# Végesállapotú transzducerek mindenkinek

Gyepesi György[1], Gábor Bálint[2], Halácsy Péter[3], Kertész Zsuzsa[1]

[1] ALL Kut. és Fejl. Szöv. {ggyepesi,kzsuzsa}@all.hu
[2] BME Kognitív Tudományi Tanszék, bgabor@cogsci.bme.hu
[3] BME Média Oktató és Kutató Központ, hp@mokk.bme.hu

**Kivonat** Cikkünkben bemutatunk két véges állapotú fordítóval működő nyílt és szabad szövegfeldolgozó komponenst, a `huntokent` flexibilitásban meghaladó tokenizálót és a `hunmorph` csomag jelenlegi elemző programjánál az `ocamorph-nál` nagyságrenddel gyorsabban futó morfológiai elemzőt. Mindkét szoftverre jellemző, hogy nagy korpuszok gyors feldolgozására készült és nem csak parancssorból lehet őket használni, hanem fejlesztői könyvtárként bármilyen alkalmazásba könnyen beilleszthetőek.

## 1. Tokenizáló és mondatrabontó

A neten már több tucat tokenizáló program érhető el [6], azonban ezek egyike sem felel meg néhány számunkra fontos követelménynek. Egyrészt fontos a bővíthetőség és konfigurálhatóság, mert más-más alkalmazásoknak másmilyen szegmentálásra van szükségük. Másrészt nagyon fontos szempont a sebesség, hiszen gigabájt méretű korpuszok esetén egy okos, de lassú tokenizáló használhatatlanná válik.

### 1.1. Standoff annotáció

A hagyományos szövegfeldolgozó csővezeték az eredeti szöveget lépésről lépésre átírja, transzformálja. A végeredmény manapság általában egy XML fájl különböző elemekkel teletűzdelve. Ezzel több probléma is van: egyrészt a formátum, az ún. inline annotáció nem teszi lehetővé, hogy többféle annotációt használjunk ugyanarra a szövegre. Ez azt is jelenti, hogy későbbiekben nem tudjuk fejleszteni vagy éppen lecserélni a feldolgozó sor valamelyik komponensét.

Kevésbé triviális, de ugyanolyan fontos probléma, hogy a monolitikus inline annotáció használata esetén az eredeti szöveget, a jelenségek eredeti kontextusát elvesztjük.

Reméljük, a fenti problémákat megoldja az egyébként már több helyen is alkalmazott ún. *standoff* annotáció. Ennek a lényege, hogy az eredeti szöveget érintetlenül hagyjuk és a szöveg mellett csak felsoroljuk, hogy a szöveg mely része milyen címkét, annotációt kapott (de nem tesszük bele a szövegbe a címkét).

### 1.2. Sebesség

Tokenizáláshoz a leggyakrabban alkalmazott megközelítés az egymás utáni cserélgetés. Ez általában csővezetékbe kötött `lex/flex` programok (lásd huntoken vagy a Multext segment), `sed` szkriptek vagy több egymásutáni sztring replace parancs végrehajtásával végzik, mint Greffenstette [1] mára klasszikussá vált tokenizálója. Ezekben az a közös, hogy a szöveget többször olvassák végig, tulajdonképpen több reguláris kifejezés illesztés történik egymás után.

Például a hunglish projektben használt egyszerű tokenizálóban csővezetékbe kötött `sed` szkriptek egymás után több cserét hajtanak végre. Első lépésben a mondatvége jelek után szóközt, majd – hogy a kötőjeleket leválassza a szavakról – a kötőjelek elé majd után újabb szóközt szúr be, majd a több szóközt egymás mellett egyre cseréli.

Ezzel a megközelítéssel az a baj, hogy a kontextusérzékeny beszúrás – főleg, ha regexpes sztring csere műveletként implementáljuk – eléggé lassú tud lenni.

### 1.3. FST alapú tokenizálás

A regexp illesztés utáni cseremműveletek egymásutánja kifejezhető *véges állapotú transzducerekkel* (FST). Ennek az a nagy előnye, hogy az egymás után fűzött FST-k az ún. kompozíció művelettel egy nagy FST-vé alakíthatóak, aminek köszönhetően egy sztring feldolgozásához a sztringet csak egyszer kell végigolvasni. Így a kompozicióval előállított transzducer kevesebb input-output műveletet igényel, és a számítási igénye is alacsonyabb. Bár az állapotok száma általában nő, ez csak az automata memóriaigényét befolyásolja.

Minden karakterhez meghatározzuk azt az FST csere-szabályt, amelyik leellenőrzi, hogy ennek a karakternek az egyes előfordulásai – a kontextusuk alapján – token- vagy mondatzáró pozicióban vannak-e. A tokenizáló javarészt ilyen csereszabályok által meghatározott transzducerek kompozíciójából áll. Vannak olyan bonyolultabb esetek, amikor távoli karakterek hatását kell figyelembe vennünk egy esetleges tokenzáró karakter vizsgálatakor. Ilyen eset például amikor zárójelek között szereplő mondatok esetén csak a zárójelen kívüli – ezeket magában foglaló – mondat végét szeretnénk mondathatárnak nyilvánítani. Ilyenkor szükség lehet olyan transzducerek használatára, amelyek ideiglenes, a későbbi automaták számára üzenetet hordozó szimbólumokat helyeznek el a szövegben. Megjegyezzük, hogy ez esetben is transzducerek kompozíciójával dolgozunk, és nem egymás után futtatjuk őket.

### 1.4. Implementáció

Az SFST[4] nyílt forráskódú FST fordítót használjuk a hálózat építésére, de futásidőben saját algoritmussal dolgozunk. A tokenizálás azért gyors, mert tudjuk,

[4] http://www.ims.uni-stuttgart.de/projekte/gramotron/SOFTWARE/SFST.html

hogy (1) determinisztikus automatával van dolgunk, (2) a felszíni szimbólumok csak karakterek lehetnek, (3) a végállapotoknak nincs jelentősége, (4) epszilon sztringgel minden állapotból csak egyfelé lehet menni és (5) ott vannak a tokenhatárok, ahol csak egy ilyen élen lehet továbbmenni.

Az SFST-vel előállított automatát egy saját elemzőprogrammal használjuk, ami folyamatosan olvassa a szöveget, és ha tokenhatárhoz ért, akkor annak pozicióját visszaadja.

## 2. FST alapú szóelemző

Az affix-stripping alapú morfológiai elemzés a hunspell nyelvi erőforrásait, az un. `affix` és `dictionary` állományait használja. A morphdb kiterjeszti ezt a szabályleírási formát morfológiai információkkal, és keretet biztosít a morfológiai szabályok nyelvészeti fogalmakkal történő leírására. Az FST morfológiai elemzők modellje a kétszintű morfológia [3]. Az XFST, SFST és más FST alapú morfológiai elemzők saját, gyakran körülményes, az FST-építés technikáját is szabályozó nyelven követelik a morfológiai szabályok leírását. Emiatt nyelvészek számára nehézkes az ilyen elemzők használata és a morfológiai szabályok ilyen nyelven való megfogalmazása.

Magának az FST alapú elemzésnek számos előnye van: az FST erőforrást (a transzducer állapotainak és input-output-címkézett éleinek listája) beolvasó és a beolvasott transzduceren morfológiai elemzést végző program bármilyen programozási nyelven nagyon egyszerű és rövid. Az affix-stripping alapú elemzők ezzel szemben meglehetősen bonyolult programok. Ráadásul az FST alapú elemzés sebessége legalább egy nagyságrenddel nagyobb az affix-stripping alapúakénál. Az FST erőforráson működnek az általános transzducer algoritmusok, és más eljárások is ki tudják használni a reprezentáció egyszerűségét.

Összefoglalva: az affix-stripping alapú elemzők számára egyszerű az erőforrás elkészítése, az FST alapú elemzők morfológiai szabályreprezentációja viszont sokkal egyszerűbb és az elemző algoritmus lényegesen gyorsabb. Ezért van értelme affix-stripping erőforrásból FST építésének.

### 2.1. FST építés folyamata

Az affix-stripping affix és dictionary fájljaiból először egy címkézett élű és csúcsú irányított gráfot építünk. A gráf csúcsai a szótárban szereplő szavak és az affix file-ban szereplő szabályok. A csúcsok címkéje az a morfológiai annotáció, ami a szóhoz illetve toldalékhoz tartozik. Egy szócsúcsból azokba a szabálycsúcsokba vezet él, amely szabályokkal a szó toldalékolható, egy szabálycsúcsból pedig azokba a szabálycsúcsokba vezet él, amely szabályokkal a toldalékolás folytatható. Két szócsúcs között akkor van él, ha a két szó összetételben szerepelhet. Az élek címkéje írja le azt a változást, amit a morfológiai művelet kivált.

Ebből a gráfból építjük a transzducert úgy, hogy végrehajtjuk az éleken szereplő műveleteket és a műveletek eredményét írjuk a transzducer éleire mint inputot. A gráf-élek végcsúcsaiban szereplő címkék lesznek a megfelelő transzducer éleken megjelenő outputok. Problémát csak az okoz, amikor az affix fájl egy suffix szabályából megy él egy prefix szabályba (pl. igeképzőkön megjelenik az igekötő, vagyis az igeképzőnek megfelelő csúcsból él megy egy igekötő csúcsába). Ilyenkor megismételjük a gráfnak azt a részgráfját, ami a suffix szabály csúcsán végződik, a másolat minden a suffix szabályétól különböző csúcsát nem elfogadóvá tesszük a transzduceren és a másolat minden kezdőállapotába élet húzunk a prefix csúcsából.

Az így elkészített transzducert betűsítjük, azaz minden élet felbontunk olyan élekre, melyeken egy-egy karakter az input, végül „minimalizáljuk": determinizáljuk és minimalizáljuk azt az automatát, amit úgy kapunk, hogy egy speciális jellel minden élen összekötjük az inputot és outputot majd újra szétválasztjuk őket.

## 3. Összefoglalás

A két bemutatott FST alapú eszköz, a szó- és mondathatár bejelölő és a morfológiai elemző (tövező) új NLP funkcionalitást nem hozott a BME MOKK által fejlesztett szószablya családba, viszont mérnöki szemmel nézve nagy előrelépést jelentenek.

Egyrészt sokkal gyorsabban végzik el feladatukat, mint az eddigi eszközök és támogatják a standoff annotációt. Ennél talán fontosabb az az új lehetőség, ami a bonyolult online elemzők esetében egyáltalán nem volt adott: egy BA-t végzett informatikus képes bármilyen programozási nyelvre tokenizálót, mondatrabontót és morfológiai elemzőt írni, mert a végesállapotú transzducer online rétege max. 100 sorban megírható. Cserébe az erőforrások előállítása nehezebb, de ezt egyszer kell futtatási plattformtól függetlenül előállítani. Reméljük, e két új feljesztéssel a `huntoken` és `hunmorph` programok használhatóvá válnak nagy ipari rendszerekben is, ahol a feladat sok szöveg gyors és hatékony feldolgozása.

## Hivatkozások

1. Gregory Grefenstette, Pasi Tapanainen, 'What is a word, what is a sentence? problems of tokenization'. In: The 3rd International Conference on Computational Lexicography, 79–87, Budapest, (1994).
2. Halácsy, P., Kornai, A., Németh, L, Sas, B., Varga, D., Váradi, T., and Vonyó, A: A Hunglish korpusz és szótár, In: III. Magyar Számítógépes Nyelvészeti Konferencia, Szegedi Tudományegyetem, (2005).
3. Koskenniemi, K..: Two-level morphology: A general computational model of word-form recognition and production. Tech. rep. Publication No. 11, Department of General Linguistics, University of Helsinki, (1983).

4. Ronald M. Kaplan: A Method for Tokenizing Text, In: *Inquiries into Words, Constraints and Contextus*, (2005).
5. Trón, V., Halácsy, P., Rebrus, P., Rung, A., Vajda, P., and Simon, E.: Morphdb.hu: Hungarian lexical database and morphological grammar, In: Proceedings of 5th International Conference on Language Resources and Evaluation. ELRA, 1670–1673, (2006).
6. Ying He, Ph.D. és Mehmet Kayaalp, M.D., Ph.D.: A Comparison of 13 Tokenizers on MEDLINE , Technical report.
`http://lhncbc.nlm.nih.gov/lhc/docs/reports/2006/tr2006003.pdf`, (2006).