emtsv – Egy formátum mind felett

Indig Balázs[1,2], Sass Bálint[1], Simon Eszter[1],
Mittelholcz Iván[1], Kundráth Péter[1], Vadász Noémi[1]

[1]MTA Nyelvtudományi Intézet,
1068 Budapest, Benczúr u. 33.
[2]ELTE Bölcsészettudományi Kar
1088 Budapest, Múzeum krt. 4.
vezeteknev.keresztnev@nytud.mta.hu

Kivonat Az `e-magyar` nyelvfeldolgozó rendszer elkészülése óta több ízben felmerült az igény a hatékonyságának növelésére és használhatóságának egyszerűsítésére, melyek figyelembevételével továbbfejlesztettük a meglévő szövegfeldolgozó rendszert. Célunk a modulok közötti hatékony kommunikáció megvalósítása, valamint az egyes modulok láncba építésének és önálló használatának egyenrangú támogatása. Ezt egy nemzetközi szabványokkal összeegyeztethető, egyszerű, egységes és általános be- és kimeneti formátum használatával valósítjuk meg. Ez terveink szerint hosszú időre jövőállóvá teszi a rendszert, valamint még szélesebbre tárja a külső fejlesztők előtt a kaput, hogy saját moduljaikat a rendszerünkhöz tudják illeszteni, megosztva a meglévő kompetenciákat a magyar nyelv számítógépes feldolgozásának területén. A cikkben bemutatjuk az `e-magyar` új verzióját, az `emtsv` elnevezésű rendszert.
Kulcsszavak: e-magyar, emtsv, eszközlánc, erőforrás, tsv, modularitás

1. Bevezetés

Az `e-magyar` nyelvfeldolgozó rendszer [1] elkészültekor nem kisebb célt tűzött ki maga elé, mint hogy a magyar nyelv feldolgozásához szükséges state-of-the-art eszközöket integrálva egy egységes, könnyen kezelhető, karbantartott és frissített rendszert alkosson, mely elősegíti a magyar nyelv kutatás- és alkalmazásközpontú feldolgozását egyaránt. Fontos cél volt, hogy a rendszer kutatási célra teljesen nyílt legyen – bátorítva ezzel a későbbi bővítést –, ugyanakkor a laikusok számára is könnyen használhatóvá és jó kísérletező tereppé váljon, mely az elérhető legjobb teljesítményt adja úgy a feldolgozás sebessége, mint a kimenet helyessége tekintetében.

A rendszert közzététele óta jónéhányan letöltötték, és használják a mai napig is. Történtek próbálkozások nagyméretű korpuszok (MNSZ2, Webkorpusz) elemzésére is, aminek következtében korábban ismeretlen hibák és gyengeségek kerültek napvilágra. Ezeket a magyar nyelvtechnológiai közösség közreműködésével javítottuk, illetve a további fejlesztéseknél figyelembe vettük. Jelen cikkben két szempont összefonódásának mentén szeretnénk bemutatni az elvégzett munkát.

Az első a modulok közötti egységes kommunikációs formátum kérdése. Ez az `e-magyar` első verziójában, amiatt, hogy a rendszer integrációja a GATE [2] keretrendszerben valósult meg, adott volt, célszerűnek tűnt a GATE által definiált belső formátumot használni. A felmerült igényekből és az üzemeltetési tapasztalatokból az tükröződött, hogy a felhasználók jelentős része nem ismeri vagy nem kívánja használni munkájához a GATE rendszert: a nyelvi érdeklődésű felhasználóknak kényelmetlen volt, a technikai érdeklődésűeknek pedig szükségtelenül nehézkes. Továbbá a GATE sok esetben inkább megnehezíti az eszközök használatát, az eszközökkel kapcsolatos munkát, mivel az általa bevezetett komplexitás (bonyolult telepítés, nehéz hibakeresés, kényelmetlen formátum, túl nagy erőforrásigény az XML-re alapuló standoff annotáció következtében) sok esetben aláássa a stabilitást, mely szolgáltatáskimaradáshoz is vezethet. Ezért úgy döntöttünk, hogy egy GATE-től független új, egységes formátumot hozunk létre, mely könnyen összeegyeztethető a nemzetközi trendeknek megfelelő szabványokkal. Ezzel megnyílik az út a meglévő eszközök külön-külön modulként történő használatára, az egyes modulok kimenete jobban áttekinthetővé (ezáltal manuálisan könnyebben módosíthatóvá) válik, valamint a rendszerbe könnyebben beépíthetők lesznek a mások által készített különféle – akár nyelvfüggetlen – eszközök. Emellett a GATE-hez való kapcsolódás is megmaradhat megfelelő formátumkonverziós eljárások segítségével.

A második fejlesztési szempont magának az architektúrának az átdolgozása volt, mely az `e-magyar` megalkotása előtt rendelkezésre álló korábbi modulok öröksége felől (nem egységes nyelvi kódok és programnyelvek, nem kellően modularizált és átlátható felépítés) a jelenleg és a jövőben elvárt funkcionalitások (egységesség, felcserélhetőség, összehasonlíthatóság, tanulmányozhatóság) kiszolgálása felé tolja a hangsúlyt.

A cikkben bemutatjuk, hogy az első verzióhoz képest milyen módon alakítottuk át a felhasznált eszközöket abból a célból, hogy a Unixból ismert „eszköztár filozófiának” és az „egy modul egy feladatot végezzen el, de azt tegye jól” elvnek megfelelően az átstrukturált modulok akár egymástól maximálisan függetlenül is, de szükség esetén egymással összekapcsolhatóan és egymással teljesen kompatibilis módon működjenek. Ezzel létrejön az a fontos új lehetőség, hogy a szerelőszalag tetszőleges szakasza lefuttatható, azaz bármely ponton be, illetve ki tudunk lépni, ami magával hozza annak a lehetőségét, hogy az egyes modulok között a felhasználó szabadon rendelkezhet az adattal, akár kézzel is módosíthatja azt, amíg betartja a formátum által támasztott elvárásokat.

A fejlesztés során körültekintően jártunk el, hogy lépést tartsunk más, egy adott nyelvből kiinduló, de többnyelvűnek vagy akár univerzálisnak szánt feldolgozóláncokkal, valamint a megváltozott igényekkel, melyek újabban a installálást és karbantartást nem igénylő, skálázható felhőalapú technológiákat részesítik előnyben, mintegy szolgáltatásként tekintve az feldolgozóláncra. A következő fejezetben az `e-magyar`hoz hasonló, jelenleg elérhető nyelvfeldolgozó rendszereket tekintjük át, hogy összehasonlíthassuk őket rendszerünkkel.

2. Háttér

A magyart mint elsődleges célnyelvet tekintve, az `e-magyar`-ral egyedül a *Magyarlánc* [3] hasonlítható össze[1], ami jelenleg a 3.0 verziónál tart. Ez egy Java-alapú, zártan integrált (*tightly coupled*) láncot bocsájt a felhasználók rendelkezésére. A rendszer tanulmányozása közben azt látjuk, hogy a rendszer a legfrissebb nemzetközi state-of-the-art modulokat használja, ugyanakkor nem bővíthető kényelmesen új modulokkal. Arra alkalmas, hogy nagy mennyiségű magyar szöveget részletes nyelvi elemzéssel lássunk el megfelelő minőségben, de a rendszer módosítása, esetleges új modulokkal való kiegészítése nem volt elsődleges prioritás a rendszer fejlesztése közben, így az nehézkes. Alkalmazói felhasználásra kiváló, de továbbfejlesztésre kevésbé alkalmas, mely tulajdonságból fakadóan a felhasználót az eszköz létrehozójához nem egyenrangú kapcsolat köti. A Magyarlánchoz hasonló rendszerekre a nemzetközi színtéren is több példa van, melyek általában ugyanezektől a hiányosságokról szenvednek. Ugyanakkor olyan előnyöket is fel tudnak mutatni, mint a nyelvfüggetlenség (vagy legalábbis sok nyelv támogatása), illetve nagy mennyiségű adat gyors feldolgozása. Ezekkel az aspektusokkal nem szándékozunk versenyre kelni, hanem arra koncentrálunk, hogy a magyar nyelvre a legjobb eredményt a leghatékonyabban állítsuk elő, valamint egy a szabványokhoz közel álló, jól átalakítható formátumot hozzunk létre. A teljes irányítást a felhasználó kezébe szeretnénk adni azáltal, hogy egy nyíltan integrált (*loosely coupled*) rendszert hozunk létre.

Fontos jellemzője a több nyelvet támogató eszközöknek, hogy jellemzően a *Universal Dependencies and Morphology*[2] (UD) nevű nemzetközileg elterjedt, univerzálisnak szánt annotációs sémát használják. Az általános célú egységes annotáció nyilvánvaló előnyei mellett érdemes látni, hogy az ilyen annotáció nem feltétlenül képes egy nyelv morfológiai jelenségeinek teljeskörű leírására. Ezért tartottuk fontosnak, hogy a magyar esetében egy jó minőségű, speciálisan magyar nyelvre kialakított morfológiai elemzőt építsünk be a láncba, az `emMorph`-ot [4]. Az általunk ismert nyelvfüggetlen elemzőláncok közül csak két eszközt emelünk ki példaként, mivel a többi meg nem nevezett alternatíva is ugyanazokkal az ismertetett hátrányokkal bír.

A *UDPipe* [5] C++ nyelven íródott nagyjából az `e-magyar` rendszerrel egy időben, és a célja általános szövegek elemzése a UD annotációs sémáját és formátumát követő tanítóanyag felhasználásával. Bár sok nyelvre van interfésze, valamint valóban hatékonynak mondható, nem teszi lehetővé a könnyű kiterjesztést és fejlesztést, annak ellenére, hogy forráskódja elérhető[3]. Többek között nem ad lehetőséget saját modulok, például szabályalapú morfológiai modul bevezetésére. Hasonló programnak indult a Python-alapú *Spacy*[4], mely eredetileg

[1] Habár az összehasonlítás alapját képező láncok moduljai között van átfedés, az összehasonlításban a lánc egészének felépítésére koncentrálunk, mely független az egyes moduloktól.

[2] `http://universaldependencies.org`

[3] `https://github.com/ufal/udpipe`

[4] `https://spacy.io`

zártan integrált modulokból állt, de a 2.0 verzióval a támogatott nyelvek számának növelése céljából architekturálisan egyre nyíltabbá válik a fejlesztés során[5].

Egy másik stratégiát követ a *WebSty*[6], mely a CLARIN-PL, illetve a *Weblicht*[7], mely a CLARIN-D projekt keretében jött lére. Ezekben az eszközökben ugyanis integrálni próbálják a meglévő – akár nyelvfüggő – eszközöket is, hogy az egyes nyelvek jobban támogatva legyenek. Egyedüli kritérium, hogy a felhasznált eszközök támogassák a UD formátumot. A megközelítés lényege, hogy egy nagy számítógépklaszteren a felhőben futó feladatütemező segítségével az egyes modulok szükség szerint skálázhatóak legyenek a feladatok sorbaállításával. Az egész rendszer egy webalapú API-n keresztül érhető el, melyben feladatokat kell megadni az adatfájl kíséretében, és megvárni, amíg a feladat feldolgozásra kerül. Ebben az esetben a szoftver forráskódja nem érhető el saját példány futtatása céljából, valamint a modulok fejlesztése külső fejlesztőként nem lehetséges.

Az `e-magyar` rendszer új verziójában, az `emtsv`-ben az ismertetett rendszerek fent leírt hátrányait szeretnénk kiküszöbölni úgy, hogy ugyanakkor azok előnyös tulajdonságait is át tudjuk venni. A következő fejezetekben ismertetjük az ezzel kapcsolatban végzett munkát.

3. Az egységes adatformátum

Az `e-magyar` rendszer klasszikus felépítése [6] nagyban támaszkodott az eredeti eszközök örökölt felépítésére, így függött azok bemeneti és kimeneti formátumaitól. Az eredeti elképzelés szerint a GATE volt az architektúra azon rétege, mely megteremti a közös, egységes adatformátumot, és így átjárhatóságot biztosít a független, egymásról mit sem tudó modulok között. Az elképzelés egészen addig megfelelő volt, amíg a felhasználó a GATE rendszer ökoszisztémáján belül kívánta használni a rendszert.

A közös formátum a GATE által definiált GATE XML formátum volt. Ez nem egy szabványos és bárki által könnyen implementálható megoldás, mivel nem ismert a formátumot leíró DTD vagy Schema fájl. Ezekre elméletben nincs is szükség, hiszen elméletileg a GATE rendszer a formátum egyetlen előállítója és feldolgozója, azaz gyakorlatilag belső formátumnak tekinthető. A felépítését tekintve standoff annotációként teszi hozzá a bemeneti szöveghez az összes elemzési információt. Ez azzal jár, hogy mivel a szöveg és az annotáció egymástól elkülönülten helyezkedik el egyazon XML-fájlban, folyamatosan ugrálni kell a kettő között – például az XML-feldolgozásból ismert DOM stratégiával fát építve –, ehhez pedig végeredményben a teljes szöveget és annotációt a memóriában kell tartani. Ez a követelmény legjobb esetben is lelassítja a nagyméretű XML-fájlok feldolgozását (a GATE nem erre lett tervezve), mivel lehetetlenné teszi az adatfolyamként való feldolgozást. Emellett a GATE rendszer nehézkes telepíthetősége önmagában is nagyban megnövelte a rendszer komplexitását a felhasználók és

[5] Bár a SpaCy és az `e-magyar` fejlesztésének iránya megegyezik, a jelenlegi állapotuk túl távoli ahhoz, hogy összemérhetőek legyenek.

[6] `http://ws.clarin-pl.eu/websty.shtml`

[7] `https://weblicht.sfs.uni-tuebingen.de/weblichtwiki/index.php/Main_Page`

az üzemeltetők számára is, függetlenül attól, hogy valóban szükségük volt-e a biztosított többletfunkciókra.

Ez motivált minket arra, hogy egy nem XML-alapú, GATE-től függetlenül működő, egyszerű, egységes és könnyen kezelhető formátumot tervezzünk, amely könnyen átalakítható más formátumra. Fontosnak tartottuk, hogy ne zárjunk ki egy potenciális felhasználót sem a formátum miatt. A könnyű konvertálhatóság lehetővé teszi, hogy a nemzetközileg elismert szabványos formátumokra, mint a CoNLL-X [7] vagy a CoNLL-U[8], vagy akár GATE XML formátumra is, át lehessen alakítani a meglévő adatot veszteség nélkül. A könnyű átalakíthatóság ugyanakkor azt is megengedi, hogy a saját igényeinknek megfelelően módosítsuk a formátumot, mivel a definíciója rugalmas, főként ajánlásokat tartalmaz (pl. adatként JSON vagy szabad szöveg), és a lehető legkevesebb megkötést.

```
form     lemma  xpostag
# Ez egy mondat eleji komment
A        a      [/Det|Art.Def]
kutyák   kutya  [/N][Pl][Nom]
ugatnak  ugat   [/V][Prs.NDef.3Pl]
.        .      [Punct]

A        a      [/Det|Art.Def]
...
```

1. ábra: Példa a formátumra. Egy három oszlopot tartalmazó fejléces TSV fájl: szóalak, szótő, egyértelműsített morfológiai elemzés.

Az új formátum (1. ábra) valójában egy fejléccel rendelkező TSV (*tab separated values*) fájl, azaz egy (akár táblázatkezelőbe is betölthető) táblázat sorokkal és oszlopokkal. A klasszikus vertikális formátumnak megfelelően egy sor egy tokent ad meg, az oszlopokban (mezőkben) pedig az adott tokenhez tartozó információk, annotációk kapnak helyet. Az egyszerű TSV-hez képest két kiegészítéssel éltünk. Egyrészt a CoNLL-U formátumhoz hasonlóan a mondathatárok üres sorokkal vannak jelölve. Másrészt lehetőség van arra, hogy az egyes mondatok előtt egy kettőskereszt (`#`) karakterrel kezdődő sorban megjegyzéseket toldjunk be, melyek változatlanul átmásolódnak a kimenetre. Bár a CONLL-U formátum miatt a mondateleji megjegyzést megengedtük, használata a kettőskereszt miatt ellenjavallott. Nagy korpuszban bármilyen karakter előfordulhat, ezért bármely (ritkának vélt) karakter speciális használata hosszú távon hibához vezet: a karakter eredeti korpuszbeli előfordulása és speciális használata összeütközésbe kerül. Ezek gyakran csak jóval később felismerhető, nem várt hibákat eredményeznek, valamint lassítják és korlátozzák a rendszer működését és későbbi bővíthetőségét – a speciális karakter ügyes megválasztásakor is.

Kiemelten fontos a fejléc szerepe, ugyanis ez az, ami a teljes rendszer működését meghatározza. A fejlécben szigorúan definiált oszlopnevek segítségével

[8] `http://universaldependencies.org/format.html`

azonosítják az egyes modulok a feldolgozáshoz szükséges bemeneti adatok helyét (függetlenül az oszlopok sorrendjétől!), és ugyanígy kimeneti adataikat szigorúan definiált nevű új oszlopokba helyezik el, az összes többi oszlopot változatlanul hagyva. Ennek a következménye az a kívánalom, hogy egy modul a bemeneti sorok számát ne változtassa meg. (Ha esetleg olyan modul készül a jövőben, mely megváltoztatja a sorok számát, akkor nagyon körültekintően gondoskodnia kell az új sorok mezőinek tartalmáról, az adatok teljeskörű integritásáról, beleértve a szekvenciális címkézés kezelését is.)

Az újonnan létrehozott oszlopok az ajánlásunk szerint, a jelenlegi implementációban egyszerűen mindig a meglévő oszlopok után kapnak helyet, de ez az oszlopok névvel való azonosításának köszönhetően nem kötelező. A szöveg így emberi szemnek is jól olvasható marad, lokálisan van tárolva az annotáció, valamint az opcionálisan elhelyezhető tetszőleges számú extra oszlop teret ad az igény szerinti bővítésnek és a kiegészítő információknak – akár nagyméretű fájlok esetén is. Az oszlopok elnevezése és tartalma az előállító és feldolgozó programok közötti megegyezésen múlik, és elengedhetetlen, hogy az egymásra épülő modulok között szinkronizálva legyen. A mezők tartalma ajánlottan szabad szöveg vagy a szabványos és kiforrott JSON formátum[9], mely lehetővé teszi kötött struktúrák átadását is, valamint használatával elkerülhető a házi formátumok és extremális karakterek használata.

4. Az architektúra

A TSV formátum egyszerűsége miatt jól kezelhető, számtalan eszköz támogatja, egyúttal megadja a szükséges szabadságot a későbbi modulok írásához is. Habár – a felhasználói igényeknek eleget téve – Python nyelven implementáltuk az egyes modulokat összekötő interfészeket, ezek a specifikáció alapján más programozási nyelveken is egyszerűen implementálhatók, akár egymástól függetlenül, heterogén összeállításban is. Elsődleges célunk volt, hogy megkönnyítsük a további modulok egyszerű fejlesztését és bekapcsolását a rendszerbe. Továbbá a hagyományos parancssoros (CLI) és a formátumfüggetlen Python könyvtár (library) interfész mellett egy programozási nyelvektől független, úgynevezett REST API-t is létrehoztunk.

A CLI a hagyományos unixos szerelőszalagok segítségével egy olyan jól használható eszközt ad a haladó nem technikai érdeklődésű és technikai érdeklődésű felhasználók kezébe egyaránt, amely akár nagy adatokon is használható a modulok belső működésének ismerete nélkül. A Python könyvtár a nagyobb programrendszerekbe történő könnyebb integrációt segíti az informatikus/nyelvtechnológus felhasználóknak. A REST API viszont sokkal inkább a modern felhős trendeknek megfelelően az akár teljesen laikus (nem nyelvtechnológus) felhasználók, illetve üzleti körök előtt nyitja meg a rendszer igénybevételének lehetőségét: segítségével telepítés nélkül, azonnal, a felhőben futtatva, jól

[9] Bár a JSON formátumnál a strukturáló elemek közötti térköz választható tabulátornak is, a TSV-be történő beillesztés miatt ezt nem ajánlott megtenni.

skálázható módon szolgáltatásként elérhetővé tehető a rendszer széles igényeket kielégítve, bármilyen programozási nyelven keresztül.

Az egyes modulokat az általunk megalkotott, a TSV mint kommunikációs formátum kezelését általánosan megvalósító, `xtsv` keretrendszer fogja össze. Ez teszi lehetővé a 3. fejezetben leírt formátumon keresztüli kommunikációt (a bemeneti oszlopok kiválasztását, a kimeneti oszlopok hozzáillesztését és az egyéb oszlopok megőrzését), a REST API-k létrehozását és a dinamikus formátumellenőrzést (5. fejezet) is, a modulok konkrét tartalmától függetlenül. Az egyes modulokat néhány deklaratív stílusban megadott paraméter révén illeszthetjük a rendszerbe: szükséges a modul funkcióját megvalósító programegység neve, a kimeneti és a bemeneti oszlopok nevei, valamint az esetleges modellek és egyéb paraméterek megadása. Különböző modellek dinamikus használatához egy adott modulból alternatív példányokat is létrehozhatunk ezen a módon. Az `xtsv` a fentiek szerint dinamikusan létrehozza és futtatja a kívánt láncot.

A fenti tulajdonságok (nyílt integráció, új modulok írásának lehetősége, szabványos TSV formátum, a háromféle API, kis erőforrásigény, jó skálázhatóság, szabályalapú és statisztikai rendszerek kombinálásának lehetősége) együttes fennállásával és a nyílt forráskóddal[10] a 2. fejezetben említett versenytársakhoz képest sokkal szélesebb lehetőségeket (csővezeték, programkönyvtár és REST API-n keresztül elérhető szolgáltatás, mely akár saját felhőben is futtatható, saját modulok vagy akár kézi javítás beillesztése a láncba, tanulmányozhatóság, módosíthatóság, újrataníthatóság, összehasonlíthatóság) biztosítunk a rendszer leendő felhasználói számára.

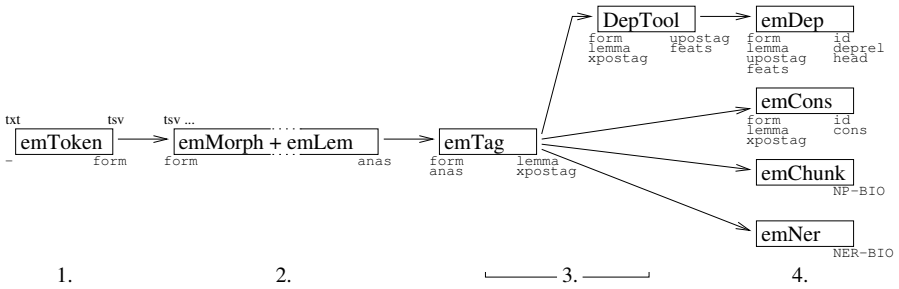
A következő fejezetben ismertetjük az egyes rendelkezésre álló modulok szerepét a láncban, valamint az új modulokkal szemben támasztott minimális elvárásokat, melyek lehetővé teszik a lánc szabad kiterjesztését új modulokkal és a meglévőknek az adott keretek közötti módosításával.

5. A modulok

A modulok láncban történő kezeléséhez szükséges, hogy a láncban előrébb levő modulok által felhasznált mezőket gyártó modulok kimenete elérhető legyen a következő modul futtatása előtt. Ennek ellenőrzését a fejléc révén már a szerelőszalag felépítésének idejében meg lehetett oldani, megfelelően korán jelezve az esetleges hibát, akár dinamikusan definiált szerelőszalag esetében is. Ezen megfontolásból úgy alkottuk meg a rendszert, hogy az egyes modulok definíciójának inherens része, hogy milyen oszlopokat igényel, és milyeneket állít elő. Ezenkívül törekedtünk arra, hogy a logikailag különválasztható funkciók külön modulba tartozzanak, akkor is, ha korábban egy modulban egybe voltak építve. Így az egyes modulok feladatköre pontosabban megragadható, és a tesztelésük és fejlesztésük is egyszerűsödik. Az `e-magyar` korábbi verziójából a meglévő modulok felhasználásával jelenleg a 2. ábrán látható láncot definiáljuk. Az `xtsv` általi egységes kezelhetőség miatt (lásd 4. fejezet), a Java-ban írt modulokat egy-egy

[10] A rendszer forráskódja a `https://github.com/dlt-rilmta/emtsv` címen érhető el LGPL 3.0 licenc alatt.

Python nyelvű modulba csomagoltuk, mely minden esetben önálló használatra – Python modulként – is alkalmassá teszi az adott programot. Ezen kiterjesztések nevei egységesen `py` végződést kaptak. A Python interfészekben a Java Pythonon keresztüli meghívásáért egységesen a PyJNIus nevű könyvtár[11] felelős. A kiterjesztett modulok Java natív típusokon keresztül kommunikálnak az eredeti modullal, kiiktatva az eredeti bemeneti és kimeneti formátumok különbségeit, melyeket a láncban az `xtsv` hivatott elfedni. A következőkben az egyes modulokat érintő fentieken túli változásokat ismertetjük.



2. ábra: Az `emtsv` jelenlegi feldolgozó lánca, a bemeneti és kimeneti mezőkkel. A definiált mezők alapján a lánc összeállításának idejében tudható például, hogy a POS taggeléshez kell a `form` és az `anas` oszlop (megfelelő formátumban), vagy hogy a dependenciaelemzést meg kell előznie a POS taggelésnek, a chunkolásnak viszont nem, ahogy ez az ábrán is látszik.

5.1. `emToken`

Az új eszközlánchoz – bár maguk a tokenizálási szabályok maradtak a régiek – jelentősen át kellett dolgoznunk a tokenizálót is. Az `emToken`-t [8] alkotó modulok eddig egy bináris fájlra fordultak, ami mindent tartalmazott, ami a futtatásához szükséges. Az új verzióban az egyes modulok külön-külön futtatható binárisokra fordulnak, ezek a szabványos bemenetről olvasnak, a szabványos kimenetre írnak, és egy Python program köti össze őket. Az új struktúrához át kellett írni az `emToken`-hez használt tesztelési rendszert is, ugyanakkor ez lehetővé tette a szerves integrációt az `emtsv` keretrendszerrel.

[11] `https://github.com/kivy/pyjnius`

5.2. emMorph és emLem

Az emMorph-ot, valamint az emMorph és emLem együttműködését érintő bizonyos hibákat javítottuk, mások megoldása folyamatban van. A morfológia belső formátumnak tekinthető kimenetét (a kétszintű morfológia felszíni alak–mély alak párjait) nem használjuk fel közvetlenül, hanem elemzéssel ellátott morfémasorozattá alakítjuk, valamint a morfémákból a lemmát is meghatározzuk. E két feladatot végzi az emLem modul, melyhez az eddigi Java implementáció helyett egy új, Pythonban írt változatot[12] készítettünk, amely egyszerűségre és a kód átláthatóságára törekszik.

A modult kiegészítettük egy speciális, saját REST API-val, melynek segítségével a felhasználó egy adott szó elemzéséhez egyszerűen a böngészőből, a szónak egy speciális URL-be történő beírása után férhet hozzá. A https://emmorph.herokuapp.com/dstem/terem címen a felhőben található demó segítségével bárki könnyen meg tudja nézni bármely magyar szó – esetünkben a *terem* – morfológiai elemzéseit az emMorph szerinti kódrendszerben, lemmával együtt.

```
{
  "terem": [
    {
      "lemma": "terem",
      "morphana": "terem[/N]=terem+[Nom]=",
      "readable": "terem[/N] + [Nom]",
      "tag": "[/N][Nom]",
      "twolevel": "t:t e:e r:r e:e m:m :[/N] :[Nom]"
    },
    ...
  ]
}
```

3. ábra: Példa a morfológiai elemző és lemmatizáló JSON kimenetére. A *terem* szó lehetséges elemzései: főnév, ige, valamint a *tér* birtokos személyragos alakja.

A modul kimenete mindkét esetben egy speciálisan formázott JSON (ld. 3. ábra), mely emberi és gépi felhasználásra egyaránt alkalmas. Minden elemzés négy mezőt tartalmaz, rendre: a token szótöve ("lemma"), a morfémákra bontott alak először géppel olvasható ("morphana"), majd ember által is olvasható formátumban ("readable"), a puszta címke morfológiai szegmentumok nélkül ("tag"), végül az emMorph kétszintes kimenete hibakeresés céljából ("twolevel"). A REST API egyszerre több szó elemzését is képes visszaadni, amennyiben HTTP POST metódussal hívjuk a dokumentációban megadott feltételeknek megfelelően. Az új JSON formátum előnye, hogy szabványossága és kiforrottsága révén véd a nagy korpuszokban előforduló, nem várt karakterek okozta hibáktól is. A TSV-be illeszthetőség kedvéért a JSON-ban tilos a sztringen kívüli tabulátor használata.

[12] https://github.com/ppke-nlpg/emmorphpy

5.3. emTag

A PurePOS [9] hagyományos, nem szabványos formátumához (ld. a PurePOS dokumentációjában[13]) képest a 3. fejezetben ismertetett új formátum nagy előrelépést jelent. A nagy korpuszokban előforduló, nem várt karakterek okozta hibák így kiküszöbölhetőkké váltak.

A fejlesztésünknek köszönhetően most már natív Java-adatszerkezetként is megadhatók az alternatív elemzések a Java nyelven írt PurePOS számára (akár már Java programból is), így függetlenítve azt az adat mindenkori formátumától. A PurePOS Python interfésze tartalmazza az emtsv-hez szükséges kiegészítéseket. A Python interfész segítségével a PurePOS használható önmagában előelemzett bemenettel, vagy csak a beépített statisztikai morfológiai elemzővel, illetve az emMorph+emLem szabályalapú morfológiát mintegy belső morfológiaként használva.

5.4. emChunk és emNER

A modulok alapjául szolgáló HunTag3 [10] konfigurációját átalakítottuk, hogy megfeleljen az e-magyar új formátuma által támasztott követelményeknek: az egyes jellemzőket mostantól nem oszlopsorszám, hanem név alapján éri el a program. Ezenkívül számos belső átalakításon esett át, melynek következtében a be- és kimeneti formátumok kezelése teljesen külön lett választva a program többi részétől, valamint egységesítve lett. Mivel a HunTag3 maga is Python nyelven íródott, ezért a különválasztott és átdolgozott bemenetiformátum-kezelés szolgált elsősorban az xtsv keretrendszer alapjául.

5.5. emMorph2UD

A Magyarlánc 3.0-ról leválasztottuk a DepTool modult, mely az emDep függőségi elemző számára konvertálja át az emMorph által kiadott és az emTag által egyértelműsített morfoszintaktikai információkat jegy–érték párok linearizált sorára. Az emDep eddigi modellje is olyan tanítóanyag alapján készült, amelyhez a DepTool konvertálta a szófajcímkéket és a morfoszintaktikai jegyeket. A DepTool-t közelebbről megvizsgálva azonban kiderült, hogy bizonyos morfológiai jegyeket nem kezel, a bemeneti emMorph címkék tartalma sok esetben elvész. A DepTool kimeneteként előálló címkék ráadásul olyan formátummal rendelkeznek, amely csak az e-magyar-on belül, két modul között hasznosítható. Ezzel szemben mi egy teljesebb konverziót szerettünk volna elérni egy olyan formátumra, amely a két modul közötti átjárhatóság mellett önálló kimeneti annotációként is használható.

Mivel az emDep modulhoz rendelkezésre állt egy másik modell, amely a Szeged Treebank UD morfológiai címkéivel ellátott tanítóanyag alapján készült, ezért úgy döntöttünk, hogy lecseréljük az emDep modelljét erre a verzióra. Az új konverter, az emMorph2UD az e-magyar elemzőlánc jelenlegi emtsv változatában egyrészt egy közbülső láncszemként az emMorph kimenetét konvertálja az emDep

[13] https://github.com/ppke-nlpg/purepos

modul számára fogyasztható jegy–érték struktúrájú UD címkékre, másrészt pedig kimeneti formalizmusként lehetővé teszi, hogy az `e-magyar` elemzőláncot használók az eddig elérhető `emMorph` kimenet mellett UD morfológiai címkéket is kaphassanak, amely egy nemzetközileg elterjedt, univerzális annotációs séma szabályait követi [11]. A konverter részletesebb ismertetését és kiértékelését lásd: [12].

5.6. `emDep` és `emCons`

A Magyarlánc 3.0-ról leválasztottuk a függőségi elemzést megvalósító Bohnet parsert [13] és az összetevős elemzést megvalósító Berkeley parsert [14], melyek így – megtisztítva azoktól a részektől, amelyeket a Magyarlánc használ, de az `e-magyar`-nak nem szükségesek – kisebb erőforráslábnyommal képesek működni. Az `emDep` modelljét lecseréltük (ld. 5.5. fejezet), így a modul bemenete a UD annotációs sémának megfelelő, az `emMorph` kimenetéből konvertált szófajcímke és morfológiai jegy–érték párok sorozata. A modulok kimenete, vagyis a szintaktikai annotáció nem változott.

6. Összefoglalás

A cikkben ismertettük az `e-magyar` nyelvfeldolgozó rendszer megújult és jelentős átalakításon átesett új verzióját, az `emtsv`-t. A rendszer immár nem csak felveszi a versenyt a szabadon elérhető versenytársaival, hanem több ponton meg is haladja azok képességét: szabványos kommunikációs formátumot használ, CLI, Python könyvtár és REST API hozzáféréssel bír, forráskódja elérhető, nyíltan integrált (*loosely coupled*), új modulokra nyitott (legyenek azok szabályalapúak vagy statisztikaiak), kis erőforrásigényű, és jól skálázható. Lehetőséget ad a REST API révén szolgáltatás üzemeltetésére, a CLI által csővezeték készítésére és a Python könyvtár API által nagyobb programrendszerekbe történő beépítésére, a nyílt forráskód miatt pedig mindez akár saját gépen, saját modulok fejlesztésével és beillesztésével is megvalósítható. A rendszer moduláris, továbbépíthető, összevethető, átírható, újrataníthatóó, tanulmányozható, módosítható. Ezzel a magyar nyelvre a funkciókban leggazdagabb, szabadon elérhető elemzőláncot hoztuk létre, mely leváltja a GATE-be integrált eredeti `e-magyar`-t.

Az új rendszerben rejlő valódi potenciál viszont csak akkor lesz teljes egészében kiaknázható, ha a modellek felépítéséhez használt, kézzel annotált korpuszok is szabadon elérhetőek lesznek, hogy az eszközök és az adat változtatásával, cseréjével mindenki szabadon kísérletezhessen új módszerek kifejlesztésével. További hiányzó elem egy olyan szabadon elérhető, kellően nagy méretű, magyar nyelvű korpusz, mely az `e-magyar` eszközkészlettel van leelemezve. Ez jó lehetőséget biztosíthat majd a rendszer részletes tesztelésére, vizsgálatára, a hibák feltárására, más kutatásokban való alkalmazására és végső soron a korpusz nyelvi adatainak elemzése révén új elméletek megszületésére.

Hivatkozások

1. Váradi, T., Simon, E., Sass, B., Gerőcs, M., Mittelholcz, I., Novák, A., Indig, B., Prószéky, G., Farkas, R., Vincze, V.: Az `e-magyar` digitális nyelvfeldolgozó rendszer. In Vincze, V., ed.: XIII. Magyar Számítógépes Nyelvészeti Konferencia. (2017) 49–60
2. Cunningham, H., Maynard, D., Bontcheva, K., Tablan, V., Aswani, N., Roberts, I., Gorrell, G., Funk, A., Roberts, A., Damljanovic, D., Heitz, T., Greenwood, M.A., Saggion, H., Petrak, J., Li, Y., Peters, W.: Text Processing with GATE (Version 6). GATE (April 15, 2011) (2011)
3. Zsibrita, J., Vincze, V., Farkas, R.: magyarlanc: A Toolkit for Morphological and Dependency Parsing of Hungarian. In: Proceedings of RANLP. (2013) 763–771
4. Novák, A., Siklósi, B., Oravecz, Cs.: A New Integrated Open-source Morphological Analyzer for Hungarian. In et al., N.C., ed.: Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC 2016), Paris, France, European Language Resources Association (ELRA) (2016)
5. Straka, M., Straková, J.: Tokenizing, POS Tagging, Lemmatizing and Parsing UD 2.0 with UDPipe. In: Proceedings of the CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies, Vancouver, Canada, Association for Computational Linguistics (2017) 88–99
6. Sass, B., Miháltz, M., Kundráth, P.: Az `e-magyar` rendszer GATE környezetbe integrált magyar szövegfeldolgozó eszközlánca. In Vincze, V., ed.: XIII. Magyar Számítógépes Nyelvészeti Konferencia. (2017) 79–90
7. Buchholz, S., Marsi, E.: CoNLL-X Shared Task on Multilingual Dependency Parsing. In: Proceedings of the Tenth Conference on Computational Natural Language Learning (CoNLL-X), New York City, Association for Computational Linguistics (2006) 149–164
8. Mittelholcz, I.: emToken: Unicode-képes tokenizáló magyar nyelvre. In Vincze, V., ed.: MSZNY 2017. (2017) 61–69
9. Orosz, Gy., Novák, A.: PurePos 2.0: a Hybrid Tool for Morphological Disambiguation. In: Proceedings of the International Conference Recent Advances in Natural Language Processing RANLP 2013, Hissar, Bulgaria, INCOMA Ltd. Shoumen, BULGARIA (2013) 539–545
10. Endrédy, I., Indig, B.: HunTag3: a General-purpose, Modular Sequential Tagger – Chunking Phrases in English and Maximal NPs and NER for Hungarian. In: 7th Language & Technology Conference, Human Language Technologies as a Challenge for Computer Science and Linguistics (LTC '15), Poznań, Poland, Poznań: Uniwersytet im. Adama Mickiewicza w Poznaniu (2015) 213–218
11. Vincze, V., Farkas, R., Simkó, K.I., Szántó, Zs., Varga, V.: Univerzális dependencia és morfológia magyar nyelvre. In Tanács, A., Viktor, V., Veronika, V., eds.: XII. Magyar Számítógépes Nyelvészeti Konferencia, Szeged, Szegedi Tudományegyetem Informatikai Tanszékcsoport (2016) 322–329
12. Vadász, N., Simon, E.: Konverterek magyar morfológiai címkekészletek között (2019) Jelen kötetben.
13. Bohnet, B., Nivre, J.: A Transition-based System for Joint Part-of-speech Tagging and Labeled Non-projective Dependency Parsing. In: Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning. EMNLP-CoNLL '12, Stroudsburg, PA, USA, Association for Computational Linguistics (2012) 1455–1465

14. Durrett, G., Klein, D.: Neural CRF Parsing. In: Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers), Association for Computational Linguistics (2015) 302–312