első szabályt válassza. (ld. 2.2.2, Szabály választás). **list** és **left** helyett ugyanazok állhatnak mint a fönti direktívánál. Erre a direktívára ritkán van szükség. Általában akkor, ha ki akarjuk kényszeríteni az automata determinisztikus működését.

### Példák

```
1. all X = x; // minden X-et x-re cserél
2. /* all X = x;
   if (A = A) B = b; // egysoros komment is lehet benne
   */
3. #include "math.pla"
4. #pragma turing
5. #pragma position(tree, all)
6. #pragma rule(lang, right)
```

1. Egysoros komment. A // jel utáni rész nem kerül lefordításra.
2. Többsörös komment. Egyik utasítás sem kerül lefordításra.
3. A fordító beszerkeszti a direktíva helyére a math.pla fájl tartalmát.
4. Utasítja a fordítóprogramot, hogy Turing-gépként értelmezze a forráskódot.
5. A fa nézetben mindegyik pozícióra illeszt szabályt.
6. A nyelv nézetben ha egy lépésnél több szabály is alkalmazható, akkor mindig az utolsót választja.

### 2.3.3 Típusok

Függvények visszatérési értéke lehet:

- **bool**: Logikai
- **void**: Érték nélküli.

Szimbólumok típusai:

- **letter**: Nyelvtani jel. Altípusai:
  - **term**: Terminális jel.
  - **nterm**: Nemterminális jel. Altípusa:
    - **start**: Startszimbólum.

### 2.3.4 Deklarációk

Nyelvtani jeleket és függvényeket deklarálhatunk, változókat nem, mivel a programnak nincsenek változói. A deklarált nevek angol betűket és számjegyeket tartalmazhatnak (kezdődhet számmal, és állhat csak számból is).

## Szintaxis

```
<deklaráció> ::= ("start" | "term" | "nterm") <név> ("," <név>)* ";"  
<név> ::= (<szám> | <betű>)*
```

## Példák

```
1. start S;  
2. nterm 1, A, alma1, 6alma;  
3. term a, 2, alma2;
```

1. Deklarál egy startszimbólumot: S.
7. Deklarál négy nemterminális jelet: 1, A, alma1, 6alma.
8. Deklarál három terminális jelet: a, 2, alma2.

## Megjegyzések

- A startszimbólum szokásos jele S.
- Terminális jeleket szokásosan a latin ábécé első kisbetűivel jelöljük: a, b, c, ...
- Nemterminális jeleket szokásosan a latin ábécé első nagybetűivel jelöljük: A, B, C, ...
- A számokat érdemes az utasítások felcímkezéséhez fenntartani, pl.: 1: S = a b c;
- Címkéket nem lehet deklarálni, felismerésük automatikus. A címkék használatát ld. az utasítások fejezetben (0).

### 2.3.4.1 Hatókör, láthatóság

Egy szimbólum hatóköre megegyezik a blokk hátralevő részével, amiben deklaráltuk.

Lehetséges globális szimbólumokat deklarálni, ezeket a program elején a blokkokon kívül kell elhelyezni.

Ha egy szimbólum neve megegyezik egy korábban deklarált szimbólumével, akkor elrejtí az. A generált nyelvtanban ezeket az azonos nevű szimbólumokat indexekkel különbözteti meg a program. Példa az elrejtésre és indexelésre:

Programozott Nyelvtan		Definíció	
ID	Szabály	$\sigma$	$\varphi$
0	$A \rightarrow a$	1	exit
1	$A1 \rightarrow b$	2	exit
2	$A2 \rightarrow c$	exit	exit

6. ábra: Példa elrejtésre és indexelésre.

### 2.3.5 Beépített konstansok

<i>Kulcsszó</i>	<i>Típus</i>	<i>Jelentés</i>
<b>true</b>	<b>bool</b>	Igaz logikai érték.
<b>false</b>	<b>bool</b>	Hamis logikai érték.
<b>eps</b>	<b>term</b>	Epsilon görög betű. Az üres szó jele.
<b>_</b>	<b>term</b>	"blank", az üres karakter T-gép esetén
<b>exit</b>	<b>label</b>	Elutasító állapot címkéje, terminálja a programot.
<b>accept</b>	<b>label</b>	Elfogadó állapot címkéje, terminálja a programot.

### 2.3.6 Logikai kifejezések

Mivel a Prolan-ban az egyetlen érték típus a **bool**, csak logikai operátorok vannak, melyek a következők:

<i>Jel</i>	<i>Jelentés</i>	<i>Precedencia</i>	<i>Asszociativitás</i>
<b>=</b>	Helyettesítés	5	nem asszociatív, bináris
<b>&amp;&amp;</b>	Lusta és	4	bal, bináris
<b>!</b>	Negáció	3	jobb, unáris
<b>  </b>	Lusta vagy	2	bal, bináris
<b> </b>	Elágazó vagy	1	bal, bináris

PNY esetén a „helyettesítés” az aktuális mondatformán hajt végre egy adott nyelvtani szabálynak megfelelő helyettesítést. A szabály bal oldala a bal oldali operandus, jobb oldala a jobb oldali. Értéke **true** ha a helyettesítés végrehajtható, **false** ha nem. VA esetén egy veremműveletet ír le: a bal oldali operandust kiveszi a veremből (amennyiben az olvasható a verem tetején), és a jobb oldali operandust írja a helyére. A példákban a PNY esetét fogjuk alapul venni, a VA-król részletesebben lásd a 2.4 fejezetben.

Az „elágazó vagy” kiértékelése nemdeterminisztikus: véletlenszerűen végrehajtja valamelyik operandusát, értéke megegyezik a kiválasztott operandus logikai értékével. A nem kiválasztott operandus nem hajtódik végre, nem befolyásolja az eredményt.

#### Szintaxis

```

<helyettesítés> ::= <mondatforma> "=" <mondatforma>
<és> ::= <logk> "&&" <logk>
<vagy> ::= <logk> "||" <logk>
<negáció> ::= "!" <logk>
<elágazó vagy> ::= <logk> "|" <logk>
<logk> ::= <helyettesítés> | <és> | <vagy> | <negáció>
          | <elágazó vagy> | <loghívás> | "true" | "false"

```

### Példák

1.  $A = B \text{ a } b$
2.  $(A = a) \ \&\& \ (B = b)$
3.  $(A = a) \ || \ (B = b)$
4.  $! \ A = a$
5.  $(A = a) \ | \ (B = b) \ | \ (C = c)$

1. A-t cseréli B-re. Értéke **true**, ha volt A a mondatformában, egyébként **false**.
2. A-t a-ra cseréli, ha sikerült akkor B-t b-re cseréli. Ha valamelyik csere nem sikerült, a kifejezés értéke **false**, különben **true**.
3. A-t a-ra cseréli, ha sikerült akkor a kifejezés értéke **true**, különben B-t b-re cseréli. Ha ez sikerült a kifejezés értéke **true**, egyébként **false**.
4. A-t a-ra cseréli, ha sikerült akkor a kifejezés értéke **false**, különben **true**.
5. Az alábbi három lehetőség közül véletlenszerűen végrehajtja valamelyiket:
  - A-t a-ra cseréli, ha sikerült akkor a kifejezés értéke **true**, különben **false**.
  - B-t b-re cseréli, ha sikerült akkor a kifejezés értéke **true**, különben **false**.
  - C-t c-re cseréli, ha sikerült akkor a kifejezés értéke **true**, különben **false**.

#### 2.3.6.1 Logikai kifejezések mint utasítások

Minden logikai kifejezésből alkotható utasítás egy pontosvesszővel:

```
<utasmag> ::= <logk> ";" | ...
```

Jelentése: a kifejezés végrehajtódik. Ha értéke **true**, a program futása folytatódik a következő utasításon. Ha értéke **false**, a program terminál.

### Példa

```
(A = a) && (B = b); C = c;
```

A-t a-ra cseréli, ha sikerült akkor B-t b-re cseréli. Ha valamelyik csere nem sikerült, a program terminál, különben folytatódik a következő utasításon, vagyis C-t c-re cseréli.

#### 2.3.7 Utasítások

Az alábbiakban felsoroljuk azokat az utasításokat, melyek mindhárom alkalmazási területen használhatóak. A T-gép és VA specifikus utasításokat ld. a 2.4 fejezetben.

- **if** (<feltétel>) <utasítás>  
Ha a feltétel igaz, végrehajtja az utasítást.
- **if** (<feltétel>) <utasítás1> **else** <utasítás2>  
Ha a feltétel igaz, végrehajtja utasítás1-et, különben végrehajtja utasítás2-t.
- **while** (<feltétel>) <utasítás>  
Amíg a feltétel igaz, végrehajtja az utasítást.
- **all** <feltétel> ;  
Amíg a feltétel igaz, végrehajtja. (Hatása megegyezik **while** (<feltétel>); utasításával)
- **try** <feltétel>;  
Kiértékeli a feltételt. Az utasítás feladata, hogy megakadályozza hamis érték esetén a program terminálását. Hatása megegyezik az **if** (<feltétel>); utasítással (a pontosvessző előtt egy utasítás hiányzik, ami így tkp. egy skipnek felel meg).

- **goto** <címke1>, <címke2>, ... , <címken>;  
Véletlenszerűen ugrik a felsorolt címkék valamelyikére.
- <címke1>, <címke2>, ... , <címken>;  
Hatása megegyezik az előbbivel, vagyis a **goto** kulcsszó elhagyható.
- **print**  
Ha a kódban nem szerepel a **#pragma printmode** direktíva, a fordító ignorál minden **print** utasítást. Ha szerepel, akkor a Lista nézetben csak azon utasításokhoz tartozó lépések jelennek meg, melyeket **print** előz meg. PNY esetén minden **print**-hez egy-egy új virtuális nyelvtani szabály jön létre, T-gép és VA esetén pedig új állapotátmenetek, melyek egyedül a kiíratást végzik, de nem változtatnak a szalag ill. a verem tartalmán. Ezek a **print**-et tartalmazó automaták nem érvényesek matematikai értelemben, de működésük ekvivalens a print nélküli változattal.

### Szintaxis

```

<if> ::= "if" "(" <logk> ")" <utasítás> ("else" <utasítás>)?
<while> ::= "while" "(" <logk> ")" <utasítás>
<all> ::= "all" <logk> ";";
<try> ::= "try" <logk> ";";
<goto> ::= ("goto" <címkék>) | <címkék> ";";
<címke> ::= (<számjegy> | <betű>)+
<címkék> ::= <címke> ("," <címke>)*
<utasmag> ::= <if> | <while> | <all> | <goto> | <hívás> | <logk>";"
<utasítás> ::= (<címke> ":")? <utasmag>

```

### Példák

```

1. if (A = a) B = b;
2. while (A = a) B = b;
3. all A = a;
4. try A = a;
5. goto lab1, lab2;
6. lab1, lab2;
7. if (A = a) 1, 2; else 3, exit;
8. lab1: A = a;
   2: B = b;

```

1. A-t a-ra cseréli. Ha sikerült akkor B-t b-re cseréli.
2. Amíg az  $A \rightarrow a$  csere végrehajtható, B-t b-re cseréli.
3. Végrehajtja a feltételt amíg az igaz, vagyis minden A-t a-ra cserél.
4. Megpróbálja végrehajtani az  $A \rightarrow a$  cserét. Ha nem sikerült, akkor sem terminál.
5. Véletlenszerűen a lab1 vagy a lab2 címkeére ugrik.
6. Ugyanaz mint a fönti, vagyis a **goto** elhagyható.
7. A-t a-ra cseréli, ha sikerült, az 1 vagy a 2 címkeére ugrik. Ha nem, akkor terminál, vagy a 3 címkeére ugrik.
8. Címkével ellátott utasítások.

### 2.3.8 Blokkok

A nyelvben háromféle blokkutasítás van:

**„Szekvenciális”:** A blokk utasításai egymás után kerülnek végrehajtásra. (ez megfelel az imperatív nyelvekben megszokott blokkutasításoknak). Jele: { }

**„Elágazó”:** A blokk utasításai közül véletlenszerűen kiválaszt egyet, azt végrehajtja, majd kilép a blokkból. Jele: < >

**„Mátrix”:** A blokk utasításai közül újra és újra választ egyet véletlenszerűen, amíg a program nem terminál. Egyfajta végtelen ciklusként viselkedik, amiből csak terminálással lehet kilépni. Ez az utasítás hasznos közönséges nyelvtanok és mátrixnyelvtanok modellezésére (ld. 2.4). Jele: [ ]

**Megjegyzés:** mindhárom szerepelhet makró blokkjaként is, vagyis pl. `void main() [ S = a; ]` egy érvényes makródefiníció (ld. 2.4 fejezet).

### Szintaxis

```
<szekvblokk> ::= "{" <utasítás>* "}"  
<forkblokk> ::= "<" <utasítás>* ">"  
<mátrix> ::= "[" <utasítás>* "]"
```

### Példák

```
1. { A = a; B = b; }  
2. < A = a; B = b; >  
3. [ A = a; B = b; ]  
4. [ A = a; { B = b; C = c; } ]
```

1. A-t a-ra cseréli (ha nem sikerült, akkor terminál), majd B-t b-re (ha nem sikerült, akkor terminál).
2. Az alábbi két lehetőség közül választ egyet és végrehajtja.
  - A-t a-ra cseréli. Ha nem sikerült, akkor terminál.
  - B-t b-re cseréli. Ha nem sikerült, akkor terminál.
3. Az alábbi két lehetőség közül választ egyet és végrehajtja. Ha a végrehajtás sikerült újra válasz, ha nem akkor terminál.
  - A-t a-ra cseréli.
  - B-t b-re cseréli.
4. Az alábbi két lehetőség közül választ egyet és végrehajtja. Ha a végrehajtás sikerült újra válasz, ha nem akkor terminál.
  - A-t a-ra cseréli.
  - B-t b-re cseréli, majd C-t c-re cseréli

### 2.3.9 Makrók

A Prolan makrók néhány dologban különböznek a C előfordító makróitól, de a C függvényeitől is. Az alábbiakban összefoglaljuk hasonlóságokat és különbségeket:

Makrószerű tulajdonságok:

- A hívás feloldása fordítási időben történik. Ennek alapvető oka az, hogy nincs műveleti memória amiben futás közben lokális változókat tárolhatnánk, és elkülöníthetnénk a hívó programrésztől. Ugyanezen okból nem tárolhatnánk a hívási vermet sem.

- Rekurzív hívás engedélyezett, de ciklikus rekurzió nem. Ez a fordítási időben való feloldás következménye, ciklikus rekurzió végtelen ciklust okozna a fordítás során.
- A deklaráció és a definíció nincs különválasztva, nincs elődeklaráció. Ez a szabály biztosítja a ciklikus rekurzió tiltását.

Függvényszerű tulajdonságok:

- A szintaxis megegyezik a C függvényekével, szemben a nehezebben használható C makrók szintaxisával.
- A paraméterek lokális „változóként” viselkednek, nem termhelyettesítés történik.
- A makró **bool** vagy **void** típusú lehet, vagyis lehet visszatérési értéke, amiket a szokásos **return** utasítással adhatunk meg.

### Szintaxis

```
<függvdef> ::= ("void" | "bool") | <név> <paramdeks> <blokk>
<paramdeks> ::= "(" (<paramdek> ("," <paramdek>)*)? ")"
<paramdek> ::= ("term" | "nterm") <név>
<hívás> ::= <név> "(" <név> ("," <név>)* ")" ";"
```

### Példák

```
void main() { S = a; }
bool f(nterm X)
{
    X = a;
    if (Y = b) return Z = b;
    return false;
}
```

## 2.4 Speciális nyelvtanok, automaták programozása

A Prolan segítségével közvetlenül írhatunk le programozott nyelvtant, nemdeterminisztikus Turing-gépet vagy veremautomatát. Valamint ezek felhasználásával különféle egyéb nyelvtan és automata típus szimulálható (ezek sora remélhetőleg bővülni fog, a felhasználó is kísérletezhet új alkalmazások keresésével). Az alábbi fejezetekben bemutatom azokat az utasításokat és programozási technikákat, melyek a különböző speciális felhasználási területekhez szükségesek.

### 2.4.1 Közöséges nyelvtanok

A **#pragma grammar** direktívával utasíthatjuk a fordítóprogramot, hogy a kódot közöséges nyelvtanként értelmezze. Ez mindössze annyi különbséget jelent az alapbeállításhoz képest (**#pragma pgrammar**), hogy bekapcsolja a „zsákutcák elrejtése” opciót. Ennek értelmét az alábbi példán láthatjuk. Tekintsük a helyes zárójelezések nyelvét generáló közöséges nyelvtant, és szimuláljuk PNY-al:

címke	szabály	$\sigma$	$\varphi$
$f_1$	$S \rightarrow SS$	$f_1, f_2, f_3$	<i>exit</i>
$f_2$	$S \rightarrow (S)$	$f_1, f_2, f_3$	<i>exit</i>
$f_3$	$S \rightarrow \varepsilon$	$f_1, f_2, f_3$	<i>exit</i>

Látható, hogy a  $\sigma$  oszlop tartalma minden sorban megegyezik: tartalmazza az összes nyelvtani szabály címkéjét. Ennek jelentése az, hogy egy adott szabály végrehajtása után mindenféle



megkötés nélkül bármelyik másik szabályt alkalmazhatjuk. A  $\varnothing$  oszlopban mindenütt *exit* szerepel, vagyis ha az aktuális szabályt nem tudtuk alkalmazni, akkor minden esetben terminál a levezetés.

A fenti alakú PNY-ok tehát minden lépésben minden lehetséges szabályt kipróbálnak, és a sikertelen próbálkozások az *exit* címkén terminálnak. Ennek eredményeként egy véletlenszerűen választott PNY levezetés nagy valószínűséggel az *exit* címkén terminál akkor is, amikor pedig a levezetés még folytatható volna, mert választható alkalmas szabály. Ez a viselkedés természetesen megfelel a PNY definíciójának, de abban az esetben ha a cél közönséges nyelvtan modellezése, jó lenne elkerülni. Ebben segít a „zsákutcák elkerülése” beállítás, amely egy lépést előre tekintve elkerüli az *exit* címke választását. Pontosabban: nem választ olyan címkét, amely után mindenképpen az *exit*-et kell választani, csak akkor, ha nincs más választás.

A blokkutasítások között már találkoztunk a „mátrix” blokkal (2.3.8). Látható, hogy ennek viselkedése pont megfelel annak, amire közönséges nyelvtanok esetén szükségünk van: szabályok egy halmazából ismételten addig választ, amíg a program nem terminál. Példaként tekintsük a fenti helyes zárójelezések nyelvét generáló nyelvtant leíró kódot.

#### **Példa**

```
#pragma grammar
start S;
term a, b; // a = "(", b = ")"
void main()
[
    S = S S;
    S = a S b;
    S = eps;
]
```

#### **2.4.2 Mátrixnyelvtanok**

A mátrixnyelvtanok és Lindenmayer-rendszerek matematikai definícióval itt nem foglalkozunk (részletesen ld. (1)), csak nagyon röviden és vázlatosan illusztrálom a fogalmak jelentését a példafájlok használhatósága érdekében.

Mátrixnyelvtan alatt olyan nyelvtant értünk, melyben a nyelvtani szabályok szekvenciákba vannak rendezve. A szekvenciákon belül a szabályok végrehajtási sorrendje rögzített, viszont a szekvenciák közötti sorrend tetszőleges. Valamint a szekvenciák tranzakcióként viselkednek, vagyis vagy végrehajtjuk az elejétől a végéig, vagy el sem kezdjük. Ha már ismerjük a Prolan néhány utasítását, akkor az alábbi példakódból azonnal világossá válik a mátrixnyelvtanok működése.

**Példa:** Tekintsük a  $[S \rightarrow ABC], [A \rightarrow Aa, B \rightarrow Bb, C \rightarrow Cc], [A \rightarrow a, B \rightarrow b, C \rightarrow c]$  mátrixnyelvtan kódját.

```
start S;
nterm A, B, C;
term a, b, c;
void main()
[
    S = A B C;
    { A = a A; B = b B; C = c C; }
    { A = a; B = b; C = c; }
]
```

A mátrixnyelvtanok fordítása mindenben megegyezik a programozott nyelvtanokéval. Arra kell csak ügyelnünk, hogy ha „tisztán” mátrixnyelvtant akarunk létrehozni, akkor csak bizonyos utasításokat használhatunk: értékadás, mátrix blokk, és a **try** utasítás (pontozott szabályokhoz).

Pontozott szabály alatt olyan szabályt értünk, melyet ha alkalmazni akarunk az aktuális mondatformára, és ez sikertelen, a levezetés nem ér véget, hanem folytatódik a szekvencia következő szabályán. Pontozott szabályt a **try** utasítás segítségével hozhatunk létre, pl. az  $A \rightarrow a$  szabálynak a **try**  $A = a$ ; utasítás felel meg.

### 2.4.3 Lindenmayer-rendszerek

Lindenmayer-rendszer alatt leegyszerűsítve és vázlatosan a következőt értjük:

Tekintsünk egy  $A$  ábécét, valamint egy  $\sigma: A \rightarrow 2^{A^*}$  függvényt ( $\sigma$  tehát betűkhöz szavak egy halmazát rendeli). A levezetés egy lépése abból áll, hogy a mondatforma minden egyes  $a \in A$  betűjét átírjuk egy  $u \in \sigma(a)$  szóra.

**Példa:**  $MISS_3$  L-rendszer

Ábécé:  $\{x\}$ ; átíró függvény:  $\sigma(x) = \{\varepsilon, xx, xxxxx\}$

Egy lehetséges levezetés:  $x \rightarrow xx \rightarrow xxxxxxxx \rightarrow \varepsilon$

(a második lépésben az egyik  $x$ -et  $xx$ -re, a másikat  $xxxxx$ -re cseréltük. Az utolsó lépésben minden egyes  $x$ -et  $\varepsilon$ -ra.)

Generált nyelv:  $L(MISS_3) = \{x^n | n \in \mathbb{N}_0 \setminus \{3\}\}$

Az iménti nyelvet a következő Prolan kód generálja:

```
#pragma lindenmayer
start X;
nterm Y;
term x;
void main()
[
    {
        all X = eps | X = Y Y | X = Y Y Y Y Y;
        all Y = X;
    }
    all X = x;
]
```

A **#pragma lindenmayer** direktívával utasítottuk a fordítóprogramot, hogy a kódot Lindenmayer-rendszerként értelmezze. Ez annyiban különbözik az alapbeállítástól, hogy bekapcsolja a determinisztikus lépések elrejtését, a színek elrejtését Lista nézetben, valamint legbal levezetést állít be az összes nézet számára.

A fenti példában az „elágazó vagy”, vagyis a „|” operátort használtuk az átírási szabály megvalósítására. Emlékeztetőül: az  $A \mid B$  logikai kifejezés véletlenszerűen vagy A-t, vagy B-t értékel ki, és a kifejezés értéke megegyezik ezzel az értékkel. Az **all** utasítással együtt alkalmazva eredményül azt kapjuk, hogy az utasítás a mondatformában szereplő összes X-re alkalmazza a felsorolt szabályok valamelyikét, míg végül nem marad X a mondatformában.

Fent azért nem alkalmazhattunk  $X = X X$  és  $X = X X X X X$  szabályokat, mert ebben az esetben a program nem tudná megkülönböztetni hogy melyek az eredetileg szereplő X-ek, és melyek az újak. Így pl. a fenti példában végtelen ciklust kapnánk, de más esetekben is rossz működéshez vezetne. Ezért kellett egy új Y szimbólumot bevezetnünk, mely X helyettesének tekinthető. Miután minden X-et lecseréltünk, a helyetteseket vissza kell cserélnünk X-ekre, erre szolgál az **all Y = X;** utasítás.

A main() makró blokkja egy mátrix blokk, ami végtelen ciklusban véletlenszerűen választ a fent részletezett lépés, ill. az **all X = x;** utasítás közül. Ez utóbbi minden nemterminálist terminálisra cserél, vagyis előállítja a *MISS<sub>3</sub>* nyelv egy elemét, és terminálja a programot.

#### 2.4.4 Automaták

A Prolan közvetlenül kétféle automatát támogat:

- $\epsilon$ -átmenetes nemdeterminisztikus Turing-gép. Ez az automata alesetként tartalmazza a determinisztikus ill.  $\epsilon$ -mentes változatokat is, valamint könnyen modellezhető segítségével véges automata.
- (nemdeterminisztikus) Veremautomata. Változatai: üres veremmel felismerő, végállapottal felismerő.

A nyelv tervezésekor elsődleges cél volt, hogy az utasításkészlet minél kisebb és egyszerűbb legyen, hogy ugyanazokkal az utasításokkal lehessen a különböző automata variációkat leírni. Az automaták működtetése/futtatása szempontjából szintén cél volt a takarékoság és egységesség. Ennek következményeként a Prolan által generált automaták nem mindig felelnek meg pontosan a matematikai definícióknak, viszont ezeket is megtekinthetjük a munkalapok közepén elhelyezkedő „Definíció” fülre kattintva. Az eltérések:

- Minden ábécé tartalmazza az **nterm** típusú \_ (blank) karaktert. Veremautomatánál és véges automatánál ennek a karakternek az olvasása a bemenet végét jelöli.
- Minden automatának van előre definiált **exit** és **accept** állapota. A különböző automaták definíció szerinti megállási feltételeit (van-e végállapot, végig kell-e olvasni a bemenetet stb.) a blank karakter olvasásával és az **exit/accept** használatával lehet szimulálni.

Az automaták megadásánál használhatóak mindazok az utasítások, vezérlési szerkezetek, makróhívások mint a nyelvtanoknál. Az utasítások ügyes megválasztásával a kód sokkal rövidebb és áttekinthetőbb lehet, mint a kész automatát leíró táblázat. Ugyanakkor az

automaták esetén nagyon könnyű érvénytelen, vagy hibásan működő kódot megadni, ezért érdemes csak bevált módszereket alkalmazni. A példafájlokban mintát láthatunk arra, milyen programozói stílussal lehet egyszerűen és megbízhatóan T-gépet és veremautomatát megadni. Erre egyik lehetőség a **state** utasítás, aminek segítségével az automaták állapotait és átmeneteit gyakorlatilag explicit módon írhatjuk le. Az automaták közös utasításai:

- **input(<szó>)**  
VA esetén a <szó> adja a bemenetet, T-gép esetén a szalagra másolja a <szó>-t, és az első karakterére állítja a fejet. A <szó> alakja: **term** típusú jelek szóközzel elválasztva.
- **read(<term>)**  
VA esetén a bemenetről próbál <term>-et olvasni, az olvasás sikere az utasítás visszatérési értéke. T-gép esetén a szalag fej alatti részét próbálja olvasni.
- **(<term>)**  
Megegyezik a fentivel (vagyis a **read** elhagyható).
- **state {**  
    <feltétel1>: <utasítás1>  
    <feltétel2>: <utasítás2>  
    ...  
    <feltételn>: <utasításn>  
    **else** <utasításelse>  
}

Az automata egy állapotának leírására szolgál. A program kiértékeli <feltétel1>-et, és ha igaz, végrehajtja <utasítás1>-et. Ha a feltétel hamis volt, megvizsgálja <feltétel2>-t stb.. Ha ugyanaz a feltétel többször is szerepel, akkor véletlenszerűen hajtja végre valamelyik előfordulást. A feltétel általában egyetlen jel olvasását jelenti, de nem szükségszerűen. Az **eps** olvasása mindig sikeres, ezekkel hozhatunk létre  $\epsilon$ -átmeneteket. Ha egyik feltétel sem teljesül, végrehajtja <utasításelse>-t. Az **else** ág elhagyható, ez megfelel annak, mintha **else exit;** állna ott (ez a működés a programozás szempontjából szokatlan lehet, de pontosan megfelel az automaták megadásával kapcsolatos konvencióknak).

A fordítóprogram a **state** utasítást valójában **if-else** ágak láncaként értelmezi. Erre az utasításra külön adunk majd példát T-gép és VA esetén.

### Példák

```
1. input(a a b b);
2. read(a)
3. a
4. (a);
5. a || b
```

1. VA esetén beállítja bemenetnek az a a b b szót. T-gép esetén rámásolja a szalagra, és az első karakter fölé állítja a fejet.
2. Ha a bemenet/szalag aktuális betűje a, akkor **true**, különben **false**.
3. Ugyanaz mint fent, vagyis a **read** elhagyható.
4. Ha a bemeneten a-t olvas, akkor folytatja a végrehajtást, különben terminálja a programot. Ebben az esetben (vagyis amikor az olvasás önálló utasításként szerepel

és nem részkifejezésként) zárójelbe kell tenni a szimbólumot. Erre azért van szükség, mert a `sima a`; könnyen összetéveszthető volna az ugró utasítással, aminek ugyanez az alakja.

5. Ha a bemeneten `a`-t vagy `b`-t olvas, az értéke **true**, különben **false**.

### 2.4.5 Turing-gép

A programunkban néhány kivétellel használhatjuk a korábban felsorolt típusokat és utasításokat, bizonyos esetekben megváltozott jelentéssel. Az eltérések:

- A **start** típusnak T-gép esetén nincs jelentése, az így definiált szimbólumot **nterm** típusúnak tekinti a fordító.
- A **term**-ként deklarált jelek alkotják a „bemenő jelek ábécé-jét”.
- Az **nterm**-ként deklarált jelek hozzáadódva a „bemenő jelek ábécé-jéhez” megadja a szalagábécé elemeit. A `_` karakter (blank) alapértelmezés szerint része ennek az ábécének.
- Nem használható a helyettesítés művelet. (vagyis az „`=`” operátor)

#### 2.4.5.1 Specifikus utasítások

- **left(<term>);**  
A `<term>`-et kiírja a szalagra, majd balra lépteti a fejet.
- **left;**  
Ha a szalagon **term** típusú betű olvasható, akkor visszaírja a szalagra ezt a betűt, azután balra lép. Vagyis eredményét tekintve egyszerűen balra lép, függetlenül a szalag tartalmától. Ha az olvasott betű **nterm** típusú, akkor a program terminál. (a tapasztalatok szerint ez a működés felel meg leginkább a gyakorlatnak, és eredményezi a legkevesebb redundáns állapotátmenetet)
- **right(<term>);**  
A `<term>`-et kiírja a szalagra, majd jobbra lépteti a fejet.
- **right;**  
Jobbra lép, függetlenül a szalag tartalmától, avagy terminál. (ld. **left** utasítás)
- **stand(<term>);**  
A `<term>`-et kiírja a szalagra, a fej állva marad.
- **stand;**  
A fej állva marad.

#### Példa

```
q0: state
{
    a: { left; q1; }
    b || c: { right(d); q2; }
    eps: { right(e); q3; }
}
```

A következő lehetőségek közül véletlenszerűen végrehajtja valamelyiket:

- Ha a szalagon a fej alatt „a” olvasható, a fej balra lép, és ugrik a q1 címkére.
- Ha b vagy c olvasható, d-t ír a szalagra, jobbra lép majd q2 címkére ugrik.
- Bármilyen is olvasható a szalagon, e-t ír a szalagra, jobbra lép és ugrik q3-ra.

- Ha a szalagon olyan jel olvasható mely nem szerepel a többi olvasó utasításban (pl. e), akkor az automata elutasító állapotba ugrik, és terminál.

Összefoglalva: a fenti kód végeredményben a Turing-gép egy  $q_0$  állapotát írja le, melynek átmenetei:

$$\begin{array}{ll} \delta(q_0, a) = (q_1, a, \text{left}) & \delta(q_0, b) = (q_2, d, \text{right}) \\ \delta(q_0, c) = (q_2, d, \text{right}) & \delta(q_0, \varepsilon) = (q_3, e, \text{right}) \end{array}$$

Valamint: minden egyéb esetben a program terminál, vagyis az állapotnak nincs több átmenete.

A fenti példában a **state** utasítás címkéje egy egyszerű utasításcímke, amit viszont érdemes magával az állapottal azonosítani.

#### 2.4.5.2 Állapotok és állapotátmenetek

A Turing-gép minden állapotátmenete három műveletet tartalmaz: olvasás, írás, léptetés. A Prolan-ban az írás és a léptetés csak együtt hajtható végre (ez a gyakorlat szempontjából nem jelent valódi megszorítást). A fordítóprogram feladata megtalálni az összetartozó olvasó ill. író/léptető műveleteket, és ezekből meghatározni az automata állapotait és átmeneteit. Minden átmenetet pontosan egy író/léptető művelet zár le (**left**, **right** vagy **stand**), de tetszőleges számú olvasó műveletet tartalmazhat. Néhány példa átmeneteket definiáló kódra:

```
1. (a); left(b);
2. (a); left;
3. (a); left; left;
4. (a || b); left(d);
5. (a || b); left;
6. (a && b)
7. if (! a) { (b || c); left; }
8. while (! _) right;
9. q1: if (a) q2;
   q2: if (a) q1;
10. q1: if (a) { stand; q2; }
    q2: if (a) { stand; q1; }
```

- $\delta(q, a) = (q, b, \leftarrow)$
- $\delta(q, a) = (q, a, \leftarrow)$
- $\delta(q_1, a) = (q_2, a, \leftarrow)$   
 $\delta(q_2, x) = (q_3, x, \leftarrow)$ , ahol x felveszi az összes **term** típusú szimbólumot.
- $\delta(q_1, a) = (q_2, d, \leftarrow)$ ,  $\delta(q_1, b) = (q_2, d, \leftarrow)$
- $\delta(q_1, a) = (q_2, a, \leftarrow)$ ,  $\delta(q_1, b) = (q_2, b, \leftarrow)$
- Az  $(a \ \&\& \ b)$  tkp. egy **false** utasításnak felel meg, hiszen az automata nyilvánvalóan nem olvashat egyszerre két különböző szimbólumot.
- $\delta(q_1, b) = (q_2, b, \text{left})$ ,  $\delta(q_1, c) = (q_2, c, \text{left})$
- $\delta(q_1, x) = (q_2, x, \text{right})$ , ahol x tetszőleges **term** szimbólum, kivéve a „\_”-t.
- Ez a kód érvénytelen. Látható, hogy ha a szalagon „a” olvasható, akkor a gép végtelen ciklusba esik. A Prolan esetén ez a végtelen ciklus már a fordítás szakaszában létrejön, mivel a fordító az olvasó utasításokat nem képes összepárosítani író/léptető műveletekkel. Ez a működés tkp. elvárható, hiszen a kódhoz éppen az említett okból nem rendelhető Turing-gép.

10. Ez a kód csak kevésben különbözik az előzőtől, de már érvényes (habár a végtelen ciklus ezúttal a működés során jelentkezik majd).

$$\delta(q_1, a) = (q_2, a, \text{stand}), \quad \delta(q_2, a) = (q_1, a, \text{stand})$$

### 2.4.5.3 Példaprogram

A T-gép fordításának részleteit legjobban a példafájlok tanulmányozásával érthetjük meg. A teljesség kedvéért viszont álljon itt egy komplett T-gépet leíró kód, amely pontosan azokat a 0-1 sorozatokat fogadja el, melyekben páros számú 1-es található.

```
#pragma turing
term 0, 1;
void main()
{
    input(0 0 1 1 0);
q0: state
{
    0: { right; q0; }
    1: { right; q1; }
    _: accept;
}
q1: state
{
    0: { right; q1; }
    1: { right; q0; }
}
}
```

Állapotok halmaza:	$\{q_0, q_1, \text{accept}, \text{exit}\}$
Kezdőállapot:	$q_0$
Elfogadó állapot:	$\text{accept}$
Elutasító állapot:	$\text{exit}$
Bemenő jelek ábécéje:	$\{0, 1\}$
Szalagábécé:	$\{0, 1, \_ \}$
Átmenetek:	
$\delta(q_0, 0) = (q_0, 0, \rightarrow)$	$\delta(q_0, 1) = (q_1, 1, \rightarrow) \quad \delta(q_0, \_) = (\text{accept}, \_, \rightarrow)$
$\delta(q_1, 0) = (q_1, 0, \rightarrow)$	$\delta(q_1, 1) = (q_0, 1, \rightarrow)$

### 2.4.6 Végtes automata

A Prolan Turing-gép segítségével képes a végtes automaták modellezésére: Az automata bemenetét egy T-gép bemenetének tekintjük, a bemenet első karakterére állítjuk a T-gépet, majd jobbra lépegetve egyenként beolvassuk a bemenetet, és végrehajtjuk a végtes automata átmeneteit. Az automata akkor fogadja el a bemenetet, ha a bemenet végigolvasása után (vagyis amikor már  $\_$  karaktert olvas) elfogadó állapotban van.

Az **exit** címke nem használható, mivel egy végtes automata nem terminálhat azelőtt, hogy végigolvasta a bemenetet. Ha mégis ilyen működést akarunk megvalósítani, arról a felhasználónak kell gondoskodnia (pl. „csapda” állapotokkal).

A fentieknek megfelelően a tennivalók:

- Használjuk a **%pragma turing** fordítási direktívát

- Minden állapotváltás előtt adjunk ki egy **right**; utasítást.
- Minden elfogadó állapothoz adjunk hozzá egy **\_: accept**; átmenetet (ill. ha nem **state** utasításokkal dolgozunk, ennek megfelel egy **(\_); accept**; szekvencia is). Egyéb helyen ne használjunk **accept** címkét, valamint sehol se használjunk **exit**-et.

A 2.4.5.3 fejezetben leírt T-gép egyben tekinthető véges automatának is, és megfelel a fent leírt szabályoknak.

#### 2.4.7 Veremautomata

- VA esetén a **start** típussal a verem kezdőszimbólumát adhatjuk meg.
- A **term** típusú jelek adják a bemenő jelek ábécéjét, az **nterm** típusúakkal kiegészítve megkapjuk a veremábécét.
- A helyettesítés művelet jelentése eltér a PNY-hez képest. Alakja: **<betű>=<szó>**. A **<betű>**-t leolvassuk a verem tetejéről, és helyére a **<szó>**-t írjuk. Igaz értékkel tér vissza, ha a szabály végrehajtható, vagyis a verem tetején **<betű>** volt, különben hamissal. A **<szó>** verembe írását úgy kell értelmezni, hogy a szó betűit jobbról balra haladva egymás után betesszük a verembe, vagyis a szó első betűje lesz végül a verem tetején.
- Az **exit** címke nem használható, mivel egy VA nem terminálhat azelőtt, hogy végigolvasta a bemenetet. Ha mégis ilyen működést akarunk megvalósítani, arról a felhasználónak kell gondoskodnia (pl. „csapda” állapotokkal). Hasonló okból az **accept** csak speciális esetekben használható (ld. megállási feltételek).

#### Példák

```

1. a = a b c;
2. (a || b) && (c = e)
3. a && c = d; e = f;
4. q0: state
   {
     a:      { c = d; q1; }
     b || c: { e = f; q2; }
     eps:    { a = b; q3; }
   }

```

1. A verem tetején lévő *a* helyére *a b c*-t ír. Az írás sorrendje *c, b, a*, vagyis a verem tetején végül *a* lesz.
2. Ha a bemenet aktuális betűje *a* vagy *b*, akkor a verem tetején lévő *c*-t lecseréli *e*-re. Ha a bemenet nem *a* vagy *b*, ill. ha a veremtető nem *c*, akkor **false** értékkel tér vissza, és nem csinál semmit.
3.
  - Ha a bemeneten *a*-t olvas, akkor ellenőrzi a veremtetőt, különben a program terminál. Ha a veremtető *c*, akkor *d*-re cseréli (majd folytatja a köv. ponton), különben a program terminál.
  - Ha a veremtető *e*, akkor kicseréli *f*-re, különben a program terminál.
4. Ha a bemeneten *a*-t olvas, ellenőrzi a veremtetőt: ha a veremtető *c*, lecseréli *d*-re, majd ugrik a *q1* címkére. Hasonlóan a többi átmenetre is. Összefoglalva: a fenti kód végeredményben a veremautomata egy *q0* állapotát írja le, melynek átmenetei:
 
$$\delta(q_0, a, c) = (q_1, d) \qquad \delta(q_0, b, e) = (q_2, f)$$



$$\delta(q_0, c, e) = (q_2, f) \quad \delta(q_0, \varepsilon, a) = (q_3, b)$$

Valamint: minden egyéb esetben a program terminál, vagyis az állapotnak nincs több átmenete.

#### 2.4.7.1 Megállási feltételek

A megállási feltétel szempontjából kétféle veremautomatát támogat a program:

**Végállapottal elfogadó:** Miután az automata végigolvasta a bemenetet (ami a Prolan esetén annyit tesz, hogy `_` jelet olvas), ha elfogadó állapotban van, akkor elfogadja a bemenetet, különben elutasítja. Ezt a viselkedést úgy tudjuk szimulálni, hogy az elfogadó állapotokhoz hozzáveszünk egy `_`: **accept**; utasítást. (ez megegyezik a véges automatáknál leírtakkal). Végállapottal elfogadó veremautomata esetén a **#pragma pushdown** fordítási direktívát kell használnunk.

**Üres veremmel elfogadó:** Miután az automata végigolvasta a bemenetet, ellenőrzi, hogy a verem tartalma üres-e (semmi nem szerepelhet benne, még a verem kezdőszimbólum sem). Ha üres, elfogadja a bemenetet, ha nem, elutasítja. Ezt a viselkedést úgy tudjuk szimulálni, hogy **#pragma epushdown** direktívát használunk, valamint nem használjuk az **accept** címkét egyszer sem.

#### 2.4.7.2 Példaprogram

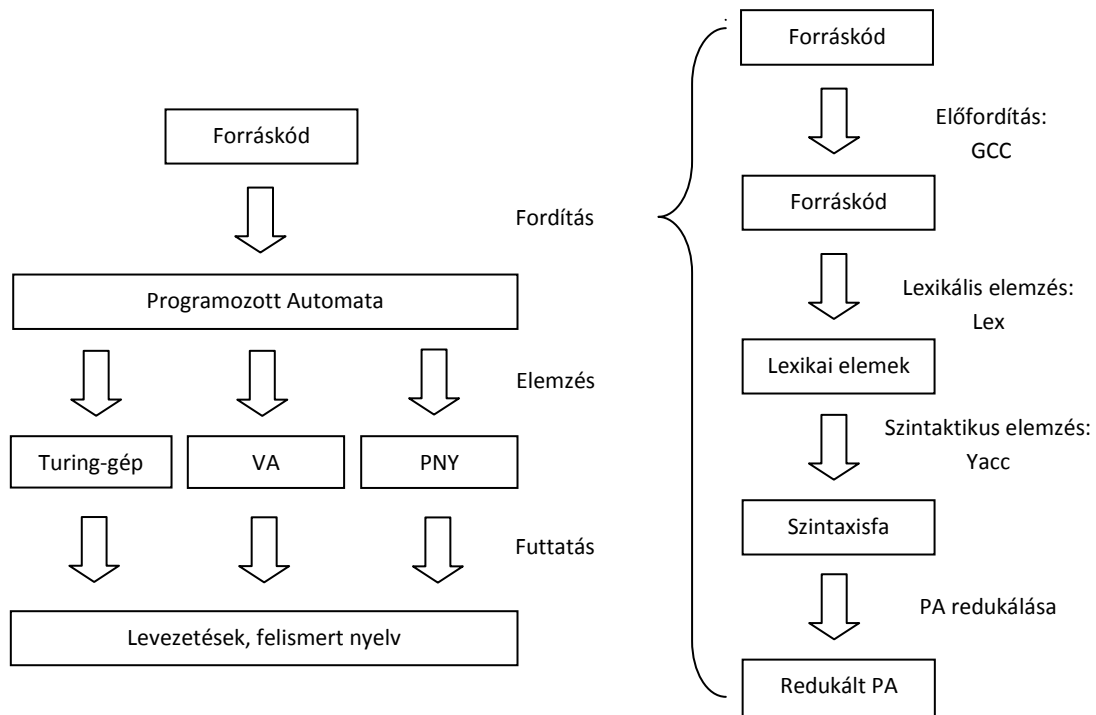
Az alábbi egy üres veremmel elfogadó veremautomata, mely felismeri a helyes zárójelezések nyelvét.

```
#pragma epushdown
start Z;
term 0, 1;
void main()
{
    input(0 1 0 0 1 0 1 1);
    q0: state
    {
        0: { Z = 0 Z | 0 = 0 0; q0; }
        1: { 0 = eps; q0; }
        eps: { Z = eps; q0; }
    }
}
```

Állapotok halmaza:	$\{q_0\}$	
Kezdőállapot:	$q_0$	
Átmenetek:	$\delta(q_0, 0, Z) = (q_0, 0Z),$	$\delta(q_0, 0, 0) = (q_0, 00),$
	$\delta(q_0, 1, 0) = (q_0, \varepsilon)$	$\delta(q_0, \varepsilon, Z) = (q_0, \varepsilon)$

### 3 Fejlesztői dokumentáció

A ProLan egy interpreter, vagyis a forráskódot futtatja, nem hoz létre gépi kódot. Futtatás előtt lefordítja a forráskódot egy közbenső reprezentációs szintre (programozott automatára), majd ezt „futtatja le”, vagyis szimulálja a PNY, T-gép és VA működését. A fordítás és futtatás fenti kombinációja hasonló pl. a Perl, Python, MATLAB és a Ruby esetén.



7. ábra: Az interpreter működése

#### 3.1 Fogalmak, algoritmusok

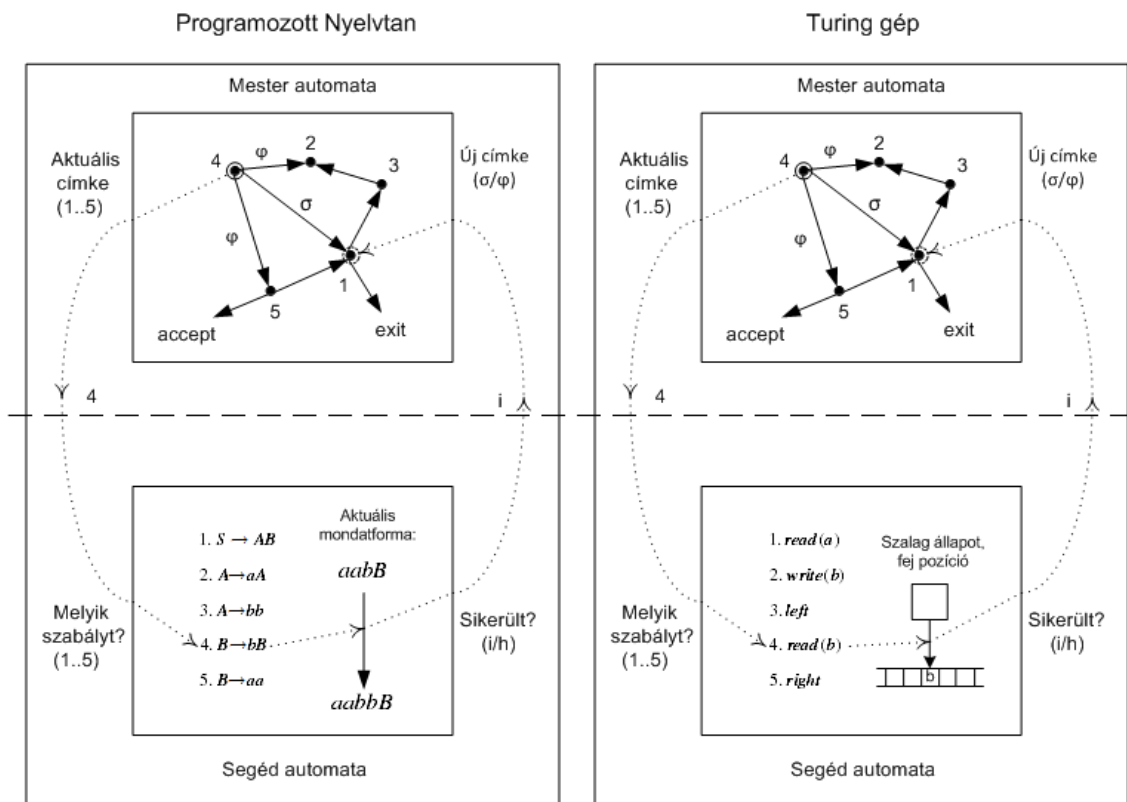
Ebben az alfejezetben a fordítóprogramban felhasznált fogalmakat és algoritmusokat részletezem. Ez a rész elméleti jellegű, független az implementációtól. A többi alfejezet az interpreter és a felhasználói felület konkrét implementációjával foglalkozik.

##### 3.1.1 Programozott automata

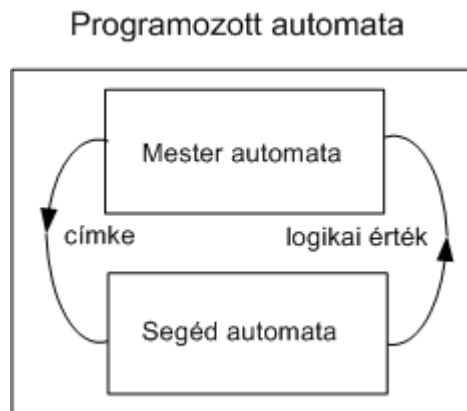
Célunk egy olyan közbenső reprezentációs szint beiktatása, amelyig a PNY-ok, T-gépek és VA-k fordítása megegyezhet, ezzel növelve a kód centralizáltságát, valamint nyitva hagyva a lehetőséget további speciális nyelvtanok, automaták rendszerhez való könnyű hozzáadása előtt. Ez a belső reprezentációs szint egy „programozott automatát” fog jelenteni, ami egy a programozott nyelvtanokhoz hasonló táblázattal adható meg: címkékkel,  $\sigma$  és  $\varphi$  halmazokkal. A fő különbség, hogy ezúttal a címkékhez tartozó utasítások nem csak nyelvtani szabályok lehetnek, hanem gyakorlatilag bármilyen utasítás. (megj.: a programozott automata fogalma egy ad hoc elnevezés, melynek bevezetését egyedül a fordítási folyamat leírása tette szükségessé).

Kiindulásul tekintsük a programozott nyelvtanokat. Egy PNY alapvetően két dolgot tesz: végrehajtja az aktuális címkéhez tartozó nyelvtani szabályt, majd ennek a szabálynak a végrehajthatóságától függően választ egy új címkét a  $\sigma$  vagy a  $\varphi$  halmazból. Ezt a két

műveletet válasszuk külön: az első, vagyis a nyelvtani szabályok végrehajtása legyen egy „segéd automata” (SA) feladata, a második pedig, vagyis a címkék közötti lépegetés pedig egy „mester automatáé” (MA). Az elvet az alábbi ábra szemlélteti:



8. ábra: A programozott nyelvtan és T-gép működésének felbontása azonos elvek szerint.



9. ábra: A programozott automata sémája.

A SA bemenetként egy címkét kap, kimenete pedig egy logikai érték. A gép többi része számára közömbös, hogy a SA-nak milyen belső állapotai vannak, és azokon milyen átmeneteket végez. Ezért könnyen lecserélhető úgy, hogy belső állapotában ne mondatformákat, hanem pl. egy T-gép szalagjának állapotát és a fej pozícióját tárolja, és a címkék ne nyelvtani szabályokat, hanem a T-gépen végzendő író/olvasó műveleteket kódoljanak. Valamint ekkor a MA címkéi a T-gép fejének belső állapotát kódolja (a megfeleltetés nem triviális, ld. később a T-gép

fordítása fejezetben). Veremautomata esetén szintén elvégezhető a fenti felbontás: MA kódolja az automata belső állapotát, SA pedig a bemenetet és a verem tartalmát.

**Definíció:** Segéd automatának (SA) nevezünk egy  $\langle L, Q, \delta \rangle$  hármast, ahol  $L$  egy véges halmaz (a címkék halmaza),  $Q$  halmaz (az automata állapotainak halmaza),  $\delta: Q \times L \rightarrow 2^Q \times \{true, false\}$  állapotátmeneti függvény.

**Definíció:** Mester automatának (MA) nevezünk egy  $\langle L, \sigma, \varphi, Jump \rangle$  négyest, ahol  $L$  egy véges halmaz (a címkék halmaza),  $\sigma: L \rightarrow 2^{L'}$ ,  $\varphi: L \rightarrow 2^{L'}$ ,  $Jump: L \rightarrow 2^{L'}$  függvények, amik minden címkéhez hozzárendelnek egy címkehalmazt, ahol  $L' = L \cup \{exit, accept\}$ .

**Megjegyzés:**  $L'$  mindig az *exit* és *accept* elemekkel kibővített címkék halmazát jelöli a továbbiakban.

**Megjegyzés:** A MA a PNY-hez képest két fő dologban különbözik: nincs megadva semmilyen ábécé ill. nincsenek benne nyelvtani szabályok, valamint a  $\sigma, \varphi$  függvények mellett megjelenik egy hasonló alakú *Jump* függvény, ami a feltétel nélküli ugrás leírására hivatott. Ennek jelentősége majd a fordítás leírásakor válik világossá.

**Definíció:** Programozott automatának (PA) nevezünk egy  $\langle M, S, l_0, q_0 \rangle$  négyest, ahol  $M$  mester automata,  $S$  segéd automata,  $l_0 \in L_M$  kezdő címke és  $q_0 \in Q_S$  kezdő állapot. Kikötjük, hogy  $L_M = L_S$ , vagyis a két automata címkehalmaza megegyezik. Az automata konfigurációinak halmaza:  $L'_M \times Q_S$  (vagyis a mester és a segéd automata konfigurációinak direktszorzata).

**Megjegyzés:** Mivel kikötöttük, hogy a két címkehalmaz közös, valamint „állapotai” csak a segéd automatának vannak, a továbbiakban egyértelműen beszélhetünk egy programozott automata címkeiről és állapotairól, amit  $L$ -el és  $Q$ -val jelölünk.

**Definíció:** Tekintsünk egy programozott automatát. Legyen  $l \in L, m \in L'$  címkék, és  $q, r \in Q$  állapotok. Ekkor az  $(l, q)$  konfigurációból közvetlenül elérhető  $(m, r)$ , ha:

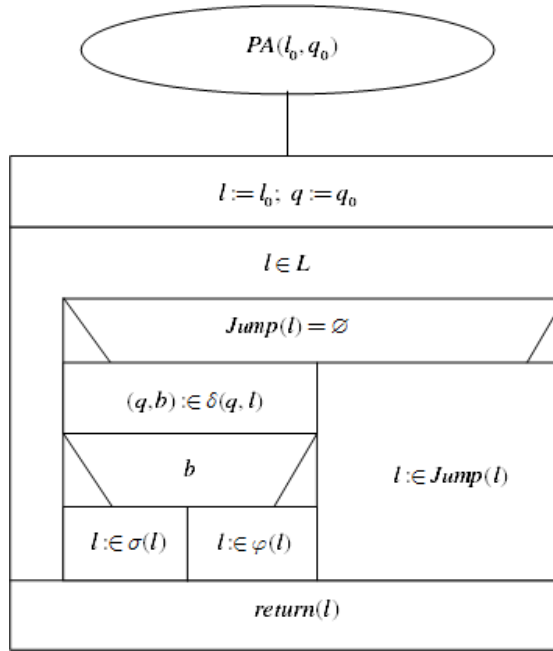
$$m \in Jump(l) \wedge r = q \vee \\ Jump(l) = \emptyset \wedge [(r, false) \in \delta(l, q) \wedge m \in \sigma(l) \vee (r, true) \in \delta(l, q) \wedge m \in \varphi(l)]$$

**Definíció:** Azt mondjuk, hogy az  $(l, q) \in L' \times Q$  megállási konfiguráció, ha  $l \in \{accept, exit\}$ . Ha  $l = accept$  akkor elfogadó konfiguráció, ha  $l = exit$  akkor elutasító.

**Definíció:** Tekintsünk egy programozott automatát és rögzítsünk egy  $l_0 \in L$  kezdőcímkét. Azt mondjuk, hogy az automata felismeri a  $q \in Q$  állapotot, ha az  $(l_0, q)$  konfigurációból elérhető egy elfogadó konfiguráció.

**Megjegyzés:** A fentiekben „állapot felismeréséről” és nem szavakéről beszéltünk. Ez azért van, mert a SA nem szavakon végez műveleteket, csak állapotai vannak. Viszont SA állapotaiban és átmeneteiben nyilvánvalóan kódolhatjuk pl. egy T-gép szalagját, és a szalagon végezhető műveleteket. A különböző automaták definíciói (T-gép, VA stb.) tkp. pont ennek a kódolásnak a leírását tartalmazzák, ezért ezektől most eltekintünk.

A programozott automata működését az alábbi struktogram szemlélteti:



10. ábra Programozott automata végrehajtásának struktogramja

#### Magyarázat

1. Beállítjuk a mester automata kezdőcímkéjét, és a segéd automata kezdőállapotát.
2. Ha az  $l$  címke nem eleme  $L$ -nek (ez csak  $l \in \{exit, accept\}$  esetén lehet), ugrunk a 7. pontra.
3. Ha az  $l$  címke  $Jump$  halmaza nem üres (vagyis a címkéhez egy feltétel nélküli ugrás tartozik), választunk egy új elemet  $Jump(l)$ -ből, és visszaugrunk a 2. pontra.
4. Választunk egy új  $q$  állapotot a  $\delta(q, l)$  függvény által meghatározott halmazból. Ez a függvény megad egy logikai értéket is, amit eltárolunk  $b$ -ben. Ha ez az érték igaz, ugrunk az 5. pontra, ha hamis, a 6.-ra.
5. Választunk egy új címkét a  $\sigma(l)$  halmazból, és ugrunk 2.-re.
6. Választunk egy új címkét a  $\varphi(l)$  halmazból, és ugrunk 2.-re.
7. A függvény visszatér az  $l$  címkével, aminek értéke *accept* vagy *exit* lehet. Ezek jelentése: a kezdőkonfigurációt elfogadta avagy elutasította az automata.

**Példa** Tekintsük az alábbi **PNY**-t:

Címke	Szabály	$\sigma$	$\varphi$
$f_1$	$S \rightarrow ASB$	$\{f_1, f_2\}$	$\{exit\}$
$f_2$	$A \rightarrow a$	$\{f_2\}$	$\{f_3\}$
$f_3$	$B \rightarrow b$	$\{f_3\}$	$\{exit\}$

Ekkor felírhatjuk a vele ekvivalens **PA**-t. A **mester automata**:

Címke	$\sigma$	$\varphi$	$Jump$
$f_1$	$\{f_1, f_2\}$	$\{exit\}$	$\emptyset$
$f_2$	$\{f_2\}$	$\{f_3\}$	$\emptyset$
$f_3$	$\{f_3\}$	$\{exit\}$	$\emptyset$

**Megjegyzés:** Mivel a  $Jump$  halmazoknak nincs megfelelője a PNY-ben, ezeket egyszerűen üresnek választottuk.

### Segéd automata:

- Q (állapotok): az  $\{S, A, B, a, b\}$  ábécé feletti nyelv.
- Címkék:  $\{f_1, f_2, f_3\}$
- Átmenetek:  $\delta(q, l) = \begin{cases} (q, false) & , ha f(l, q) = \emptyset \\ \{(r, true) \mid r \in f(l, q)\} & , különben \end{cases}$

Ahol  $f(f_1, q)$  a  $q$  állapothoz tartozó szóból az  $S \rightarrow ASB$  helyettesítéssel levezethető szavak halmaza, valamint  $f(f_2, q)$  és  $f(f_3, q)$  pedig a  $A \rightarrow a$  ill. a  $B \rightarrow b$  helyettesítéssel levezethető szavak halmaza.

T-gép esetén a PA-ra átírásra a 3.1.3 fejezetben mutatunk példát.

### 3.1.2 Programozott automata redukálása

A szintaktikus elemzés eredményeként megkaptuk a szintaxisfát. Belátható, hogy a szintaxisfa is tekinthető PA-nak, és a megszokott vezérlő, ugró utasítások szemantikája tökéletesen leírható a  $\sigma$ ,  $\varphi$  és  $Jump$  halmazokra vonatkozó szabályokkal. Tekintsük pl. a következő PNY-t kódoló PA-t:

Címke	Jelentés	$\sigma$	$\varphi$	$Jump$
$f_1$	<b>if</b> (A=a) B=b;	$\{f_4\}$	$\{exit\}$	$\{f_2\}$
$f_2$	A=a;	$\{f_3\}$	$\{f_4\}$	$\emptyset$
$f_3$	B=b;	$\{f_4\}$	$\{exit\}$	$\emptyset$
$f_4$	...	...	...	...

### Megjegyzések:

- Az A=a kifejezés jelentése: végrehajtja az aktuális mondatformán az  $A \rightarrow a$  helyettesítést. A kifejezés értéke megegyezik azzal, hogy a helyettesítés végrehajtható volt-e.
- Az  $f_4$  annak a fel nem tüntetett utasításnak a címkéje, ami az **if** (A=a) B=b;-t követi.
- A SA-t nem tüntettük fel, ez a korábbiakban már leírt nyelvtani levezetések végrehajtó automata lesz. A SA természetesen nem tudna mit kezdeni az  $f_1$  címkéjű utasítással, viszont a PA végrehajtásának definíciója alapján ezt a címkét soha nem kapja bemenetként, mivel a  $Jump$  halmaza nem üres.

A fenti táblázat két dologban különbözik egy PNY-tól: szerepel benne egy  $Jump$  oszlop, valamint szerepelnek olyan utasítások, melyek jelentése nem  $A=a$  alakúak, vagyis nem nyelvtani szabályok. Ha el tudjuk érni, hogy ez utóbbi szabályokat kiszűrjük, valamint a  $Jump$  oszlop tartalma mindenhol  $\emptyset$  legyen (ez a két feltétel valójában egyet jelent), akkor ezzel megkaptuk a PA által kódolt PNY-t.

Kiindulásul tehát a szintaxisfa elemeit is egy PA utasításainak tekintjük:

- A fa minden csúcsát ellátjuk címkével.  
(az implementációban ez egyszerűen a csúcs memóriabeli címe)
- A fa minden csúcsát ellátjuk  $\sigma$ ,  $\varphi$  és  $Jump$  halmazokkal, melyeket ha megfelelően kitöltünk, a fa jelentésének megfelelően működő PA-t kapunk. Ez a program tehát a futása során a szintaxisfa csúcsain „ugrál”.

- Az áttekinthetőség kedvéért  $\sigma(A)$  helyett az  $A.\sigma$  jelölést használjuk.

A további tárgyalás megkönnyítése érdekében bevezetünk néhány szóhasználatot:

- A PA címkéit a PA utasításainak is nevezhetjük.
- Műveleti utasításnak nevezzük azokat az  $l \in L$  címkéket, melyekre  $Jump(l) = \emptyset$
- Vezérlő utasításnak nevezzük azokat az  $l \in L$  címkéket, melyekre  $Jump(l) \neq \emptyset$

### 3.1.2.1 Vezérlő utasítások

Először tekintsük a vezérlő utasításokra vonatkozó szabályokat. Az alábbi táblázat a szintaxisfa csúcsai és gyerek-utasításai közötti összefüggéseket írja le, ahol  $P$  (parent) a csúcs,  $A, B, C$  pedig a gyerekei.

if (A) B;	$P.Jump = \{A\}$	$A.\sigma = \{B\}, A.\varphi = P.\varphi, B.\sigma = P.\sigma, B.\varphi = P.\varphi$
if (A) B;	$P.Jump = \{A\}$	$A.\sigma = \{B\}, A.\varphi = \{C\}, B.\sigma = P.\sigma, B.\varphi = \{exit\}$
else C;		$C.\sigma = P.\sigma, C.\varphi = exit$
while (A) B;	$P.Jump = \{A\}$	$A.\sigma = \{B\}, A.\varphi = P.\sigma, B.\sigma = \{A\}, B.\varphi = \{exit\}$
all A;	$P.Jump = \{A\}$	$A.\sigma = \{A\}, A.\varphi = P.\sigma$
try A;	$P.Jump = \{A\}$	$A.\sigma = P.\sigma, A.\varphi = P.\sigma$
{A; B;}	$P.Jump = \{A\}$	$A.\sigma = \{B\}, A.\varphi = \{exit\}, B.\sigma = \{C\}, B.\varphi = \{exit\}$
[A; B;]	$P.Jump = \{A, B\}$	$A.\sigma = \{A, B\}, A.\varphi = \{exit\}, B.\sigma = \{A, B\}, B.\varphi = \{exit\}$
<A; B;>	$P.Jump = \{A, B\}$	$A.\sigma = P.\sigma, A.\varphi = \{exit\}, B.\sigma = P.\sigma, B.\varphi = exit$
A && B	$P.Jump = \{A\}$	$A.\sigma = \{B\}, A.\varphi = P.\varphi, B.\sigma = P.\sigma, B.\varphi = P.\varphi$
A    B	$P.Jump = \{A\}$	$A.\sigma = P.\sigma, A.\varphi = \{B\}, B.\sigma = P.\sigma, B.\varphi = P.\varphi$
A B	$P.Jump = \{A, B\}$	$A.\sigma = P.\sigma, A.\varphi = P.\varphi, B.\sigma = P.\sigma, B.\varphi = P.\varphi$
!A	$P.Jump = \{A\}$	$A.\sigma = P.\varphi, A.\varphi = P.\sigma$
exit	$P.Jump = \{exit\}$	
accept	$P.Jump = \{accept\}$	
return	$P.Jump = CALL.\sigma$	$CALL = \text{a függvényhívás címkéje}$
return A	$P.Jump = \{A\}$	$A.\sigma = CALL.\sigma, A.\varphi = CALL.\varphi$
true	$P.Jump = P.\sigma$	
false	$P.Jump = P.\varphi$	
goto l1, l2;	$P.Jump = \{l1, l2\}$	
lab1: A	$P.Jump = \{A\}$	$A.\sigma = P.\sigma, A.\varphi = P.\varphi$
SKIP	$P.Jump = P.\sigma$	
void main(){A}	$P.Jump = \{A\}$	$P.\sigma = \{exit\}, P.\varphi = \{exit\}, A.\sigma = P.\sigma, A.\varphi = P.\varphi$

**Példa:** Tekintsük a  $P \equiv A \&\&B$  logikai kifejezést (vagyis egy  $P$  csomópontot melynek gyerekei  $A$  és  $B$ ):

- Először is ki kell értékelni az  $A$  utasítást. Ez azt jelenti, hogy ha a  $P$  csúcsra kerül a vezérlés, feltétel nélkül ugrani kell  $A$ -ra. Vagyis:  $P.Jump = \{A\}$
- Ha  $A$  igaz, ugrunk  $B$ -re:  $A.\sigma = \{B\}$
- Ha  $A$  hamis, ugrani kell oda, ahova  $P$ -nek kell ugrania ha értéke hamis, vagyis  $A.\varphi = P.\varphi$
- Ha  $B$  igaz, ugrani kell oda, ahova  $P$ -nek kell ugrania ha értéke igaz, vagyis  $B.\sigma = P.\sigma$
- Ha  $B$  hamis, ugrani kell oda, ahova  $P$ -nek kell ugrania ha értéke hamis, vagyis  $B.\varphi = P.\varphi$

Látható, hogy a fenti szabályok alkalmazásakor a  $P.\sigma$  és  $P.\varphi$  halmazokat adottnak kell tekintenünk, és segítségükkel tudjuk meghatározni az  $A$  és  $B$   $\sigma, \varphi$  halmazait. Vagyis a  $\sigma, \varphi$

halmazok valójában a szintaxisfa öröklött attribútumai, az információ felülről lefelé áramlik. A *Jump* meghatározásához viszont adottnak kell tekintenünk a gyerek csúcsok  $\sigma$  és  $\varphi$  halmazait, vagyis a *Jump* egy szintetizált attribútum, aminek meghatározása csak a  $\sigma$  és  $\varphi$  után következhet (ld. pl. a **true** utasítást).

$P$  az elvárt módon működik, a rész kifejezéseinek értékétől függően a  $\sigma$  avagy a  $\varphi$  halmazának elemeire ugrik. Ez az „érték” többféleképpen jöhet létre:  $A$  és  $B$  lehetnek műveleti utasítások (ld. következő fejezet), vagy további összetett utasítások is. Tekintsünk egy ilyen összetettebb példát:

Legyen  $P \equiv (!C) \&\& (E || F)$

- A  $P$  csomópontnak két gyereke van:  $A \equiv !C$  és  $B \equiv E || F$ .  $A$ -nak egyetlen gyereke  $C$ ,  $B$  gyerekei  $E$  és  $F$ .
- A  $P \equiv A \&\& B$  kifejezésre vonatkozó szabályokat már a fentiekben leírtuk. Első lépésként tehát meghatároztuk  $P$ .*Jump*-ot, valamint  $A$  és  $B$   $\sigma$ ,  $\varphi$  halmazait.
- $A \equiv !C$ 
  - A kiértékelés  $C$  kiértékelésével kezdődik, vagyis  $A$ .*Jump* =  $\{C\}$
  - Ha  $C$  igaz, oda kell ugrani, ahová  $A$  ugrik ha hamis:  $C.\sigma = A.\varphi = P.\varphi$
  - Ha  $C$  hamis, oda kell ugrani, ahová  $A$  ugrik ha igaz:  $C.\varphi = A.\sigma = \{B\}$
- $B \equiv E || F$ 
  - $B$  kiértékelés  $E$  kiértékelésével kezdődik, vagyis  $B$ .*Jump* =  $\{E\}$
  - Ha  $E$  igaz, oda kell ugrani, ahová  $B$  ugrik ha igaz:  $E.\sigma = B.\sigma = P.\sigma$
  - Ha  $E$  hamis, ugunk  $F$ -re:  $E.\varphi = \{F\}$
  - Ha  $F$  igaz, oda kell ugrani, ahová  $B$  ugrik ha igaz:  $F.\sigma = B.\sigma = P.\sigma$
  - Ha  $F$  hamis, oda kell ugrani, ahová  $B$  ugrik ha hamis:  $F.\varphi = B.\varphi = P.\varphi$

Megjegyezzük, hogy tisztán vezérlő utasításokból is alkotható PA. Pl.:

- **main() {}** Ez a kód megfelel egy olyan PA-nak aminek két utasítása van:  $P$  a **main** függvény, és ennek egyetlen gyereke  $A$  egy **skip** utasítás. A szabályokat alkalmazva a következő eredményt kapjuk:  

$$P.\sigma = P.\varphi = A.\sigma = A.\varphi = \{exit\}, P.\text{Jump} = \{A\}, A.\text{Jump} = \{exit\}$$
A PA végrehajtása a szintaxisfa elejéről, vagyis a gyökeréből indul, ez esetben a  $P$  utasításból. A stuktogram szerint a program  $P$ -t végrehajtva választ egy címkét a  $P.\text{Jump} = \{exit\}$  halmazból, majd terminál.
- **false || false && true**. Ezt a kódot lefordítva egy olyan  $P$  utasítást kapunk, melyre:  
 $P.\text{Jump} = P.\varphi$  teljesül. Vagyis akármi is  $P.\varphi$  (ezt a kód környezete határozza meg), ha a végrehajtás  $P$ -re kerül, feltétel nélkül ugrik  $P.\varphi$ -re. Vagyis gyakorlatilag  $P$  attribútumait úgy állítja be a fordító, mintha az egy **false** utasítás lenne.

### 3.1.2.2 Műveleti utasítások

Programozott nyelvtan esetén a műveleti utasítások  $X = a \ b \ c$  alakúak, vagyis nyelvtani szabályok. Ezek minden esetben a szintaxisfa levelei lesznek, ezért nincsenek gyerekei, itt a latin betűk tehát nem gyerek csúcsok, hanem a nyelvtan szimbólumai.  $\sigma$  és  $\varphi$  halmazait



szülő csomópont határozza meg (öröklött attribútumok), valamint *Jump* halmazuk minden esetben üres.

**Példa:**  $P \equiv X = b \parallel X = c$ . A  $P$  csomópontnak két gyereke van, az  $A \equiv X = b$  és  $B \equiv X = c$ . A lusta vagy operátorra vonatkozó szabályoknak megfelelően a fordítóprogram az alábbi attribútumokat állítja be:

$P.Jump = \{A\}$ ,  $A.\sigma = P.\sigma$ ,  $A.\varphi = \{B\}$ ,  $B.\sigma = P.\sigma$ ,  $B.\varphi = P.\varphi$ , valamint  $A.Jump = B.Jump = \emptyset$ .

Hajtsuk végre a  $PA$ -t a  $P$  kezdőutasítással. A program futása a következő lesz:

1. Címke:  $P$ . Mivel  $P.Jump = \{A\}$ , a végrehajtás  $A$ -n folytatódik
2. Címke:  $A$ . Mivel  $A.Jump = \emptyset$ , a segédautomata megpróbálja alkalmazni az állapotában tárolt mondatformára az  $X \rightarrow b$  nyelvtani szabályt. Ha ez sikerül, igaz értékkel tér vissza, és ekkor a következő végrehajtandó utasítás az  $A.\sigma = F.\sigma$  halmazból lesz kiválasztva. Ha nem, hamissal tér vissza, és a végrehajtás az  $A.\varphi = \{B\}$  utasításon folytatódik.
3. Címke:  $B$ . Mivel  $A.Jump = \emptyset$ , a segédautomata megpróbálja alkalmazni az állapotában tárolt mondatformára az  $X \rightarrow c$  nyelvtani szabályt. Ha ez sikerül, igaz értékkel tér vissza, és ekkor a következő végrehajtandó utasítás a  $B.\sigma = F.\sigma$  halmazból lesz kiválasztva. Ha nem, hamissal tér vissza, és a következő címke az  $B.\varphi = F.\varphi$  halmazból kerül kiválasztásra.

### 3.1.2.3 A főprogram

A  $PA$ -ra fordítás algoritmusát először csak a főprogramra adjuk meg, és a következő fejezetben bővítjük ki a makró törzsekre ill. a makróhívások feloldására.

Az algoritmus célja olyan állapot elérése, amiben a műveleti utasítások  $\sigma$  és  $\varphi$  halmazai már csak műveleti utasításokra mutatnak. Ekkor ugyanis a műveleti utasítások halmaza zárt lesz abból a szempontból, hogy a program a futás közben nem lép ki ebből a halmazból, így a vezérlő utasítások mind elhagyhatóak. PNY-ra fordítás esetén ekkor egy olyan  $PA$ -t kapunk, aminek utasításai nyelvtani szabályok, valamint mindegyik *Jump* halmaz üres, ez pedig a korábbiak szerint pontosan egy PNY-nak felel meg, vagyis a fordítás gyakorlatilag véget ért.

#### Az algoritmus kiindulási állapota:

- Adott a szintaxisfa, és a csomópontjainak címkéi. (az implementációban ezek egyszerűen a csomópontok memóriabeli címei.)
- A fa gyökere a *main* függvény:  $main.\sigma = main.\varphi = \{exit\}$ . (a *main* függvény lefutása után a program mindenképp terminál.)
- A többi elem  $\sigma$  és  $\varphi$  halmazai üresek, valamint üres az összes *Jump* halmaz.

#### Az algoritmus lépései:

1. A gyökértől elindulva mélységi bejárással feltöltjük a  $\sigma$  és  $\varphi$  halmazokat a táblázatban leírt szabályoknak megfelelően.
2. Tetszőleges fabejárással beállítjuk a *Jump* halmazokat a szabályoknak megfelelően.
3. Ismételt ugrások egyszerűsítése: Abban az esetben, ha a cél-utasítás (vagyis amire ugrunk) szintén egy vezérlő utasítás, vagyis tovább kell róla ugrani annak *Jump*

halmaza szerint, akkor ezeket az ismételt ugrásokat az algoritmus egyetlen ugrással helyettesíti. Minden  $A$  vezérlő utasításra:

$$A.Jump := (A.Jump \cap MűvUt.) \cup \{y | x \in A.Jump \cap VezUt. \wedge y \in x.Jump\}$$

Ennél a helyettesítésnél figyelni kell, hogy az ugrások gráfja tartalmaz-e kört, ebben az esetben ui. a fordító hibaüzenettel leáll. Az algoritmus ezen lépésének végén tehát minden  $Jump$  halmaz csak műveleti utasításokat tartalmaz, viszont a  $\sigma$ ,  $\varphi$  halmazok még vegyesen mutatnak műveleti és vezérlő utasításokra.

4. Kiküszöböljük a  $\sigma$ ,  $\varphi$  halmazokból a vezérlő utasításokat. Mivel minden vezérlő utasítás a  $Jump$  halmazán keresztül egy műveleti utasításra mutat, ezek a  $Jump$  hivatkozások összevonhatóak a vezérlő utasításra mutató hivatkozásokkal. Vagyis minden  $A$  műveleti utasításra:

$$A.Ro := (A.Ro \cap MűvUt.) \cup \{y | x \in A.Ro \cap VezUt. \wedge y \in x.Jump\}$$

$$A.Fi := (A.Fi \cap MűvUt.) \cup \{y | x \in A.Fi \cap VezUt. \wedge y \in x.Jump\}$$

5. Elhagyhatjuk a kódból a vezérlő utasításokat, mivel egyetlen  $\sigma$ ,  $\varphi$  hivatkozás sem mutat rájuk.

### 3.1.2.4 Makró törzs, makróhívások feloldása

A főprogram esetén  $main.\sigma = main.\varphi = \{exit\}$  feltételből indultunk ki. A makróknak viszont lehet logikai visszatérési értéke, amit a  $\sigma$ ,  $\varphi$  halmazokkal valósítunk meg. Ezért minden makró törzshöz felveszünk két PA-utasítást:  $return\_true$  és  $return\_false$ . Ha a makró belsejében ezekre az utasításokra ugrunk, a futás a makróhívás utasításának  $\sigma$  ill.  $\varphi$  címkein folytatódik. Vagyis:

- $return\_true.Jump := makróhívás.\sigma$
- $return\_false.Jump := makróhívás.\varphi$
- **void** típusú makró esetén:
- $return\_true.Jump := makróhívás.\sigma$
- $return\_false.Jump := makróhívás.\sigma$

A ProLan nyelv **return** A utasítása szolgál a makróból való visszatérésre. Ha az utasítás csomópontja a szintaxisfában  $P$ , akkor az utasítás fordítása a következőképpen történik:

- $P.Jump := \{A\}$
- $A.\sigma := \{return\_true\}$
- $A.\varphi := \{return\_false\}$

A főprogram fordítása során a makrókat is PA-ra fordítjuk, de ez a kód csak egyfajta sablon, amit minden hívás helyére módosítva be kell másolnunk. A makróhívás feloldásának lépései:

1. A lokális változókból új példányokat hozunk létre. Erre minden egyes hívásnál szükség van, nem használhatjuk fel többször ugyanazokat a szimbólumokat lokális változóként.
2. Feloldjuk azokat a makróhívásokat, amiket a makró törzs tartalmaz.
3. A makróhívás  $\sigma$ ,  $\varphi$  halmazait már a hívás helyén, annak kontextusa beállította. Ezeket a halmazokat átmásoljuk a makró törzs  $return\_true$  és  $return\_false$  utasításaiba a korábban leírtaknak megfelelően.
4. A módosított makró törzset bemásoljuk a hívás helyére. A sablon utasításaira mutató hivatkozásokat átírjuk úgy, hogy a példány utasításaira mutassanak.

5. Az így keletkezett PA még tartalmaz ugró utasításokat (pl. *return\_true*), amiket az előző fejezetben leírtak szerint feloldunk.

### 3.1.3 Turing gép fordítása

A T-gép állapotátmenet függvénye  $\delta(q, a) = \{(r, b, d), \dots\}$  alakú, ahol  $q$  az aktuális állapot,  $a$  az olvasott szimbólum,  $r$  az új állapot,  $b$  az írandó szimbólum,  $d$  a fejmozgás iránya. A PNY-ok esetén könnyebbé tették, hogy a forráskód minden műveleti utasításának egyértelműen megfeleltethető volt a PNY pontosan egy állapota. A T-gép esetén az utasítások és az állapotok ill. állapotátmenetek közötti összefüggés bonyolultabb. Pl. a **read(a); write(b); left;** szekvencia három utasításból áll, mégis egyetlen állapotátmenetet jelöl:  $\delta(q, a) = (r, b, left)$ . Az iménti esetben az összevonás egyértelmű, viszont a nemdeterminisztikus ugrásokkal együtt általában nem szekvenciákat kell összevonnunk, hanem gráfokat átalakítanunk. További nehézség, hogy egy állapothoz több átmenet is tartozhat, ezért azt is vizsgálnunk kell, hogy mely átmenetek tartoznak össze és melyek nem. Kezdetben még az állapotok halmaza (vagy annak számossága) sem ismert:

```
if (read(a)) { write(b); left; }
else if (read(b)) { write(c); right; goto lab1; }
write(read(a)); left;
```

a fenti kód a következő „T-gép részletet” írja le:

állapotok:  $\{q, r\}$

állapotátmenetek:

$$\begin{aligned}\delta(q, a) &= (r, b, left) \\ \delta(q, b) &= (p, c, right) \\ \delta(r, *) &= (s, a, left)\end{aligned}$$

Az első két átmenetben közös, hogy ugyanahhoz a  $q$  állapothoz tartoznak. Az utolsó átmenetben csillaggal jelöltük az olvasott szimbólumot, ennek jelentése, hogy minden lehetséges szimbólumra vonatkozik, vagyis ez az egy kifejezés valójában több átmenetet reprezentál.

Az említett nehézségek mind kiküszöbölhetőek volnának, ha az állapotokat és az átmeneteket a forráskódban explicit módon kellene kifejezni, vagyis egy átmenetet egyetlen pl. **transition(q, a, r, b, d)** utasítással. A Prolan erre lehetőséget ad (ld. **state** utasítás), de nem követeli meg, mivel abban az esetben nem használhatnánk programszerkezeteket, se makrókat.

Ahelyett hogy a fenti, legáltalánosabb alakban oldanánk meg a feladatot, érdemes egy egyszerűsítést tenni: Vonjuk össze az író és léptető műveleteket, vagyis alapértelmezés szerint a következő utasításokat használjuk írásra és léptetésre: **left(a)**, **right(a)**. Ezek jelentése: írjunk a szalagra  $a$ -t, majd lépünk balra, valamint írjunk a szalagra  $a$ -t majd lépünk jobbra.

Továbbra is használható a **write(a)** utasítás, de ez minden esetben annak fog megfelelni, hogy írjunk  $a$ -t, majd a fej maradjon állva (tekintet nélkül arra hogy esetleg egy ezt követő léptetéssel összevonható volna). Valamint továbbra is használható a paraméter nélküli **left** és

**right** (amik visszaírják az aktuálisan olvasott szimbólumot, majd léptetnek), ezeket nem érinti a megkötés.

A fenti átalakítás tkp. annyit jelent, hogy az író és léptető műveleteket a programozónak kell párosítania, a fordítóprogram ezzel nem foglalkozik. Ez nem jelent lényeges megkötést, mivel a gyakorlatban egy írás után szinte mindig léptetés következik, nem pedig olvasás vagy újabb írás. Olyan programot talán nem is érdemes íni, ami ilyen kombinációkat is tartalmaz. (a megkötés bevezetése azzal függ össze, hogy ez utóbbi esetek általános elemzése bonyolult volna, és nem eredményezne egyértelmű megoldást). Ugyanez már nem volna igaz az olvasás és írás összevonására, mivel egy olvasást általában több újabb olvasás is követhet, hiszen ha az első olvasás nem sikerült, újabb olvasásokkal kell megtalálnunk az aktuálisan olvasható szimbólumot.

### 3.1.3.1 Adatszerkezetek

- Adottak a PA utasításai, melyeknek címkéit,  $\sigma$ ,  $\varphi$  halmazait a már leírt algoritmussal beállítottuk, valamint kiszűrtük a vezérlő utasításokat. A T-gépet leíró PA-nak kétféle műveleti utasítása van:
  - Olvasó utasítás. Attribútumai:
    - $R$ : olvasott szimbólum, a szalagábécé egy eleme.
    - $S$ : a T-gép egy állapota, amiben a szimbólumot olvassuk. (ez egy segéd attribútum az algoritmus könnyebb leírása érdekében)
  - Író/léptető utasítás. Attribútumai:
    - $W$ : írandó szimbólum, a szalagábécé egy eleme.
    - $D$ : a léptés iránya,  $\{left, right, stand\}$  egy eleme.
- A T-gép állapottai.
- A T-gép állapotátmenetei. Attribútumai  $\delta(A, S, A, R) = (A, N, A, W, A, D)$ , vagyis:
  - $S$ : az állapot amelyhez tartozik.
  - $R$ : olvasott szimbólum.
  - $N$ : az új állapot amelybe mutat.
  - $W$ : írandó szimbólum.
  - $D$ : a léptetés iránya.

**Megjegyzés:** Ha megengedjük, hogy egy állapothoz több olyan átmenet is tartozzon, melyek ugyanazon szimbólumot olvassák, akkor ezzel kifejezhetjük hogy a  $\delta$  függvény többértékű, vagyis hogy a T-gép nemdeterminisztikus.

### 3.1.3.2 Algoritmus

A későbbiekben gyakran hivatkozunk olvasó utasítások egyfajta láncára, ami a forráskódban általában: **if (read(A1))...; else if (read(A2))...; else if (read(A3)) ...; else...;**

alakban jelenik meg. Az olvasások a T-gép ugyanazon állapotában történnek, vagyis a T-gép állapotdiagramját tekintve egy csomópontból és a belőle kiinduló élekről van szó. A PA-beli jellemzője, hogy olvasó utasítások olyan  $A[i]$  sorozata ( $i = 1..n$ ), melyre  $A[i].\varphi = A[i + 1]$  minden  $i = 1..n - 1$  esetén. A későbbiekben ezekre a sorozatokra **else if** láncként fogunk hivatkozni, habár nem csak a fenti alakú forráskódok eredményezhetnek hasonló láncot, egyáltalán nem kell hogy szerepeljen bennük **if** utasítás.

### **Az algoritmus lépései:**

1. Minden  $A$  olvasó utasításhoz vegyünk fel egy új T-gép állapotot, vagyis  $A.S := \text{ÚjÁllapot}()$
  2. **else if** láncok összevonása:  
Legyen  $A$  és  $B$  két olvasó utasítás, melyekre  $B \in A.\varphi$ . Ha  $A.\varphi = \{B\}$ , vagyis  $A.\varphi$  egyetlen eleme  $B$ , valamint  $B$ -re nem mutat egyetlen másik utasítás sem, akkor a korábban leírtaknak megfelelően  $A$  és  $B$  ugyanazon **else if** lánc tagjai, vagyis a hozzájuk tartozó T-gép állapotok megegyeznek. Ekkor tehát a két állapotot össze kell vonni, amit pl. úgy érhetünk el, hogy  $B.S$  minden előfordulását átírjuk  $A.S$ -re, majd  $B.S$ -t kitöröljük.
  3. **else if** láncok teljessé tétele:  
Az előző lépés végén maradhattak olyan olvasó utasítások, melyeket nem vontunk be egyetlen láncba sem. Ezeket egyelemű láncnak tekinthetjük. Vagyis elmondható, hogy minden állapothoz pontosan egy lánc tartozik. Ezek a láncok már erősen hasonlítanak arra ahogyan egy T-gép működik: egy adott állapotban megnézi mi olvasható a szalagról, és annak megfelelően választ egy átmenetet. Az egyetlen probléma a lánc legvégével, vagyis **else** ágával van: ez az ág végrehajtódik minden olyan szimbólum olvasása esetén, melyet a lánc korábbi elemei nem vizsgáltak. Ezt T-géppel csak úgy tudjuk kifejezni, ha minden egyes lehetséges olvasáshoz létrehozunk egy átmenetet, vagyis az **else if** láncot kiegészítjük úgy, hogy minden lehetséges esetet vizsgáljon. Tekintsük minden lánc utolsó  $A$  olvasó utasítását. Két eset lehetséges:
    - 3.1.  $A.\varphi = \{\text{exit}\}$ . Ez azt jelenti, hogy ha olyan szimbólum olvasható, mely nem szerepel a láncban, a programnak elutasító állapotban terminálnia kell. A konvenciónak pont megfelel, ha ezekhez a szimbólumokhoz nem hozunk létre átmenetet: minden „hiányzó él” a T-gép grájfjában alapértelmezés szerint terminálás. Vagyis ebben a pontban sem teszünk semmit.
    - 3.2. Egyébként: Legyen  $H$  a szalagábécé azon betűinek halmaza, melyek nem szerepelnek a lánc egyetlen olvasásában sem. „Hosszabbítsuk meg” a láncot a halmaz elemeire vonatkozó olvasásokkal: minden  $l \in H$  betűre vegyünk fel egy új  $B$  olvasó utasítást:
$$B.\sigma := A.\varphi, \quad B.S := A.S, \quad B.R := l$$
- Megjegyzés:* A 3. lépés után az olvasó utasítások  $\varphi$  halmazait már nem használjuk, ezért  $A.\varphi$  és  $B.\varphi$  beállításaival nem foglalkozunk.
4. Olvasó utasítások átírása átmenetekké:  
Minden  $A$  olvasó utasítás minden  $e \in A.\sigma$  elemére vegyünk fel egy új  $T$  átmenetet:
$$T.S := A.S, \quad T.R := A.R, \quad T.N := e, \quad T.W := \text{NIL}, \quad T.D := \text{NIL}$$
5. Minden olvasó utasításhoz véglegesen megállapításra került hogy a T-gép mely állapothoz tartozik, ezért megtehetjük, hogy minden  $\sigma$  és  $\varphi$  halmazban minden  $A$  olvasó utasítást lecserélünk  $A.S$ -re. Ezen lépés után az olvasó utasításokat már nem használjuk.
6. Ismételt olvasások feloldása:  
Az iménti lépések eredményeként keletkezhetnek olyan átmenetek, melyek valamely szimbólum olvasása után nem végeznek semmilyen írást/léptetést, hanem azonnal egy

másik állapotra mutatnak. Ilyen alakú átmenet csak akkor maradhat a végső automatában, ha  $\varepsilon - t$  olvas, vagyis  $\varepsilon$ -átmenet. Ezeket változatlanul hagyjuk, ellenkező esetben az átmenetet megpróbáljuk lecserélni.

Legyen  $A$  átmenet melyre  $A.R \neq \varepsilon$ .

Minden olyan  $B$  átmenetre melyre  $B.S = A.N$  és  $B.R = A.R$ , vegyünk fel egy új  $T$  átmenetet:

$$T.S := A.S, \quad T.R := A.R, \quad T.N := B.N, \quad T.W := NIL, \quad T.D := NIL$$

Majd ezután töröljük  $A$ -t.

Ezt a lépést addig ismételjük, amíg történik változás. Előfordulhat, hogy az algoritmus soha nem ér véget, pl. akkor, ha két állapot ugyanazt a szimbólumot olvasva egymásra ugrik. Ez gyakorlatilag egy végtelen ciklust jelent, vagyis szemantikusan hibás forráskódot. A fordítóprogram az ilyen végtelen ciklust úgy szűri ki, hogy a fenti lépést csak egy adott küszöbbig hajtja végre, utána hibaüzenettel leáll.

7. Az előző lépések eredményeként teljesül, hogy egyik nem  $\varepsilon$ -t olvasó átmenet sem mutat olvasó utasításra vagy állapotra. Legyen  $T$  egy tetszőleges átmenet, ekkor tehát az alábbi lehetőségek lehetségesek:
  - 7.1. Ha  $T.R = \varepsilon$ , legyen  $T.W := \varepsilon$  és  $T.D := stand$
  - 7.2. Ha  $T.N = accept$ , legyen  $T.W := T.R$  és  $T.D := stand$
  - 7.3. Ha  $T.N = exit$ , töröljük  $T - t$
  - 7.4. Ha  $T.N = B$  egy író/léptető utasítás, akkor minden  $e \in B.\sigma$  elemre vegyünk fel egy új  $V$  átmenetet:  $V.S := T.S, \quad V.R := T.R, \quad V.N := e, \quad V.W := T.W, \quad V.D := B.D$  valamint ha  $T.W = NIL$ , akkor legyen  $V.W := e$ . (visszaírjuk amit olvastunk) majd töröljük  $T$ -t.
8. Indítsunk mélységi bejárást a gráf gyökeréből, és jelöljük meg azokat az író/léptető utasításokat, melyekre mutat író/léptető utasítás. Ezek olyan utasítások lesznek, melyeket nem tudtunk összevonni olvasásokkal (pl. a **left(a)**; **left(b)**; szekvencia második utasítása), ezért külön állapotokat kell számukra létrehozunk:
 

Legyen  $A$  egy megjelölt utasítás. Legyen  $s$  a T-gép egy új állapota. A szalagábécé minden  $l$  betűjére és minden  $e \in A.\sigma$  elemre vegyünk fel egy új  $T$  átmenetet:

$$T.S := s, \quad T.R := l, \quad T.N := e, \quad T.D := A.D$$

valamint ha  $T.W = NIL$ , akkor legyen  $T.W := e$ .
9. *exit*-re mutató átmenetek törlése:
 

$\sigma A$  konvenció szerint ha egy adott állapotban egy adott szimbólum olvasásához nem tartozik átmenet, akkor alapértelmezés szerint *exit*-tel terminál a T-gép. Ezért minden *exit*-re mutató átmenetet egyszerűen törölhetünk. Korábban az *exit*-re vonatkozó mutatókat nem törölhettük minden esetben, mert pl. ha az  $A.\sigma := \{B, exit\}$  halmazból töröljük,  $A.\sigma = \{B\}$ -t kapunk, aminek már más a jelentése mint az eredeti utasításnak.
10. Az előző lépés után keletkezhetnek olyan állapotok, melyeknek nem maradt egyetlen átmenete sem. Ezeket az állapotokat töröljük, valamint töröljük az összes rájuk mutató

átmenetet. Ezzel újabb átmenetek szűnnek meg, vagyis a lépést ismételnünk kell addig, amíg az összes állapotra elvégezve már nem történik változás.

### 3.1.4 Veremautomaták fordítása

A VA fordításának problémája meglehetősen hasonló a T-gépekéhez. Az egyetlen különbség, hogy most nem **read(a); write(b); left;** jellegű szekvenciákat kell összevonnunk, hanem **read(a); pop(b); push(c);** alakúakat (megj.: ezek nem Prolan utasítások, csak a jelentésüket tükrözi). Mivel itt is 3 típusú műveletről van szó, melyek közül megint csak az utolsó kettőt érdemes összevonni, a T-gép fordítására használt algoritmus kevés változtatással újra felhasználható. Ezek a különbségek:

- Az adatszerkezetek közt az író/léptető utasítás helyett most egy pop/push utasítás szerepel. Az attribútumai legyenek ugyanazok, így az algoritmus leírásán emiatt semmit sem kell változtatnunk. Az implementációban viszont biztosítani kell, hogy a változók mindkét lehetséges típusú adatot kezelni tudják (T-gép esetén D egy felsorolási típus ami a léptetés irányát tárolja, VA esetén egy szót tárol, ami gyakorlatilag egy lista).
- A 8. pontban ismételt író/léptető utasítások helyett most ismételt pop/push utasítások vannak. Ezúttal ezt érdemes úgy kezelni, hogy a hiányzó olvasó utasítást egy  $\varepsilon$  olvasásaként értelmezzük. Vagyis a 8. lépés vége így módosul:  
Legyen  $A$  egy megjelölt utasítás. Legyen  $s$  a VA egy új állapota. Minden  $e \in A.\sigma$  elemre vegyünk fel egy új  $T$  átmenetet:

$$T.S := s, \quad T.R := \varepsilon, \quad T.N := e, \quad T.D := A.D$$

(Megj.:  $T.W = NIL$  most nem fordulhat elő)

### 3.1.5 Futtatás

A 0 fejezetben tárgyalt fordítóprogram létrehozta a forráskódból a belső reprezentációt. Ebben a fejezetben a belső reprezentáció „lefuttatásával” létrehozzuk a levezetések ill. a generált nyelv néhány elemét. Mindkét feladat gyakorlatilag azt jelenti, hogy egy adott kezdőkonfigurációból kiindulva rekurzív módon meg kell határozni a levezethető konfigurációkat, amik így egy általában végtelen fa gráfot alkotnak. Tehát a teljes gráf meghatározása elvileg sem lehetséges, ezért különböző stratégiákat kell alkalmaznunk a fa lényeges jellemzőinek megjelenítésére:

**Lista:** A kezdő konfigurációból kiindulva generáljuk a közvetlenül levezethető konfigurációkat, viszont mindig csak az elsőként generált gyerekekre folytatjuk az algoritmust. Az első gyerek kiválasztása függ a fordítási direktíváktól ill. a beállításoktól (2.2 fej.). Ha a felhasználó a Lista nézetben megváltoztatja a levezetés egy lépését, akkor az újonnan kiválasztott gyerek konfigurációra hajtja végre a fenti algoritmust.

**Fa:** Létrehozuk a kezdő konfigurációt és a gyerekeit, de azok gyerekeit már nem hozzuk létre. Ha a felhasználó egy adott csomópontra kattint, a gyerekei létrejönnek, és megjelennek a fa gráfban. Vagyis a felhasználó maga tudja irányítani, hogy a gráf mely részeit építse tovább a program.

**Nyelvtanok által generált nyelv:** A kezdő konfigurációból kiindulva igyekszünk minél több terminális betűkből álló szót levezetni. A keresés alapvetően szélességi bejárással történik. Bizonyos esetekben, pl. a helyes zárójelezések nyelvénél a szélességi bejárás nagyon kevésbé hatékony, mivel egy véletlenszerűen választott utat vizsgálva, az csak kis valószínűséggel fog terminálni, hiszen a három szabályból kettő ( $S \rightarrow aSb$  és  $S \rightarrow SS$ ) növeli a nemterminális jelek számát, és csak egy ( $S \rightarrow \varepsilon$ ) csökkenti, így a mondatforma egyre távolabb kerül attól az állapottól, hogy csak terminális jeleket tartalmazzon. Ezért érdemes egy olyan heurisztikát beépíteni, amely előnyben részesíti azokat a konfigurációkat, melyekben kevés nemterminális jel van.

A heurisztika meghatározásakor nem csak az a cél, hogy minél több megoldást kapjunk, hanem hogy ezek a megoldások minél jobban reprezentálják a generált nyelv egészét. A helyes zárójelezések nyelve esetén, ha csak a fenti heurisztikát alkalmazzuk, a program gyorsan megtalálja a  $()$ ,  $(( ))$ ,  $(( ( )) )$ ,  $(( ( ( )) ) )$ , .... megoldásokat, mivel ezek mind olyan megoldások, melyek levezetésében mindig legfeljebb egy nemterminális szerepel. Viszont emiatt hamar elérve a megoldásszám-küszöböt, egyéb megoldásokat alig produkálna, ez pedig hamis képet adna a nyelvtan által generált nyelv jellegéről.

A fenti jelenség miatt előnyben kell részesítenünk azokat a megoldásokat, melyek a kezdőkonfigurációhoz közelebb vannak. Összefoglalva: az eredeti szélességi bejárásban szereplő sor adatszerkezetet egy elsőbbségi sorra kell cserélnünk, ahol az elem prioritását a fenti két tényező súlyozott összege adja. A tapasztalataim alapján az esetek többségében kielégítő eredményt kapunk, ha a nemterminálisok darabszáma és a gyökértől való távolság 1-1 súllyal szerepel az összegben (a program ezt a beállítást alkalmazza).

**Automaták által felismert nyelv:** T-gép és VA esetén a felismert nyelv meghatározása számításigényesebb, hiszen különböző bemenetekre kell végigjátszanunk azt, amit PNY esetén csak egyetlen kezdőállapotról kellett. A program először egy üres (csak blankot tartalmazó) szalagon futtatja az automatát, majd a bemenő jelek ábécéjéből sorban generálja az összes lehetséges szót egy adott hosszúságig. Minden bemenetre csak egy adott mennyiségű lépést tesz meg, mivel ha az adott bemenetre nem terminál az automata (vagy csak túl későn), akkor nem marad idő a többi bemenet vizsgálatára. Automaták esetén a program nem használ semmilyen heurisztikát, mivel egy T-gép működését megjósolni általában csak egy másik T-géppel volna lehetséges. Ez összefügg azzal, miért nem találunk általánosan megfelelő heurisztikát nulladrendű nyelvekhez sem.

## 3.2 Implementáció

### 3.2.1 Felhasznált technológiák

**Fejlesztői környezet:** Delphi 2007 for Win32.

**Delphi Yacc & Lex:** Lexikális és szintaktikus elemző-generátor. A Yacc és Lex szoftverek delphis változatai (3).

**TNT Unicode Controls:** Unicode karaktereket a Delphi csak a 12. verziótól kezdve támogat (Delphi 2009), ezért a régebbi verziók esetén külön komponenseket kell használnunk erre a célra. Egy erre szolgáló elterjedt és ingyenes megoldás a TNT (4).



**MinGW gcc:** GPL licenz alatt használható C fordító, amit a program mint előfordítót használ.  
(2).

**Egyéb:** Elsőbbségi sort (kupacot) megvalósító osztály: UHeap.pas (5).

### 3.2.2 Fájlok

- Prolan\ : Futtatható fájlok
  - Prolan.exe: A program
  - gcc.exe, cc1.exe: A gcc (C fordító) fájljai.
  - kézikönyv.pdf: Felhasználói kézikönyv.
- Prolan\lib\ : A könyvtárfájlok alapértelmezett helye. Itt találhatóak az automaták és a PNY kódsablonjai, valamint a matek.pla, ami matematikai PNY rutinokat tartalmaz.
- Prolan\samples\ : Példafájlok (ld. tesztelés, 3.4 fejezet).
- Prolan\src\ : Forrásfájlok.
  - Prolan.groupproj: Projektcsoport.
  - Prolan.dpr: Prolan projekt.
  - LanAniPack.dpk: Vizuális komponensek csomagja.
  - UBase.pas: Bázisosztályok.
  - UGrammar.pas: A programozott nyelvtan osztályai.
  - UTuring.pas: A T-gép osztályai.
  - UPushDown.pas: A veremautomata osztályai.
  - UMainForm.pas: Főablak.
  - UPageFrm.pas: Munkalapokat megjelenítő Frame.
  - UBasFrm.pas: Táblázatos munkalap.
  - UBasPLFrm.pas: Táblázatok a táblázatos munkalaphoz.
  - UAdvFrm.pas: Program munkalap.
  - UCodeFrm.pas: Forráskódot megjelenítő Frame.
  - UExeFrm.pas: Levezetésbongésző (Lista, Fa, Nyelv nézet).
  - UPIFrm.pas: A program munkalap középső része (definíció, automata átmenetei stb.)
  - UTntNiceGrid.pas: A táblázatos munkalaphoz készült vizuális komponens.
  - UWorkDlg.pas: A „Dolgozom...” üzenetet megjelenítő form.
  - UAniString.pas: Animációt megjelenítő vizuális komponens.
  - UAniFrm.pas: Az animáció frame-je.
  - UPrefDlg.pas: Beállítások dialógusablak.
  - UHeap.pas: THeap, elsőbbségi sort (kupacot) megvalósító osztály (5).
  - \*.dfm: az azonos nevű framet ill. formot leíró bináris állomány.
  - \*.res: Az egyes projektekhez tartozó bináris fájlok (pl. ikonokat fordít bele a fordító).
  - A többi fájlt a fejlesztői környezet generálja, pl. az ablakbeállítás és hasonló eltárolására.
- Prolan\src\pics\ : Ikonok.
- Prolan\src\dcu\ : lefordított unitok.
- Prolan\src\dyacclex-1.4\ : Delphi Yacc & Lex.
- Prolan\src\dyacclex-1.4\src\ : A Yacc & Lex forráskódja.

- Prolan\src\dyacclex-1.4\prolan\
  - prolanyacc.y: Yacc forrásfájl.
  - prolanlex.l: Lex forrásfájl. fájlok mappája.
  - prolanlex.pas: a Lex által generált kód.
  - prolanyacc.pas: a Yacc által generált unit.

Az ini és log fájl helye Win Xp rendszer esetén általában:

C:\Documents and Settings\User name\Local Settings\Application Data\ProLan

Vista és Win7 esetén C:\ProgramData\ProLan

- settings.ini: A program beállításai tartalmazó fájl.
- log.txt: A program futásáról tartalmaz információkat hibajavítás céljából.

### 3.2.3 Telepítés

A Prolan forráskódjának lefordításához két csomagot kell telepítenünk, a TNT Unicode Controls-t (a telepítéséről bővebben ld. itt (4)), valamint a LanAniPack.dpk csomagot. Ez utóbbi csomag a Prolan részeként készült, a táblázat munkalap ill. az animáció megjelenítését végző vizuális komponenseket tartalmazza (TTntNiceGrid és TLANAnimation).

### 3.2.4 Előfordítás

Az előfordítás feladata a kommentek eltávolítása a kódból, valamint a fordítási direktívák végrehajtása. Az előfordítás néhány funkciója könnyen megvalósítható lett volna közvetlenül is, viszont célszerűbbnek tűnt egy nyílt forráskódú C előfordítót felhasználni erre a célra: MinGW-GCC (2). Ezen megoldás előnyei: teljes szolgáltatás, jól dokumentált, szabványos, elterjedt, megbízható.

Az előfordító általában egybe van építve magával a fordítóprogrammal, vagyis tkp. a komplett fordítóprogram egy funkciójaként használható. A gcc esetén ennek alakja a következő:

```
gcc -E file1 -o file2 -I mappa
```

#### Magyarázat

- -E: A gcc-t előfordítóként fog működni, vagyis nem tárgykóddal vagy objektumkóddal tér vissza, hanem a forráskód előfordított alakjával.
- file1: Forráskód. A gcc a fájl kiterjesztéséből határozza meg a formátumát, ezért a kiterjesztést érdemes .c-nek választani, hogy tudja, forráskódról van szó.
- -o file2: Kimenet.
- -I mappa: A megadott mappa tartalmára hivatkozhatunk a kódban az `include` direktívával.

A gcc.exe a Prolan\bin\ mappában található. Az előfordítást a TSPProgram alábbi metódusa végzi:

```
class function TSPProgram.PreProc(List: TStringList): TProgType;
```

Működése: a **List**-ben lévő szöveget forráskódként értelmezi, amit átad a gcc-nek, majd visszatérési értéként megadja a forráskód típusát (**grammar**, **turing**, **pushdown**, **epushdown**).

### 3.2.5 Lexikális, szintaktikus elemző

A lexikális elemző feladata a forráskódot lexikai elemekre bontani (mint pl. változónév, szám, kulcsszó stb.). A szintaktikus elemző a lexikális elemekből felépíti a szintaxisfát, mely a forráskód szintaktikai felépítését tükrözi. A ProLan ezt a két feladatot a nyílt forráskódú Delphi Yacc & Lex (3) segítségével oldja meg, ami az elterjedt Yacc és Lex szoftverek delphis változatai.

A Lex (lexikális elemző generátor) bemenete egy Lex nyelven megírt kód: `prolan.l`, amiből Object Pascal kódot generál: `prolanlex.pas`. A forráskód felépítése egyszerű: tartalmazza a különböző kulcsszavakat (pl. **true**, **if**, **return**), operátorokat (pl. **&&**, **||**, **+**) valamint reguláris kifejezéseket az egyéb lexikai elemek leírására: változónevek, számok, az idézőjellel lezárt stringek.

A Yacc (Yet Another Compiler-Compiler) működése hasonló a Lex-éhez: bemenete egy Yacc nyelven megírt kód: `prolan.y`, amiből Object Pascal kódot generál: `prolan yacc.pas`. Ez utóbbi fájl egy komplett pascal unit, ami include-olja a `prolanlex.pas`-t, és amit a ProLan forráskódja már közvetlenül felhasznál.

A fent említett fájlok mind a `Prolan\src\dyacclex-1.4\prolan` könyvtárban találhatóak.

A Delphi Yacc & Lex használata: parancssorból hívjuk meg az alábbi parancsot:

```
src\dyacclex-1.4\src\dyacclex-1.4\make prolan
```

## 3.3 Osztályhierarchia

Osztályok többségét külön fejezetben tárgyalom. Megadom az osztály feladatát, új mezőit, jellemzőit és metódusait, vagyis azokat melyek a bázisosztályban még nem szerepeltek. Emiatt előfordulnak olyan osztályok, melyekhez a feladatukon kívül semmilyen megjegyzés nem tartozik. A tömörség kedvéért ezeket az osztályokat nem külön fejezetben, csak a bázisosztály végén egy „leszármazottak” pontban adom meg. Ez a pont nem tartalmazza az összes leszármazottat, csak a fent említetteket.

Az alábbiakban megadom néhány Delphi-specifikus fogalom magyarázatát, melyeket osztályhierarchia leírásában felhasználók.

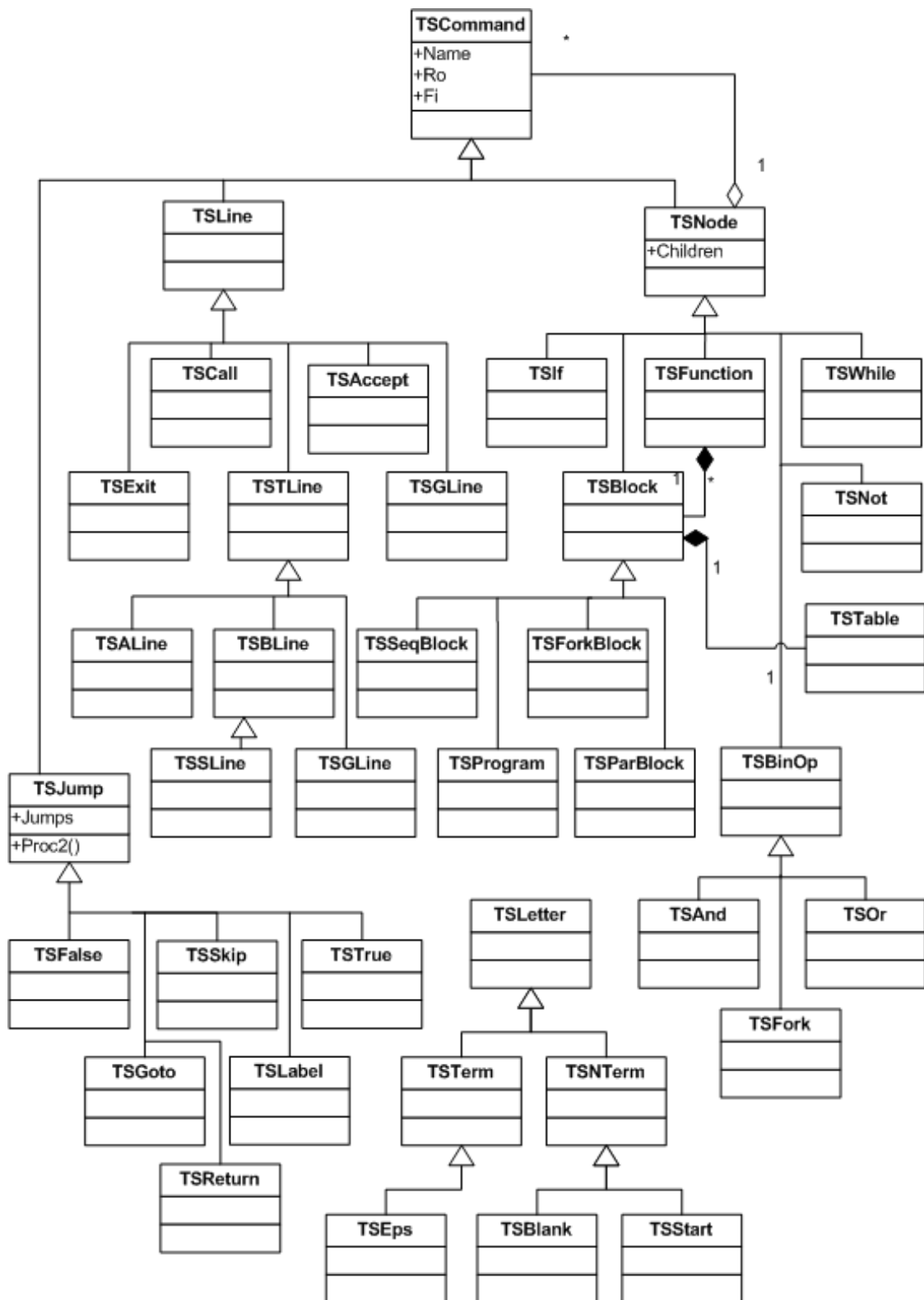
**Jellemzők:** Delphi-ben egy osztálynak mezőkön és metódusokon kívül jellemzői (properties) is lehetnek. Minden jellemzőhöz tartozhat egy **read** és egy **write** metódus, ami a jellemző olvasásakor ill. írásakor meghívódik, ezek általában egy mező értékét olvassák, módosítják, miközben ellenőrzéseket is végeznek. A jellemzők jellemzően publikusak, a hozzájuk tartozó metódusok ill. mezők pedig rejtettek. A továbbiakban a tömörség kedvéért elsősorban csak a jellemzőket sorolom fel, a hozzájuk tartozó mezők és metódusok értelemszerűek.

**Unitok:** A Delphi fordítási és könyvtári egysége. Minden unit külön névteret alkot, lehetnek publikus (más unitok által is látható) ill. rejtett eljárásai, változói. Uniton belül nincs adatretetés, vagyis azonos unitban lévő osztályok egymás „friend”-jének számítanak. Minden unit nevét nagy U-val kezdődőnek írom.

**Típusok:** A konvenció szerint Delphiben minden típust nagy T-vel kezdünk, az ezt követő néhány betű szintén nagy, és utal a típus unitjára, ill. az osztályhierarchiára amihez tartozik.

### 3.3.1 A szintaxisfa osztályai

A szintaxisfát felépítő osztályok az UBase unitban találhatók. A típusok nevei TS-sel kezdődnek (S mint syntactic).



11. ábra: a szintaxisfa osztályai

### 3.3.1.1 TSCOMMAND

**Bázisosztály:** TObject

**Unit:** UBase

**Feladata:** A szintaxisfa elemeinek őssztálya.

**Jellemzők:**

- **Name:** String;
- **Sig, Fi:** TList; A  $\sigma, \varphi$  halmazokat reprezentáló listák.
- **Lab:** TSLabel; Az utasításhoz (opcionálisan) tartozó címke.

**Eljárások:**

- **procedure Proc1;**  
A fordítási folyamat első lépése:  $\sigma, \varphi$  halmazok beállítása mélységi bejárással.
- **procedure Proc3;**  
A fordítási folyamat utolsó lépése:  $\sigma, \varphi$  végleges beállítása.

### 3.3.1.2 TSLINE

**Bázisosztály:** TSCOMMAND

**Unit:** UBase

**Feladata:** A szintaxisfa egy levelét reprezentálja (pl. nyelvtani szabály, T-gép író/olvasó utasításai stb.)

**Jellemzők:**

- **LineNo:** Integer; Az utasítás sorának száma a forrásfájlban.
- **Instance:** TEItem; Az utasítás futtatható példánya (a Build eljárás legutóbbi eredménye).

**Eljárások:**

- **procedure Build(P: TEPProgram);**  
Létrehoz önmagából egy futtatható utasítást, és hozzáadja a P programhoz.

### 3.3.1.3 TSCALL

**Bázisosztály:** TSLINE

**Unit:** UBase

**Feladata:** Függvényhívás leíró.

**Jellemzők:**

- **Function:** TSFunction; A meghívandó függvény neve.
- **Params:** TList; Paraméterek listája.

**Leszármazottai:**

- **TSExit:** **exit** utasítás
- **TSAccept:** **accept** utasítás

### 3.3.1.4 TSNode

**Bázisosztály:** TSCOMMAND

**Unit:** UBase

**Feladata:** A szintaxisfa csomópontjait reprezentálja.

**Jellemzők:**

- Children: TList; A gyerekek listája.

**Leszármazottai:**

- TSIf: **if** utasítás.
- TSWhile: **while** utasítás.
- TSNot: **!** operátor.

**3.3.1.5 TSFunction**

**Bázisosztály:** TNode

**Unit:** UBase

**Feladata:** Függvénydefiníció leírása.

**Jellemzők:**

- Block: TSBLOCK: A függvénydefiníció törzse.
- Table: TSTable: A törzs szimbólumtáblája, kiegészítve a formális paraméterekkel.
- OutTrue: TSReturn: Segéd címke az igaz értékkel való visszatéréshez. (ld. *return\_true*)
- OutFalse: TSReturn: Segéd címke a hamis értékkel való visszatéréshez. (ld. *return\_false*)
- FuncType: TSFuncType: A visszatérési érték típusa.
- Params: TList: Formális paraméterek listája.

**Metódusok:**

- **procedure** CheckPara(L: TList)  
Ellenőrzi, hogy az L lista illeszthető-e a formális paraméterekre (Params)

**3.3.1.6 TSBLOCK**

**Bázisosztály:** TNode

**Unit:** UBase

**Feladata:** Blokkutasítások őse.

**Jellemzők:**

- Table: TSTable: A blokk szimbólumtáblája.

**Leszármazottai:**

- TSSeqBlock: szekvenciális blokk: { }
- TSParBlock: mátrix blokk: [ ]
- TSForkBlock: elágazó blokk: < >

**3.3.1.7 TSProgram**

**Bázisosztály:** TSBLOCK

**Unit:** UBase

**Feladata:** A szintaxisfa gyökere.

**Jellemzők:**

- Input: TESentence. A program bemenete (pl. T-gépnél a szalag kezdőállapota).
- ErrorLine: Integer. Fordítási hiba esetén a hiba sora a forráskódban
- Main: TSFunction. A főprogram (void main())

- **ProgType**: TProgType. A program típusa (programozott nyelvtan, Turing gép vagy veremautomata)

**Metódusok:**

- **procedure** CheckPara(L: TList)  
Ellenőrzi, hogy az L lista illeszthető-e a formális paraméterekre (Params)
- **class function** PreProc(L: TStringList): TProgType  
Az L-et mint forráskódot előfeldolgozza. Az eredményt visszatéríti L-be.  
A visszatérési érték a program típusa.
- **class function** Compile(L: TStringList): TSPProgram  
L-et mint forráskódot lefordítja. Először meghívja rá az előfordítót, majd a lexikális és szintaktikus elemzőt, ellenőrzi hogy nem történt hiba, végül elvégzi a PA-ra fordítást.  
Visszatér az elkészült PA-val.
- **function** Build(EP: TEPProgram): TEPProgram  
Felépíti önmaga futtatható példányát EP-be.

### 3.3.1.8 TSJump

**Bázisosztály:** TSCCommand

**Unit:** UBase

**Feladata:** Ugró utasítások őssosztálya.

**Jellemzők:**

- **Jumps**: TList. Céluutasítások halmaza. (mivel nemdeterminisztikus az ugrás, ez egy halmaz, amit listával reprezentálunk)

**Leszármazottak:**

- **TSGoto**: goto utasítás, TSLabel-ek halmazára mutathat.
- **TSLabel**: Virtuális utasítás a címkék kezelése érdekében. Arra az utasításra mutat, amihez a címke tartozik.
- **TSTrue**: A true konstanst (utasítást) reprezentálja. A true végrehajtása ui. nem más, mint hogy feltétel nélkül ugrik a  $\sigma$  halmaz egy elemére.
- **TSFalse**: A false konstanst reprezentálja.
- **TSReturn**: A függvényből való visszatéréshez használt címke. A függvény meghívásának helyére ugrik vissza.

### 3.3.1.9 TSBinOp

**Bázisosztály:** TNode

**Unit:** UBase

**Feladata:** Bináris operátorok őssosztálya.

**Leszármazottak:**

- **TSAnd**: Lusta és
- **TSOr**: Lusta vagy
- **TSFork**: Elágazó vagy

### 3.3.1.10 TSLetter

**Bázisosztály:** TObject

**Unit:** UBase

**Feladata:** Szimbólumok őszosztálya.

**Leszármazottak:**

- TSTerm: Terminális jelek
- TSNterm: Nemterminális jelek
- TSB1ank: „blank” jel T-gépeknél
- TSEps: Görög epszilon, az üres szó jele
- TSStart: Startszimbólum (PNY), veremtető szimbólum (VA).

### 3.3.2 Programozott automata

Az alábbi osztályok reprezentálják a programozott automatát, vagyis azt a belső reprezentációs szintet, amelyet már nem kell tovább elemezni, hanem közvetlenül „futtatható”. Ezeknek a típusoknak a neve TE-vel kezdődik (E mint executable). A szintaxisfa néhány elemének közvetlenül megadható a párja ebben a hierarchiában. A fordítás során pl. egy TSTerm osztályú objektumnak (ami egy terminális jelet reprezentál a szintaxisfában) egy TETerm osztályú objektum felel meg. Más esetekben, pl. **if**, **while** és egyéb programszerkezeteknél a cél éppen az, hogy a sokféle szintaktikus elemet csak kevés típusú (PNY esetén egyetlen: TERule) futtatható utasítással helyettesítsük. Ezt az átírási folyamatot ismertettem a 0 fejezetben.

#### 3.3.2.1 TEItem

**Bázisosztály:** TObject      **Unit:** UBase

**Feladata:** Futtatható elemek őszosztálya.

**Jellemzők:**

- Name: WideString

#### 3.3.2.2 TELetter

**Bázisosztály:** TEItem      **Unit:** UBase

**Feladata:** Nyelvtani jelek őszosztálya.

**Leszármazottak:**

- TETerm: terminális jelek
- TENTerm: nemterminális jelek
- TEB1ank: blank jel (automaták esetén)

#### 3.3.2.3 TESentence

**Bázisosztály:** TObject      **Unit:** UBase

**Feladata:** Mondatformát reprezentál.

**Jellemzők:**

- Letters[index: integer]: TELetter; A mondatforma betűi
- First: Integer; A kijelölt részmondat első elemének indexe (kijelölt részmondat pl. a List nézetben a pirossal jelölt rész)
- Last: Integer; A kijelölt részmondat utolsó elemének indexe



### **Metódusok:**

- **procedure** Assign(Source: TESentence); **overload**; Leklónozza Source-t.
- **procedure** Assign(Source: TList); **overload**; Leklónozza Source-t.
- **function** Clone: TESentence; Leklónozza önmagát.
- **procedure** AssignFromInt(N: Integer; L: TList);  
Az N egészt egy n jegyű számként ábrázolja, ahol a számjegyek az L elemei. Az így kapott számot mondatformaként értelmezi.
- **function** Equal(S: TESentence): Boolean; Igaz, ha egyenlő S-sel.
- **procedure** Push(S: TESentence); Az elejére másolja S tartalmát.
- **procedure** Pop; Kitörli az első elemet.
- **function** TopItem: TLetter; Visszaadja az első elemet.
- **procedure** Write(I: Integer; L: TLetter); L-et az I-edik helyre írja.
- **function** IsEps: Boolean; True, ha a mondatforma üres.
- **function** Len(UseSpace: Boolean): Integer;  
A mondatforma hossza karakterekben. UseSpace: használjon-e szóközt a betűk között.
- **procedure** Draw(C: TCanvas; Left, Top: Integer);  
Kirájzolja a C canvasra, Left, Top koordinátákkal.
- **function** ToString(UseSpace: Boolean; First: Integer; Last: Integer): WideString; **overload**;  
String-be konvertálja, a First indexű karaktertől kezdve a Last-ig.
- **function** ToString(UseSpace: Boolean = true): WideString; **overload**;  
String-be konvertálja a teljes tartalmat.
- **function** ToTapeString: WideString;  
String-be konvertálja a teljes tartalmat (T-gép szalagos változat)
- **function** CharPos(LetterPos: Integer; UseSpace: Boolean = true): Integer;  
Visszaadja a LetterPos indexű betű pozícióját a stringben.
- **function** ToAnsiString: String; Ansi String-be konvertálja.
- **function** Count: integer; A betűk száma.
- **procedure** Add(L: TLetter); **overload**; Hozzáfűz egy betűt a végére.
- **procedure** Add(S: TESentence); **overload**; Hozzáfűzi S tartalmát.
- **function** Items: TList; Az elemek mint lista.
- **function** Find(K: Integer; L: TESentence): integer; **overload**;  
A K indextől kezdve keresi az L részmondatot. A visszatérési érték a találat pozíciója.
- **function** Find(L: TESentence; M: TFindMode; K: Integer = -1): integer; **overload**; ld. fent. Különbség: M határozza meg a keresés módját: első, utolsó vagy véletlenszerű találat.
- **function** FindAll(L: TESentence; var T: intarray): integer;  
L minden előfordulását megkeresi, és ezeket betölti a T tömbbe.
- **procedure** Insert(K, M: Integer; S: TESentence);  
A K-M intervallum helyére bemásolja S tartalmát.
- **function** IsTerminal: boolean;  
Igaz, ha a mondatforma csak terminális jeleket tartalmaz, vagy üres.

- **constructor** Create(T: array of TLetter); **overload**;  
Konstruktor, ami egy T tömbből feltölti a mondatformát.
- **constructor** Create(L: TList); **overload**;  
Konstruktor, ami egy L listából feltölti a mondatformát.

#### 3.3.2.4 TEBasLine

**Bázisosztály:** TEItem

**Unit:** UBase

**Feladata:** A végrehajtható utasítások őssztálya. Az utasítások fa gráfot alkotnak. Minden makróhívás egy csomópont, aminek gyerekei a makró utasításai (amik szintén lehetnek makróhívások).

**Mezők:**

- FSLine: TSLine: Az utasításhoz tartozó elem a szintaxisfában.

**Metódusok:**

- **function** ToString(UseSpace: Boolean = **false**): WideString;  
**virtual**;  
Stringbe konvertál.
- **function** ToAnsiString: string; **virtual**; Ansi Stringbe konvertál.
- **function** IsLetter: Boolean; **virtual**; Igaz, ha levél (vagyis nem makróhívás).

#### 3.3.2.5 TELine

**Bázisosztály:** TEItem

**Unit:** UBase

**Feladata:** A redukált programozott automata egy utasítása.

**Mezők:**

- FState: TSLabState; Színezés a gráfbejáráshoz.

**Jellemzők:**

- Sig: TList;  $\sigma$  halmaz listával ábrázolva.
- Fi: TList;  $\varphi$  halmaz listával ábrázolva.

**Metódusok:**

- **procedure** ResolveA; **virtual**;
- **procedure** Resolve; Gráfbejárások a makróhívások feloldásához.
- **class procedure** ResolveList(L: TList); Segéd eljárás a gráfbejáráshoz.

**Leszármazottak:**

- TExit: virtuális utasítás. (az erre az utasításra való ugrás terminálást jelent)
- TAccept: virtuális utasítás. (az erre az utasításra való ugrás elfogadó állapotba való ugrást jelent)

#### 3.3.2.6 TEOall

**Bázisosztály:** TELine

**Unit:** UBase

**Feladata:** Makróhívás.

**Mezők:**

- FLines: TList; A makró sorai.
- FPara: TList; Paraméterlista.
- FFunct: TSFunction; A makródefiníció osztálya.
- FStarts: TList; A makró kezdő utasításainak halmaza. (a nemdeterminisztikus végrehajtás miatt ez nem egy konkrét utasítás, hanem egy halmaz)

**Jellemzők:**

- Sig: TList.  $\sigma$  halmaz listával ábrázolva.
- Fi: TList.  $\varphi$  halmaz listával ábrázolva.

**Metódusok:**

- **procedure** Expand(EP: TEPProgram);  
Makróhívás feloldása (az utasítások példányosítása)

**3.3.2.7 TEstart****Bázisosztály:** TELine**Unit:** UBase

**Feladata:** Futtatható program gyökere. Egyfajta fejelem az egységesebb kezelhetőség érdekében.

**Jellemzők:**

- EProgram: TEPProgram. A programra mutató mutató.

**3.3.2.8 TEPProgram****Bázisosztály:** TObject**Unit:** UBase

**Feladata:** Futtatható program.

**Mezők:**

- FStart: TEnterm; Startszimbólum.
- FInput: TESentence; Bemenet (automaták esetén).
- FFfirst: TEstart; A kezdő konfiguráció címkéje
- FStarts: TList; A kezdő címkék halmaza
- FMain: TCall; A main makró.
- FLines: TList; Sorok.
- FStartTree, FStartList, FStartLang: TConfig;  
A különböző levezetések kezdő konfigurációi.
- FNames: TStringList; A programban használt szimbólum, címke, állapot stb. nevek.
- FNewLineCount: Integer; Segédváltozó a PNY sorainak elnevezéséhez.
- FLog: TTntListView; A hibaüzenetek vizuális komponense.
- FRunFrm: TObject; Az aktív levezetésbongészó frame.
- FExit: TExit; A program **exit** utasítása.
- FEps: TLetter; A program  $\epsilon$  szimbóluma.
- FSPProgram: TSPProgram; A szintaxisfa.

### ***Jellemzők:***

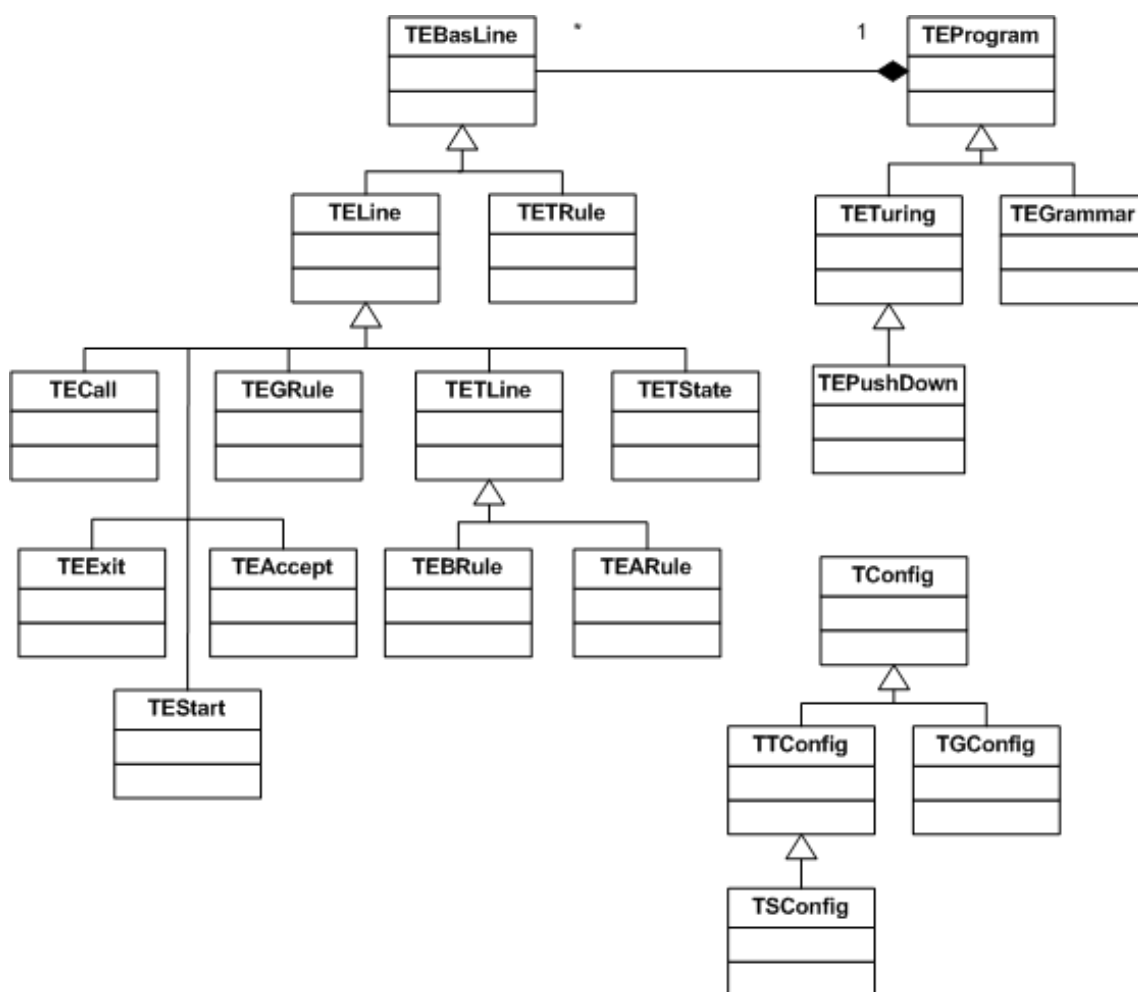
- `Lines[index: integer]: TEBasLine`. Az utasítások.
- `TreeView: TTntTreeView`; A Fa nézet vizuális komponense.
- `ListView: TTntListView`; A Lista nézet vizuális komponense.

### ***Metódusok:***

- **function** `IsEmpty: Boolean; virtual;`  
Igazsággal tér vissza, ha a programnak nincsenek utasításai.
- **function** `AddName(I: TEItem; S: string = ''): string;`  
Hozzáad egy nevet.
- **procedure** `Build(C: TCall); virtual;`  
Felépíti a programot, ahol C a főprogramot meghívó makróhívás.
- **procedure** `Select (V: TViewType);`  
Vált a különböző nézetek között.
- **function** `CanRead(L: TELetter): Boolean; virtual;`  
Igaz értékkel tér vissza, ha az L szimbólumra értelmezve van az olvasó művelet.
- **function** `CanStep(L: TELetter): Boolean; virtual;`  
Igaz értékkel tér vissza, ha az L szimbólumra nem áll meg a left és a right utasítás.
- **procedure** `GetTerms(L: TStringList);`  
L-be tölti a program terminális szimbólumait.
- **procedure** `GetNterms(L: TStringList);`  
L-be tölti a program nemterminális szimbólumait.
- **function** `ToStringList: TStringList;`  
StringList-be konvertálja a programot.
- **function** `AddRule(L: TEBasLine; S: String = ''): TEBasLine; overload;`  
Hozzáad egy utasítást S névvel.
- **function** `AddRule3(L: TEBasLine): TEBasLine; overload;`  
Hozzáad egy utasítást.
- **function** `AddCall(L: TEBasLine): TEBasLine; overload;`  
Hozzáad egy makróhívást.
- **function** `Count: Integer;` Az utasítások száma.
- **function** `SProgram: TSPProgram;` A szintaxisfa.
- **procedure** `RunTree(D: Integer); virtual; abstract;`
- **procedure** `RunList(D: Integer); virtual; abstract;`
- **procedure** `RunLang; virtual; abstract;`  
Lefuttatja a programot, vagyis az adott nézethez tartozó kezdőkonfigurációból kiindulva megtesz D lépést.
- **procedure** `ToListView(LV: TTntListView); virtual; abstract;`  
Kitölti a program állapotait, átmeneteit, a nyelvtan szabályait megjelenítő LV táblázatot.
- **procedure** `GetDefinition(L: TTntStrings); virtual; abstract;`  
L-be tölti a program matematikailag precíz definícióját.

### 3.3.3 Konfigurációk

Az alábbi osztályok konfigurációkat ábrázolnak ill. ezekkel kapcsolatos műveleteket valósítanak meg.



12. ábra Futtatható programok és elemei, konfigurációk

#### 3.3.3.1 TCMatrix

**Bázisosztály:** TObject

**Unit:** UBase

**Feladata:** Egy döntési pont leírója a lista nézetben.

**Mezők:**

- Rows: TList; Az adott lépésben választható szabályok/állapotátmenetek listája.
- Cols: TIntList; A szabály illesztésének pozíciója (a szabály bal oldalát melyik indexre illesztjük a mondatformában). Ennek a jellemzőnek csak PNY esetén van szerepe.
- Items: TObjectMatrix; Az összes lehetséges szabály-pozíció kombinációhoz tartozó konfigurációk kétdimenziós tömbje.

#### 3.3.3.2 TConfig

**Bázisosztály:** TObject

**Unit:** UBase

**Feladata:** Konfigurációkat reprezentáló osztályok őssztálya.

**Mezők:**

- FRoot: TConfig; A kezdőkonfiguráció
- FParent: TConfig; Szülő konfiguráció

**Jellemzők:**

- Runed: Boolean; True, ha a gyerekeit már meghatároztuk.
- Node: TObject; A konf.-hoz tartozó Lista/Fa grafikus elem.
- Items: TList; Gyerekek listája.

**Metódusok:**

- **function** GetLangString: WideString; **virtual**;  
A nyelv nézetben megjelenő stringet ad vissza.
- **function** GetTreeString: WideString; **virtual**;  
A fa nézetben megjelenő stringet ad vissza.
- **function** MakeTreeNode(TV: TObject = nil): TObject; **virtual**;  
Létrehoz egy elemet a fa nézetben (TTreeNode)
- **function** MakeListNode(TV: TObject = nil): TObject; **virtual**;  
Létrehoz egy elemet a lista nézetben (TListItem)
- **function** MakeLangNode(TV: TObject = nil): TObject; **virtual**;  
Létrehoz egy elemet a nyelv nézetben (TListItem)
- **procedure** DrawTreeNode(C: TCanvas; Left, Top: Integer); **virtual**;  
Fa elemet rajzol az adott canvas adott koordinátájára.
- **function** IsTerminal: Boolean; **virtual**; True, ha a konfiguráció terminál.
- **function** GetMatrix: TCMatrix; **virtual**; **abstract**;  
Létrehozza a konfiguráció gyerekeit leíró mátrixot (ld. TCMatrix)
- **function** GetProgram: TProgram;  
Visszaadja a futtatható programot melyhez a konfiguráció tartozik.
- **function** GetListView: TTntListView; Visszaadja az aktuális Lista nézetet.
- **function** GetTreeView: TTntTreeView; Visszaadja az aktuális Fa nézetet.
- **procedure** DetachAndFree; Lecsolja a szülőről és felszabadítja.
- **function** Line: TEBasLine; **virtual**;  
A konfigurációhoz tartozó szabály/állapotátmenet.
- **function** Add(C: TConfig): TConfig; **virtual**; Hozzáad egy gyereket.
- **function** Parent: TConfig; Visszatér a szülővel.
- **function** ToString(UseSpace: Boolean = false): WideString;  
**virtual**; **abstract**; stringbe konvertál.
- **function** StateToString(UseSpace: Boolean = false): WideString;  
**virtual**; **abstract**; A Lista nézetben a mondatformánál megjelenő stringet adja vissza.
- **procedure** MakeChildren; **virtual**; **abstract**;  
Létrehozza a gyerek konfigurációkat.
- **procedure** RunTree(D: Integer); **virtual**;  
Rekurzív módon, D mélységben építi a fát, a Fa nézet beállításával.

- **procedure** RunList(D: Integer); **virtual**;  
Rekurzív módon, D mélységben építi a fát, a Lista nézet beállításával.
- **procedure** RunLang(D: Integer); **virtual**;  
Rekurzív módon, D mélységben építi a fát, a Nyelv nézet beállításával.
- **function** MakeNode(TV: TObject = nil): TObject; **virtual**;  
Segéd függvény a többi MakeNode számára.
- **constructor** Create(P: TConfig; L: TEBasLine);  
Konstruktor, ahol megadjuk a szülő konfigurációt, és az FLine értékét.

### 3.3.4 Programozott nyelvtan

A programozott nyelvtant modellező osztályok az UGrammar unitban találhatóak. Ez a unit az UBase egy bővítésének tekinthető: új leszármazott osztályok, melyek a szintaxisfát ill. a futtatható programot bővítik/módosítják, hogy képes legyen PNY-okat futtatni. Ezek az osztályok általában felülírják a bázisosztály néhány metódusát, és újakat nem tartalmaznak. A felülírásokat külön nem tüntetem fel.

#### 3.3.4.1 TSGRule

**Bázisosztály:** TSLine      **Unit:** UGrammar

**Feladata:** Nyelvtani szabályt reprezentál a szintaxisfában.

**Mezők:**

- FLeft: TList; A szabály baloldala.
- FRight: TList; A szabály jobb oldala.

**Metódusok:**

- **class function** MakePrint: TSGRule;  
Létrehoz egy print utasítást a szintaxisfa számára.

#### 3.3.4.2 TEGrammar

**Bázisosztály:** TEProgram      **Unit:** UGrammar

**Feladata:** „Futtatható” programozott nyelvtan.

#### 3.3.4.3 TEGRule

**Bázisosztály:** TELine      **Unit:** UGrammar

**Feladata:** „Futtatható” nyelvtani szabályt reprezentál.

**Jellemzők:**

- Left: TESentence; A szabály bal oldala.
- Right: TESentence; A szabály jobb oldala.

**Metódusok:**

- **class function** MakePrint: TEGRule;  
Létrehoz egy print utasítást a szintaxisfa számára.

#### 3.3.4.4 TGConfig

**Bázisosztály:** TConfig      **Unit:** UGrammar

**Feladata:** „Futtatható” nyelvtani szabályt reprezentál.

**Jellemzők:**

- Succeeded: Boolean; True, ha a szabály illeszthető volt a mondatformára.
- LetterInd: Integer; Az illesztés pozíciója.
- Sentence: TESentence; Aktuális mondatforma.

**Metódusok:**

- **function Rule:** TEGRule; Az alkalmazott nyelvtani szabály.

**3.3.4.5 TGThread**

**Bázisosztály:** TThread      **Unit:** UGrammar

**Feladata:** Nyelvtan által generált nyelv kiszámítását végző thread.

**3.3.5 Turing-gép és veremautomata**

A Turing-gépet és veremautomatát szimuláló osztályok a UTuring valamint UPushDown unitban találhatóak. A két automata fordítása nagyban megegyezik, a VA a T-gép leszármazottjaként van megvalósítva. A VA új elemeket már nem is tartalmaz, csak módosítja a T-gép elemeit, amik szintén jórészt csak módosítják a bázisosztályt. Az alábbiakban is csak az új mezőket, jellemzőket és metódusokat adom meg, az öröklött elemek módosításait nem ismételtem.

**3.3.5.1 Típusok**

- TDirection: T-gép esetén a lépés iránya, VA esetén a verembe írandó elem típusa.
- TColor: Színek a gráfbejáráshoz.

**3.3.5.2 TSTLine**

**Bázisosztály:** TSLine      **Unit:** UTuring

**Feladata:** A szintaxisfához tartozó T-gép utasításainak őse.

**Mezők:**

- FLetter: TSLetter; Az utasítás paramétere (amit ír, olvas stb.).

**Metódusok:**

- **class function MakePrint:** TSTLine;  
Létrehoz egy print utasítást a szintaxisfa számára.

**Leszármazottai:**

- TSALine: T-gép/VA olvasó utasítás a szintaxisfában.
- TSBLine: T-gép író/léptető utasítás a szintaxisfában. Unit: UPushDown
  - Mező: FDirection: TDirection; A léptetés iránya / a verembe művelet paramétere.
  - Leszármazottja: TSSLLine: VA veremki/verembe műveletet leíró osztály a szintaxisfában.

**3.3.5.3 TETLine**

**Bázisosztály:** TELine      **Unit:** UTuring

**Feladata:** Ideiglenes Turing-gép utasítások, amiket majd összevonunk T-gép állapotokká és átmenetekké



**Mezők:**

- **FRefs**: TList; Az utasításra mutató utasítások listája.
- **FLetter**: TLetter; Az utasítás paramétere.
- **FMark**: Boolean; „Megjelölés” különböző algoritmusok számára.
- **FColor**: TColor; Szín a gráfbejárások számára.
- **FTState**: TETState; Turing-gép állapot, melyhez az utasítás tartozik.

**Metódusok:**

- **procedure** AddRef(L: TETLine); Hozzáad egy elemet a referencialistához.
- **procedure** RemRef(L: TETLine); Eltávolít egy elemet a referencialistából
- **procedure** MakeRefs; Hozzáadja önmagát azokhoz az utasításoknak a referencialistáihoz, melyekre mutat.
- **procedure** ClearRefs; Eltávolítja önmagát azoknak az utasításoknak a referencialistáiból, melyekre mutat.

**Leszármazottai:**

- **TEARule**: Ideiglenes olvasó utasítás.
- **TEBRule**: Ideiglenes író/léptető (verembe/veremki) utasítás.

**3.3.5.4 TETRule**

**Bázisosztály:** TEBasLine      **Unit:** UTuring

**Feladata:** A végleges Turing-gép / VA egy állapotátmenetét írja le.

**Jellemzők:**

- **Left**: Az olvasott szimbólum.
- **Right**: Az írandó / veremki szimbólum.
- **Sig**: A következő állapot.
- **Dir**: A léptetés iránya / a verembe szimbólum.

**3.3.5.5 IRuleIterator**

**Bázisosztály:** IInterface      **Unit:** UTuring

**Feladata:** Egy állapot átmenetein lépkedő iterátor. Csak azokat az átmeneteket veszi figyelembe, melyek a Letter jellemzőben megadott szimbólumot olvassák. Vagyis: egy adott állapotban egy adott szimbólumot olvasunk, akkor ezzel az iterátorral nézhetjük végig a releváns átmeneteket.

**Jellemzők:**

- **property Letter**: TLetter; A szimbólum, amit olvasó átmeneteket keressük.

**Metódusok:**

- **function** GetCurrent: TETRule; **Letter**-ből beállítja az első aktuális átmenetet.
- **function** GetLetter: TLetter; Visszatér a **Letter** jellemző értékével.
- **procedure** SetLetter(const Value: TLetter); Beállítja a **Letter**-t.
- **procedure** RemoveCurrent; Eltávolítja az állapotból az aktuális átmenetet.

- **procedure** Reset; Alapállapotba állítja az iterátort.
- **function** Current: TETRule; Az aktuális átmenet.
- **function** Next: Boolean; Lépés a következő átmenetre.
- **function** Prev: Boolean; Lépés az előző átmenetre.

### 3.3.5.6 TRuleIterator

**Bázisosztály:** TInterfacedObject, IRuleIterator      **Unit:** UTuring

**Feladata:** Az IRuleIterator megvalósítása.

**Mezők:**

- FState: TETState; A T-gép / VA állapot, melynek átmenetei között keresünk.

### 3.3.5.7 TETState

**Bázisosztály:** TELine      **Unit:** UTuring

**Feladata:** A Turing-gép / VA egy állapota.

**Mezők:**

- Rules: TArrayList; Az állapothoz tartozó átmenetek. A tömb a szalagábécé betűivel van indexelve (pontosabban azok indexeivel), a tömb minden eleme egy lista, aminek elemei az adott betűt olvasó átmenetek.

**Jellemzők:**

- **property** Progi: TETuring; A futtatható Turing-gép / VA amihez az állapot tartozik.

**Metódusok:**

- **function** GetIterator: IRuleIterator; Visszaad egy átmenet-iterátort.
- **procedure** SetTermCount(N: Integer); N-re állítja be a szalagábécé elemszámát.
- **procedure** RemoveRule(R: TETRule); Eltávolít egy R átmenetet.
- **function** AddRule(Left: TELetter; Right: TELetter; Dir: TDirection; Sig: TObject; FL: TSLine): TETRule; Hozzáad egy átmenetet, ahol Left az olvasott betű, Right az írandó (veremki) betű, Dir a léptetés iránya (verembe szó), Sig az új állapot, FL a szintaxisfa azon eleme, melyből az átmenet származik.
- **function** AddComplete(Left: TELetter; Right: TELetter; Dir: TDirection; Sig: TObject): TETRule; Ugyanaz mint fent, azzal a különbséggel, hogy a hiányzó paramétereket „kitalálja” a függvény. (pl. ha az írandó betű NIL, akkor ugyanazt írja vissza mint amit olvas)
- **procedure** AddSub(R: TEARule); Hozzáad egy ideiglenes olvasó utasítást.
- **procedure** MakeRules; Az ideiglenes utasításokból átmeneteket hoz létre.
- **procedure** Expand; Ideiglenes utasításokra mutató átmeneteket von össze egyetlen átmenetté.

### 3.3.5.8 TETuring

**Bázisosztály:** TETProgram      **Unit:** UTuring

**Feladata:** Futtatható Turing-gép.

**Jellemzők:**

- **Accept:** TEAccept; Elfogadó állapot.
- **LetCount:** Integer; A szalagábécé elemszáma.

**Metódusok:**

- **procedure** GetInputAlphabet(L: TStringList); L-be tölti a bemenő jelek ábécéjét.
- **procedure** GetTapeAlphabet(L: TStringList); **virtual**; L-be tölti a szalagábécé elemeit.
- **function** LetterIndex(T: TLetter): Integer; Visszatér a T betű indexével a szalagábécében.
- **procedure** SetLetters; Beállítja a szalagábécét.
- **procedure** AddState(T: TETState); Hozzáad egy állapotot.
- **class function** RunLang2(C: TTConfig): Boolean; A C konfigurációból (tkp. egy adott bemenetről) elindulva szimulálja a T-gép futását.
- **procedure** RunLang3; **virtual**; Különböző bemeneteket generál, és azokra meghívja RunLang2-t.

**Leszármazottai:**

- TEPushDown: futtatható veremautomata. Unit: UPushDown
  - **Jellemző:** EmptyMode: Boolean; True, ha üres veremmel felismerő automata.

#### 3.3.5.9 TTurThread

**Bázisosztály:** TETHread      **Unit:** UTuring

**Feladata:** A T-gép által felismert nyelvet meghatározó thread.

**Mezők:**

- FTuring: TETuring; Az elemzendő T-gép.

#### 3.3.5.10 TTConfig

**Bázisosztály:** TConfig      **Unit:** UTuring

**Feladata:** A T-gép által felismert nyelvet meghatározó thread.

**Mezők:**

- FTuring: TETuring; Az elemzendő T-gép.

**Jellemzők:**

- **Rule:** TETRule; T-gép átmenet.
- **LetterInd:** Integer; A fej pozíciója.
- **Sentence:** TESentence; A szalag (VA esetén a verem).

**Metódusok:**

- **function** HasRule: Boolean; A konfigurációhoz tartozik érvényes átmenet.

- **function Read:** TLetter; **virtual;** A fej alatt olvasható szimbólum.
- **function State:** TETState; A T-gép állapot.

**Leszármazottai:**

- TConfig: veremautomata egy konfigurációja. Unit: UPushDown

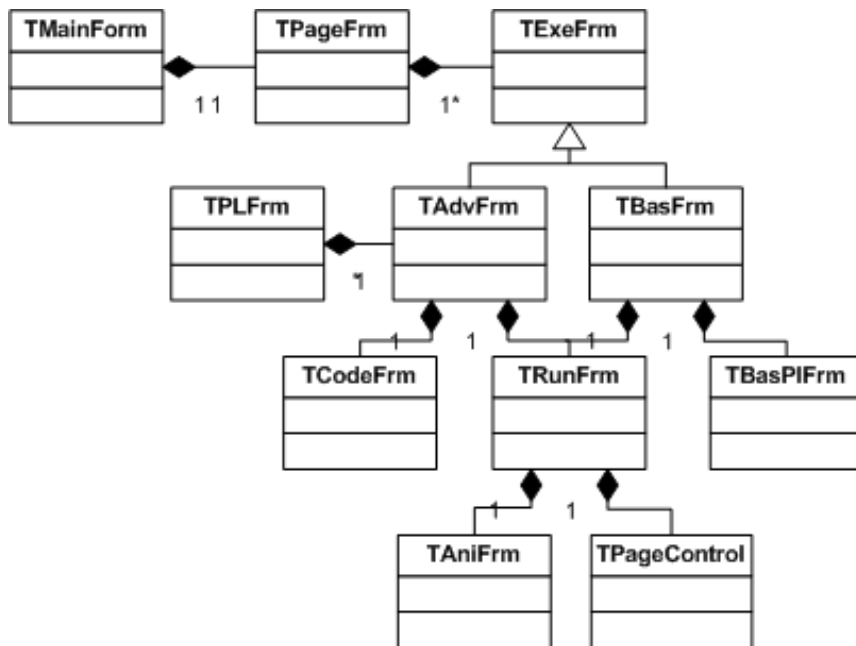
### 3.3.6 A felhasználói felület

A felhasználói felület két fő elemből építkezik: Form és Frame, ezek mindegyike külön unitokban foglalnak helyet.

**Form:** Windows-os ablakot leíró unit. A programban két Form található, a MainForm (főablak), és a PrefDlg (beállítások ablak).

**Frame:** Olyan „keret”, amiben vizuális vezérlőket és a hozzájuk tartozó programkódot helyezhetünk el, majd az így összeállított egység példányait elhelyezhetjük egy form-on, vagy akár egy másik frame-en. Használatával növekszik a programrészek újrafelhasználhatósága, módosíthatósága. A frame-ek nevét Frm végződéssel írom.

A vizuális vezérlők mezői, jellemzői és metódusai többségében egyszerű gyerek-komponensek (gombok, menük, stb.) valamint ezekhez tartozó eseménykezelők. Ezek jelentése értelemszerű, így a dokumentációban nem szerepelnek.



13. ábra: A felhasználói felület osztályai

#### 3.3.6.1 TMainForm

**Bázisosztály:** TForm

**Unit:** UMainForm

**Feladata:** Az alkalmazás főablaka. Kevés funkciót valósít meg, csak felületként szolgál a részfeladatokat megoldó frame-k számára.

#### **Komponensei:**

- Főmenü
- PG: TPageFrm; A munkalapok megjelenítésére szolgáló frame.
- lgLog: Hibaüzenetek és egyéb üzenetek megjelenítésére szolgáló lista.
- Státusz léc.

#### **3.3.6.2 TPrefDlg**

**Bázisosztály:** TForm

**Unit:** UPrefDlg

**Feladata:** Beállítások dialógusablak. Vizuális vezérlőket tartalmaz, melyek az ablak aktiválásakor a beállítások értékeit veszik fel, az OK gombra kattintva pedig az új állapotot tölti vissza a beállításokba. Cancel esetén a beállítások változatlanok maradnak.

A beállításokat az UBase unitban található globális változók tárolják. A program elindításakor ezek a változók a settings.ini fájlból töltődnek be, kilépéskor pedig oda kerülnek elmentésre.

#### **3.3.6.3 TPageFrm**

**Bázisosztály:** TTntFrame

**Unit:** UPageFrm

**Feladata:** Munkalapok megjelenítésére szolgál.

#### **Jellemzők:**

- FileName: WideString; Aktuális munkalap neve.
- Frame[index: integer]: TExeFrm; Az index-edik munkalap.

#### **Metódusok:**

- **function** GetFrame(index: integer): TExeFrm; Az index-edik munkalap.
- **function** NextMakroName: WideString;  
Nevet generál program munkalapok számára.
- **function** NextGramName: WideString;  
Nevet generál táblázatos munkalapok számára.
- **procedure** RefreshFont; Frissíti a betűtípusokat.
- **procedure** SetCursor(Line, Col: Integer); Kijelzi a sor/oszlop információkat.
- **procedure** Run; Futtatja az aktuális munkalapot.
- **function** AddBasic(ProgMode: Boolean = false): Integer;  
Hozzáad egy táblázatos munkalapot. (Ha a ProgMode True, akkor PNY-t, ha false akkor közönséges nyelvtant.)
- **function** AddBasicFromFile(S: widestring): Integer;  
Táblázatos munkalapot betölt.
- **function** AddAdv: Integer; Hozzáad egy program munkalapot.
- **function** AddAdvFromFile(S: widestring): Integer;  
Fájlból betölt program munkalapot.
- **function** AddAdvFromText(S: widestring): Integer;  
String-ből hozzáad program munkalapot. (a string lesz a program kódja)
- **function** AddFromFile(S: widestring): Integer;  
Betölt egy munkalapot.

- **procedure** Close(index: Integer);  
Bezárja az index-edik munkalapot.
- **procedure** SetCaption(index: Integer; S: WideString);  
Beállítja az index-edik munkalap címkéjét.
- **function** Current: TExeFrm; Az aktuális munkalap.
- **function** index: integer; Az aktuális munkalap indexe.
- **function** Count: Integer; Munkalapok száma
- **function** FileIsOpened(S: WideString): Boolean;  
True, ha az S fájl már meg van nyitva.
- **function** CaptionUsed(S: WideString): Boolean;  
True, ha az S címke már használatban van.

#### 3.3.6.4 TExeFrm

**Bázisosztály:** TFrame

**Unit:** UExeFrm

**Feladata:** Munkalapok őssosztálya.

##### **Jellemzők:**

- Modified: Boolean; Változtatva van-e a munkalap (a betöltéshez képest).
- EProgram: TProgram; A munkalaphoz tartozó futtatható program.
- FileName: WideString; A munkalap fájlneve.
- Caption: WideString; A munkalap címkéje (a fülön megjelenő cím).

##### **Metódusok:**

- **procedure** RefreshFont; **virtual**; Frissíti a betűtípusokat.
- **procedure** Run; **virtual**; **abstract**; Futtatja a munkalapot.
- **procedure** Deselect; **virtual**; **abstract**; Inaktiválja a munkalapot.
- **procedure** SelectRule(C: TEBasLine); **virtual**; **abstract**;  
Kijelöli a C futtatható sorhoz tartozó elemeket (a kódszerkesztőben és a Lista nézetben).
- **procedure** SaveToFile(S: WideString = ''); **virtual**; **abstract**;  
Elmenti a munkalapot.
- **procedure** LoadFromFile(S: WideString); **virtual**; **abstract**;  
Betölti a munkalapot.
- **procedure** Clear; **virtual**; **abstract**; Letörli a munkalap tartalmát.
- **procedure** Close; **virtual**; **abstract**; Bezárja a munkalapot.
- **function** SaveDialog: Boolean; **virtual**; **abstract**;  
Meghívja a Mentés dialógus ablakot, és True-val tér vissza, ha a felhasználó jóváhagyta a mentést.

#### 3.3.6.5 TAdvFrm

**Bázisosztály:** TExeFrm

**Unit:** UAdvFrm

**Feladata:** Program munkalap.

**Metódusok:**

- **procedure** SelectMode(M: TProgType);  
Beállítja a munkalap pontos típusát (PNY, T-gép, végállapottal/üres veremmel felismerő VA).
- **procedure** Compile(L: TStringList);  
Lefordítja az L-ben lévő programkódot, és létrehozza a futtatható programot.

**Komponensei:**

- CodeFrm1: TCodeFrm; Kódszerkesztő frame.
- PlFrm1: TPlFrm; Nyelvtan, automata szerkezetét megjelenítő frame.
- RunFrm1: TRunFrm; Levezetésbongésző.

**3.3.6.6 TBasFrm****Bázisosztály:** TExeFrm**Unit:** UBasFrm**Feladata:** Táblázatos munkalap.**Jellemzők:**

- ProgMode: Boolean; True, ha PNY, különben false.

**Metódusok:**

- **function** Compile(L: TStringList): Boolean;  
Lefordítja az L-ben tárolt kódot. True-val tér vissza, ha sikerült.
- **function** GetMode(P: TProgram): Integer;  
A futtatható program állapotáról, típusáról ad vissza információt.

**Komponensei:**

- blf: TBasPLFrm; Táblázat frame.
- runf: TRunFrm; Levezetésbongésző.

**3.3.6.7 TRunFrm****Bázisosztály:** TFrame**Unit:** UBasFrm**Feladata:** Levezetésbongésző.**Mezők:**

- FConfig: TConfig; Az aktuálisan kijelölt konfiguráció.

**Jellemzők:**

- ProgMode: Boolean; True, ha PNY, különben false.

**Metódusok:**

- **procedure** RefreshFont; Frissíti a betűtípusokat.
- **procedure** Clear; Alapállapotba állítja a böngészőt.
- **procedure** Deselect; Inaktíválja a böngészőt.
- **procedure** Run; Futtatja a munkalapot amin található.

**Komponensei:**

- AniFrm1: TAniFrm; Animációs frame.

- PageControl1 : TPageControl; Levezetések tartalmazó lapozható vizuális vezérlő.

### 3.3.6.8 TBasPLFrm

**Bázisosztály:** TFrame

**Unit:** UBasPLFrm

**Feladata:** Táblázatos munkalap számára a táblázat rész.

**Metódusok:**

- **procedure** ToCode(L: TStringList); A táblázat által leírt nyelvtant Prolan kódra fordítja, és beletölti L-be.

**Komponensek:**

- def: TTntNiceGrid; A nyelvtan jelei (terminális stb.)
- sg: TTntNiceGrid; A szabályok táblázata.
- TntMemo1: TTntMemo; A nyelvtan leírását tartalmazó memo.

### 3.3.6.9 TPLFrm

**Bázisosztály:** TFrame

**Unit:** UPLFrm

**Feladata:** A nyelvtanok szabályait, ill. az automaták állapotait és átmeneteit megjelenítő táblázat, valamint ezek pontos matematikai definícióját tartalmazó panel.

**Jellemzők, Mezők:**

- Line: TEBasLine; Az aktuálisan kijelölt futtatható programsor.
- FAdvFrame: TExeFrm; A program munkalap, amin található.

**Metódusok:**

- **procedure** Clear; Törli a frame tartalmát.
- **procedure** Select(C: TEBasLine); Kijelöl egy futtatható programsort.
- **procedure** SelectMode(M: TProgType);  
Beállítja a program pontos típusát (PNY, T-gép stb.)

### 3.3.6.10 TCodeFrm

**Bázisosztály:** TFrame

**Unit:** UCodeFrm

**Feladata:** Kódszerkesztő frame.

**Jellemzők, Mezők:**

- Line: TEBasLine; A kijelölt futtatható programsor.
- FAdvFrame: TExeFrm; A program munkalap, amin található.

**Metódusok:**

- **procedure** SelectLine(L: Integer; C: TColor); Kijelöli az L-edik sort, C színnel.

**Komponensek:**

- Memo1: TCodeMemo; Szövegszerkesztő vezérlő.



### 3.3.6.11 TCodeMemo

**Bázisosztály:** TMemo

**Unit:** UCodeFrm

**Feladata:** Kód szerkesztésére való szövegszerkesztő vezérlő. Alkalmas a sorok számának módosítására (hibajelzéshez és nyomkövetéshez.)

**Jellemzők:**

- property LineColor: TColor; A kijelölt sor színe.
- property SelectedLine: Integer;  
A kijelölt sor száma. Értéke -1, ha nincs kijelölve sor.

**Metódusok:**

- procedure Clear; Kitörli a tartalmát.

### 3.3.6.12 TAniFrm

**Bázisosztály:** TFrame

**Unit:** UAniFrm

**Feladata:** Az animáció megjelenítésére szolgáló frame.

**Jellemzők:**

- IsPlaying: Boolean; True, ha a lejátszás folyamatban van.
- Config: TConfig; Az aktuálisan megjelenített konfiguráció.

**Metódusok:**

- procedure Play;  
Elindítja a lejátszást (végiglépked az összes konfiguráción amíg nem terminál).
- procedure Next; Következő konfigurációra ugrik.
- procedure Prev; Előző konfigurációra ugrik.
- procedure Stop; Megállítja a lejátszást.

**Komponensek:**

- pnlAni: TLANimation; A lejátszást megjelenítő vezérlő.
- ToolBar1: TToolBar; A lejátszást vezérlő gombok eszközléce.

### 3.3.6.13 TAniString

**Bázisosztály:** TExeFrm

**Unit:** UAniString

**Feladata:** Egy stringet animál: egy kezdőpozícióból, kezdőszínből egyenletes átmenettel átranzformálja a stringet egy célpozícióba, célszínbe. Másként: szavakat mozgat, miközben azok elhalványulnak, előtűnnek.

**Jellemzők:**

- Step: Integer; Hányadik lépésnél tart.
- Inverse: Boolean; Ha true, a célból a kezdőállapot felé halad.
- Font: TFont; Betűtípus.
- Text: WideString; A szöveg.
- Canvas: TCanvas; A Canvas amin meg kell jeleníteni az animációt.
- Size: TSize; A szöveg méretei pixelben.

- StartPos: TPoint; Kezdőpozíció.
- EndPos: TPoint; Célpozíció.
- StartColor: TColor; Kezdőszín.
- EndColor: TColor; Célszín.
- Time: double; A teljes átmenet időtartama (amíg a kezdő állapotból a végállapotba ér).
- Interval: Cardinal; Egy lépés időtartama (milyen gyakran frissítse a képet)

**Metódusok:**

- **procedure** Animate(Sender: TObject); Megtesz egy lépést, és kirajzolja.
- **procedure** Draw; Kirajzolja az aktuális állapotot.
- **procedure** Move; Megtesz egy lépést rajzolás nélkül.
- **procedure** Start; Kezdőállapotba állítja.
- **procedure** StartTimer; Elindítja az animációt.
- **procedure** Finish; Leállítja az animációt.

### 3.3.6.14 TAniAnimation

**Bázisosztály:** TPaintBox      **Unit:** UAniString

**Feladata:** TAniString-ek felhasználásával ábrázol egy PNY, T-gép vagy VA konfiguráció-átmenetet. Egy szó részét cseréli ki. A lecserélt rész felfelé mozogva eltűnik, miközben a helyére kerül az új részszó.

**Mezők:**

- FLeft: TAniString; A részszótól balra lévő részszó.
- FRight: TAniString; A részszótól jobbra lévő részszó.
- FUp: TAniString; A részszó.
- FDown: TAniString; Az új részszó.

**Jellemzők:**

- Inverse: Boolean; Ha true, a célból a kezdőállapot felé halad.
- Font: TFont; Betűtípus.
- Text: WideString; A szöveg.
- Time: double; A teljes átmenet időtartama (amíg a kezdő állapotból a végállapotba ér).
- Interval: Cardinal; Egy lépés időtartama (milyen gyakran frissítse a képet)
- First: Integer; A Text-ben kicserélendő részszó első betűjének indexe.
- Last: Integer; A Text-ben kicserélendő részszó utolsó betűjének indexe.
- NewString: WideString; Az új részszó.
- Pos: TPoint; A szó job felső sarkának pozíciója.
- UpColor: TColor; A részszó színe.
- Freezing: Boolean;  
Ha értéke true, nem mutatja animációt, hanem végig a kezdőállapot látható. (ez abban az esetben kell, ha pl. T-gépnél csak a fej mozog, de nem történik írás.)
- OnFinished: TNotifyEvent; Eseménykezelő, amit akkor hív meg ha véget ért a lejátszás.

**Metódusok:**

- **procedure Start;** Elindítja az animációt.
- **procedure Init;** Kezdőállapotba állítja a komponenst a jellemzők alapján.
- **procedure IniConfig;** A szín, stílus, idő stb beállításokat alkalmazza a TAniString-ekre.
- **procedure Finish;** Befejezi az animációt.
- **procedure Draw;** Kirajzolja az aktuális állapotot.
- **function IsPlaying:** Boolean; **True**, ha éppen tart a lejátszás.
- **property read** FOnFinished **write** SetOnFinished;
- **procedure Animate(Sender: TObject);**

**3.3.6.15 TTntNiceGrid**

**Bázisosztály:** TTntStringGrid      **Unit:** UTntNiceGrid

**Feladata:** A TNT-s komponens egy módosítása, hogy jobban megfeleljen a program igényeinek. Szébb fejléc, megváltozott szerkesztési működés.

**Mezők:**

- **FModified:** Boolean; **True**, ha módosította a felhasználó a komponenst.

**Metódusok:**

- **procedure SaveToStream(S: TStream);** Stream-be menti.
- **procedure LoadFromStream(S: TStream);** Stream-ből betölti.
- **procedure InsertRow(N: Integer);** Beszúr egy sort.
- **procedure DeleteRow(N: Integer);** Töröl egy sort.
- **procedure EditCell(X, Y: Integer);**  
Szerkesztési módba kapcsolja az X, Y koordinátájú cellát.

### 3.4 Tesztelés

Mivel a samples könyvtárban található példafájlok célja az, hogy lehetőleg minél több használati esetet lefedve bemutassa a program működését, megfelelőek a teszteléshez is. A fájlok kommentek formájában tartalmazzák felhasználásuk módját, valamint hogy milyen használati esetet fednek le. Ezeket az információkat alábbiakban röviden összefoglalom, valamint azt is, hogy mely utasítások, fordítási direktívák tesztelésére alkalmas az adott fájl. Az alapvető utasítások (pl. **if**, makródefiníció stb.) majdnem mindegyikben előfordul, így ezeket nem említem külön. A fájlok lefuttatása után szemrevételezéssel ellenőrizzük, hogy a forráskódból megfelelő nyelvtan vagy automata jött létre, valamint azok a megfelelő levezetéseket és nyelvet generálják, ill. a megfelelő nyelvet ismerik-e fel.

- pny -  $a^{(2^n)}$ .pla: PNY
  - Generált nyelv:  $\{a^{2^n} | n \in \mathbb{N}_0\}$
  - **#pragma leftmost**
  - Mátrix blokk
  - blokkok egymásba ágyazása
  - blokkok szekvenciája
- pny -  $a^n b^n c^n$ .pla: PNY
  - Generált nyelv:  $\{a^n b^n c^n | n \in \mathbb{N}_+\}$
- pny - faktoriális.pla: PNY
  - Faktoriális eljárás. Az eljárás első paraméterében szereplő szimbólum darabszámát beállítja  $n!$ -ra, a másodikat pedig nullára, ahol  $n \in \mathbb{N}_0$  a második paraméterben szereplő szimbólum darabszáma.
  - **#include**
- pny - közönséges - zárójelezés.pla: PNY
  - Generált nyelv: helyes zárójelezések nyelve.
  - Közönséges nyelvtan megadása programozott nyelvtannal.
  - **#pragma grammar**
- pny – Lindenmayer - alga.pla: PNY
  - Generált nyelv: algás Lindenmayer-rendszer nyelve (1).
  - **#pragma lindenmayer**
- pny – Lindenmayer - Fibonacci.pla: PNY
  - Generált nyelv: olyan nyelv, melynek szavainak hossza Fibonacci-sorozatot alkotnak.
- pny – Lindenmayer - MISS3.pla: PNY
  - Generált nyelv:  $\{x^n | n \neq 3\}$ .
- pny - LNKO.pla: PNY
  - A két paraméter előfordulási darabszámainak, mint két nem negatív egésznek kiszámolja a legnagyobb közös osztóját. Az első paraméter új darabszáma az eredmény, a második paraméteré nulla.
- pny - mátrix.pla: PNY
  - Generált nyelv:  $\{uu | u \in \{a, b, c\}^*\}$
  - mátrixnyelvtanok
  - **try** utasítás, pontozott szabály
- táblázat - közönséges -  $a^{(2^n)}$ .gr: Közönséges környezetfüggő nyelvtan

- Generált nyelv:  $\{a^{2^n} | n \in \mathbb{N}_1\}$
  - Környezetfüggő nyelvtan.
- táblázat - közönséges - zárójelezés.gr: Közönséges másodrendű nyelvtan
  - Generált nyelv: helyes zárójelezések nyelve
- táblázat - pny -  $a^n b^n$ .gr: PNY táblázattal.
  - Generált nyelv:  $\{a^n b^n | n \in \mathbb{N}_0\}$
- turing -  $0^n 1^n$ .pla: T-Gép
  - Felismert nyelv:  $\{0^n 1^n | n \in \mathbb{N}_0\}$
  - **#pragma turing**
- turing - palindróma.pla: T-Gép
  - Felismert nyelv:  $\{u | u \in \{0, 1\}^* \wedge u = u^{-1}\}$
  - **state** utasítás
- turing - u u.pla: T-Gép
  - Felismert nyelv:  $\{u \# u | u \in \{0, 1\}^*\}$
- turing - u u - nemdet.pla: T-Gép
  - Felismert nyelv:  $\{uu | u \in \{0, 1\}^*\}$
  - Megsejti hol a második szó eleje.
  - Nemdeterminisztikus Turing-gép.
  - Elágazó blokk: < >
- turing - u u - nemdet.pla - print: T-Gép
  - Ugyanaz mint a fenti, azzal a különbséggel, hogy a kulcslépéseket **print** utasítással jelenítjük meg.
  - **#pragma printmode, print** utasítás használata.
- turing - véges automata.pla: T-Gép
  - Felismert nyelv:  $\{u | u \in \{0, 1\}^* \wedge 2 | l_1(u)\}$
  - véges automatát modellező T-Gép.
- turing - zárójelezés.pla: T-Gép
  - Felismert nyelv: helyes zárójelezések nyelve
- verem -  $0^n 1^n$ .pla
  - Felismert nyelv:  $\{0^n 1^n | n \in \mathbb{N}_+\}$
  - **#pragma pushdown** (végállapottal felismerő veremautomata)
- verem - palindróma - nemdet.pla
  - Felismert nyelv:  $\{u | u \in \{0, 1\}^* \wedge u = u^{-1}\}$
- verem - zárójelezés.pla
  - Felismert nyelv: helyes zárójelezések nyelve
  - **#pragma epushdown** (üres veremmel felismerő veremautomata)
  - $\epsilon$ -átmenet.

## 4 Irodalomjegyzék

1. **Dr. Hunyadvári, László.** Automaták és formális nyelvek II. [Online]  
<http://aszt.inf.elte.hu/~hunlaci/fnyau2.pdf>.
2. Minimalist GNU for Windows. [Online] <http://www.mingw.org/>.
3. *Delphi Yacc & Lex.* [Online] <http://www.grendelproject.nl/dyacclex/>.
4. TNT Unicode Controls. [Online]  
<http://www.yunqa.de/delphi/doku.php/products/tntunicodecontrols/index>.
5. **Swett, Justin.** A Heap ADT (Priority Queue) Example in Delphi. [Online]  
<http://cc.embarcadero.com/item/17246>.