FELADATKIÍRÁS

A feladatkiírást a tanszéki adminisztrációban lehet átvenni, és a leadott munkába eredeti, tanszéki pecséttel ellátott és a tanszékvezető által aláírt lapot kell belefűzni (ezen oldal *helyett*, ez az oldal csak útmutatás). Az elektronikusan feltöltött dolgozatban már nem kell beleszerkeszteni ezt a feladatkiírást.

Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Szemantikai elemzés gráf-transzformációkkal

Bachelor's Thesis

Author
Ákos Holló-Szabó

Advisor
Evelin Ács
dr. Dávid Nemeskey

April 24, 2019

Contents

HALLGATÓI NYILATKOZAT

Alulírott *Holló-Szabó Ákos*, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2019. április 24.

Holló-Szabó Ákos
hallgató

Kivonat

Jelen dokumentum egy diplomaterv sablon, amely formai keretet ad a BME Villamosmérnöki és Informatikai Karán végző hallgatók által elkészítendő szakdolgozatnak és diplomatervnek. A sablon használata opcionális. Ez a sablon LaTeX alapú, a *TeXLive* TeX-implementációval és a PDF-LaTeX fordítóval működőképes.

Abstract

This document is a LaTeX-based skeleton for BSc/MSc theses of students at the Electrical Engineering and Informatics Faculty, Budapest University of Technology and Economics. The usage of this skeleton is optional. It has been tested with the *TeXLive* TeX implementation, and it requires the PDF-LaTeX compiler.

Chapter 1

Introduction

Manapság egyre több technológia jelenik meg, aminek alapja az NLP (Natural Language Processing). Ezek a technológiák nagy része nem lenne megvalósítható szemantikai elemzés nélkül. A szemantikai elemzés célja, hogy egy nyers szövegből, vagy beszédhangból előállítsa annak a szemantikai reprezentációját. Ez a reprezentáció egy irányított gráf is lehet, amit ha a mondat szintaktikai szerkezetét reprezentáló fákból állítunk elő, akkor a teljes feladat felfogható egy gráf-transzformációként.

Bár szemantikai elemzésre számos Deep Learning-es megoldás létezik, ezek pontatlansága nagy igényt teremt egy analitikus mély szemantikai elemzési módszerre. A gráf transzformációs megközelítés ígéretes eredményeket mutatott fel, mint például a Stanford Parser, ami TREGEX-ek segítségével végzi el tiszta analitikus módon a transzformációkat. Több formalizmus is létezik a transzformációk leírására, mint például a HRG (hyperedge-replacement grammar) vagy az IRTG(interpreted regular tree grammars). Jelenleg is egy ezekkel kapcsolatos kutatás folyik az AUT tanszéken.

A kutatás során az ALTO(Algebraic Language Toolkit)-val dolgoztunk, ami a jelenlegi leghatékonyabb környezet IRTG-k futtatására. Ugyanakkor a kutatásnak állandó gátját jelenti, hogy az IRTG még egy fejletlen nyelvtan és nehezen átlátható; és az ALTO-ból is hiányoznak fontos funkcionalitások. A problémán sokat enyhítene, ha az IRTG szabályokat REGEX-ek segítségével is meglehetne hivatkozni.

Szakdolgozatom keretében egy templatelésre alkalmas nyelvet fejlesztettem ki, ami a Slime fantázianévre hallgat. Segítségével az IRTG nyelvtanokat tömörebben és átláthatóbban lehet definiálni. Mivel az ALTO java-ban készül, a nyelv Kotlinban készül ANTLRv4 segítségével. Még nincs teljesen kifejlődve, de a feladathoz szükséges megoldásokat tartalmazza. Ilyen például a template definiálás, egymásba ágyazás, regexxel hivatkozás és sok egyéb. Teljes formájában egy univerzális bővítmény lesz, ami bármely nyelv vagy szöveg felett használható.

A dolgozat a következőképpen épül fel: Az 1. fejezetben bemutatom a . . . ., a 2. fejezetben a . . . .-ról írok, majd a 3. fejezetben a . . . ., végül . . . .

Chapter 2

Szemantikai elemzés gráf-transzformációkkal

2.1 Szintaxis

Egy nyelv jelentéstől független struktúráját nevezzük szintaxisnak. Szintaxis-ról beszélhetünk természetes nyelvek és programozási nyelvek esetén is. Ide tartozik, hogy a szöveg hogyan bomlik eltérő szerepű szavakra és jelekre [Nem kell, de miért jelennek meg a stanford fákban? a penn treebank egy korpusz, és valahogy kezelni akarták az írásjeleket, így döntöttek az annotátorok, de a nyelvészetben az írásjel nem a konstituens része, látni fogod majd, ha elolvasod az idevágó irodalmakat, amiket küldtem] és hogy ezek milyen hierarchikus rendszert alkotnak. A nyelvészetben a konstituens fa, avagy szintaktikai fa reprezentálja a szöveg szintaxisát, ami a tokeneket kifejezésekbe, majd mondatrészekbe és mondatokba rendezi. [hm, ez szerintem kissé pontatlan, megkérdezem a Gábort, hogy szerinte hogyan lehetne jobban] Az NLP-ben e helyett a függőségi gráfok terjedtek el, amik a szavak között címkézett irányított élekből építenek egy irányított gráfot. Ebben a fejezetben ezt a két formalizmust fejtem ki részletesebben.

2.1.1 A Szintaktikai Fa

A nyelvészet szemlélete szerint bármelyik emberi nyelv szintaxisa leírható egy Környezetfüggetlen Nyelvtannal.

Környezet Független Nyelvtannak egy $G = (N, \sum, P, S)$ rendezett négyest nevezünk, ahol:

- N nemterminális ábécé
- Σ terminális ábécé amire $N \cap \Sigma = \varnothing$
- $S \in N$ a kezdőszimbólum
- P pedig $\alpha \to \beta$ alakú átjárási szabályok véges halmaza, ahol $\alpha, \beta \in (N \cup \Sigma)*$ és α -ban van legalább egy nemterminális betű.

$https://people.inf.elte.hu/kubuaai/FoNYa/formalis_nyelvek_kidolgozott_tetelek.pdf$[másikforrást]
Ebben a nyelvtanban a terminális szimbólumok a szavak és az azokat tagoló írásjelek. A nemterminális szimbólumok három szintre sorolhatóak. A legalsó szint a szavak szintje, ahova a szófajokat és írásjel[kell-e?] [írásjel alapjáraton nem kell szerintem; a mi

nyelvtanunk a penn treebank sajátosságai miatt kezeli ugyan] kategóriákat jelölő csúcsok tartoznak. Ezek a csúcsok átírásával közvetlen a terminális szimbólumokat kapjuk. A terminális szimbólumokat eredményező átírási szabályokat "terminális szabály"-oknak hívják. A második szint a kifejezések szintje, ahova a szavak alkotta kifejezéseket jelölő (nagyjából 20 féle) csúcsok tartoznak . Ezek a csúcsok írhatóak át a típusuknak megfelelő headerű kifejezésekké, amik további kifejezésekből és szavakból állnak. A harmadik szint a mondatrészek szintje, amihez a mondat egészét jelölő 'S'(sentance) és a mondatrészeket jelölő négy további szimbólum tartoznak. A kezdő karakter az S, mint "sentence". Egy mondat akkor helyes szintaktikailag, ha létezik a környezetfüggetlen nyelvtanban levezetése. Ha létezik levezetése, akkor a levezetési fa maga a mondat Szintaktikai Fája.

Ezek például a 2.2.4-es pontban található szintaktikai fa szabályai

(a "John loves Mary." mondaté:

S3(NP1(NNP(John)), VP2(VBZ(loves), NP1(NNP(Mary))), .(.))) :

- $S \rightarrow NPVP$.
- $NP \rightarrow NNP$
- $VP \rightarrow VBZNP$
- $NNP \rightarrow John|Mary$
- $VBZ \rightarrow loves$
- . $\rightarrow$.

A szintaktikai fa a legbevettebb reprezentációja a mondatok szintaxisának a nyelvészetben. Analitikus megközelítés esetén is legtöbbször a szintaktikai fa az első, amit elkészítünk. Ehhez először a szavakat feltokenezzük, avagy meghatározzuk a terminális szimbólumok határait és kigeneráljuk a megfelelő "szófaj token"-t mindegyikhez. Végezetül megkeressük valószínűségi súlyozásokkal gyorsítva a legvalószínűbb levezetését az adott nyelv átírási szabályai alapján.

2.1.2 UD és a dependencia elemzés

A dependencia alapú megközelítés szerint a szintaktikai szerkezet lexikai elemekből áll, amiket bináris kapcsolatok kötnek össze. A dependencia fogalma azt jelenti, hogy a szavak(fej és dependens) irányított kapcsolatokkal vannak összekötve. Általában az állítmány a gráf gyökércsomópontja vagy a strukturális középpontja a mondatnak. A mondat szerkezetét a fejek és dependensek közötti viszonyok adják. Sok jól ismert elmélete van a függőségi nyelvtanoknak, összefoglalásért lásd (Nivre, 2005, p. 3).

Az UD(Universal Dependency) [UD weboldalát, (De Marneffe et al. (2014))] információt egy DAO gráfon szemlélteti, aminek csúcsai a szavak, és élei pedig a szavak közötti viszonyok. Az UD-hez hasonló Függőségi Gráfoknak sok féle formalizmusa létezik. Az NLP területén ezt a reprezentációt használják legtöbbször a szintaxis reprezentálására. Mi a kutatásunk során az UD-t használtuk.

Az UD projekt egy nyelvek közötti konzisztens annotációs rendszer és fa adatbázis hatvannál is több nyelvre. Kategóriák és annotációk univerzális készletét nyújtja miközben miközben megenged nyelvfüggő kiterjesztéseket is. A szavak közötti nyelvtani viszonyt szemlélteti, mint például az alany-állítmányi vagy tárgy-állítmányi vagy jelzői. Az UD a

Stanford Dependencies [hivatkozás: (De Marneffe and Manning (2008)] fejlődött ki, amit egyesítettek a Google univerzális címkékkel [Petrov et al. (2011)], az "Interset feature inventory"-nak egy átdolgozott rész halmazával [Zeman (2008)] és a CoNLL-X formátum egy átdolgozott verziójával [Buchholz and Marsi (2006)].

[példaelemzést LaTexben]

Az alap függőségek két csoportba sorolhatóak. Az egyik csoport a klauzális ("clausal") viszonyok, amik szintaktikai szerepeket írnak le a predikátumra vonatkozóan. A másik csoport a módosító viszonyok, amik azt írják le, hogy a fejet hogyan módosítja a függőben lévő szó. (Jurafsky and Martin (2018b)) Az egységes analízis okán a függőségek fejének a főnevet tekintjük, amit bemutat egy elöljáró vagy van hozzá csatlakozó. Más esetben a jelölőt tekintjük a függőség fejének. A formalizmus egy lexikalista megközelítést követ a számítás beli használat megvalósítására: a szintaktikai szerkezetek lexikai elemekből állnak, amik aszimmetrikus egy az egyhez kapcsolatokkal vannak összekötve a hatókörrel ellentétben, ami egy egy a többhöz kapcsolat. [De Marneffe et al. (2014)]Az UD lehetővé teszi dependencia parszerek nyelveken keresztüli kiértékelését. Több mint 30 csapat vett részt 2017-ben a közös céllal, hogy megvalósítsák a többnyelvű dependencia elemzést [shared taskos papert hivatkozd 2018].

szavak közötti viszonyok, angolul és jelölésük: [(Jurafsky and Martin, 2018b, p. 3)]

[itt meg lehetne olyan alfejezet, hogy "Szemantika" írhatnád benne, hogy mi az a szemantikai elemzés, hogy lehet csinálni (shallow vs. deep, vektorterek vs. gráfos cucc), de az ilyen átvezető részeket segítek majd megírni, vagyis átírok majd egyet-kettőt, ha megírtad, de olyasmik legyenek, mint a szakdolgozatomban amik vannak]

2.2 4lang

A 4lang [(Kornai et al. (2015))] egy formalizmus, ami irányított gráfokat épít szemantikai reprezentáció céljával. A gráfban a csúcsok nem szavakat, hanem nyelvfüggetlen fogalmakat jelölnek. Ezeknek a fogalmaknak már nincsenek nyelvtani jellemzőik és a kompetens beszélők közös tudását reprezentálja a fogalomról. Például a fagy[`fagy`] mit főnév vagy ige vagy fagyás vagy fagyott szava nincsenek megkülönböztetve a 4lang reprezentációban [itt hivatkozzuk, akinél ezt a példát először olvastuk] Ezáltal a 4lang fogalmak és a nekik egy nyelvben megfelelő szavak között egy a többhöz kapcsolat áll fenn.

A csúcsokat három féle él kapcsolhatja össze: 0-él reprezentálja a tulajdonságokat. Például $virág - 0 - > szép$, az $AZ\_EGY(IS\_A)$ viszony $virág - 0 - > növény$ és unáris predikció $virág - 0 - > bimbózás$. 1 és 2-élek bináris predikciókat kapcsolnak az argumentumaikhoz. Például $James < -1 - szeretet - 2 > kutya$. Binary (tranzitív) elemek, amik nem felelnek meg egyik szónak sem egy kifejezésben vagy mondatban, NAGY BETŰS nyomtatott nevekkel vannak jelölve.

Létezik egy másik él konfiguráció is, ami megjelenhet 4lang gráfban, $w_1 < -0 - 1 - > w_2$. Erre azért van szükség, hogy konzisztensen lehessen jelölni a tárgy és predikátum közötti viszonyt. Vegyük például a "I'am writing."("Én éppen írok") mondatot az $i - 0 > write$ 4lang gráffal és az "I'am writing a letter"("Én egy levelet írok éppen") mondatot az $i < -1 - write - 2 - > letter$ gráffal . Ez a két példa a dupla él nélkül azt jelentené, hogy a két az "i" és a "write" között attól függ a viszony, hogy adott e a tárgy vagy sem. [Recski (2018)]

A 4lang könyvtár tartalmaz eszközöket, amelyek képesek 4lang gráfokat építeni Nyers Szövegből és Szótári definíciókból (text_ to_ 4lang, dict_ to_ 4lang). A mag modulja a

4lang könyvtárnak, dep_ to_ 4lang, 4lang viszonyokat nyer ki szövegből, a Stanford Parser [DeMarneffe et al. (2006)] kimenetének a feldolgozásával és a Stanford függőségeknek a leképzésével 4lang részfáká. 4lang a neve egy kézzel alkotott fogalmi szótárnak is, ami négy nyelven tartalmazza nyelvfüggetlen fogalmak több mint 2000 definícióját (magyar, angol, latin, lengyel). [Kornai and Makrai (2013)] [ezt majd lehet hogy átrendezzük másik alpont alá, de amit írsz, az tök oké]

2.3 A 4lang és az UD különbségei

A munkánk során mint majd alaposabban is bemutatom a nyelvtan generálásánál, alaposan kihasználtuk a 4lang és az UD közötti hasonlóságot. Mindkét esetében egy irányított gráfról beszélünk. Mindkét esetében a csúcsok megfeleltethetőek a szavaknak, és az élek szavak közötti viszonyokat jelölik. Maguk a függőségek is sok esetben megfeleltethetőek egymásnak. Például a $w_1 - amod- > w_2$ viszony az UD-ból megfelel a $w_1 - 0- > w_2$ viszonynak a 4lang-ból.

A kettő között ugyanakkor fontos elméleti és jelölésbeli különbségek vannak. Az UD a mondatok szintaxisát reprezentálja, míg a 4lang a szemantikai jelentését. Az UD-ben a csúcsok maguk a szavak, míg 4lang-ban a szavaknak megfeleltethető fogalmak. Az UD élei a szavak közötti nyelvtani viszonyt jelölik, addig a 4lang élek a szavak közötti szemantikai függőségekket. A 4lang függőségek és az UD függőségek között egy a többhöz viszony áll fenn, ahogy a szavak, és a 4lang beli fogalmak között is. Ez persze azt jelenti, hogy az UD már tartalmazza a szemantikára vonatkozó információkat is, de csak közvetetten.

[UD-4lang megfeleltetéses táblázat]

Bár a mondat UD gráfjából a 4lang gráfja levezethető, a két formalizmus tartalma mégsem ekvivalens. A 4lang gráfból már nem mindig vezethető le egyértelműen az UD. Az UD szavairól pedig nem egyértelmű, hogy a 4lang mely fogalmaira vetítsük le őket. A több jelentésű szavak is nehezítik a helyzetet, mivel ekkor a szónak megfelelő fogalom már függ a szó környezetétől is. A projekt jelenlegi fázisában még nem foglalkoztunk a szavak és 4lang fogalmak megfeleltetésével. Még nem áll rendelkezésünkre erre automatikus módszer. Mivel egy nyelv szókészlete és szavainak jelentése is állandóan változik, teljsen elvetendő a a megfeleltetésekhez szükséges adatok manuális előállítása.

2.4 A kutatás célja

A kutatásunk célja az, hogy a nyers szöveg és a feljebb leírt interpretációk bármelyikéből le tudjuk a többit generálni. Ha az egyik interpretációhoz a másikból több is tartozik, akkor képesek akarunk lenni az összes verziót legenerálni és a legalkalmasabbat automatikusan kiválasztani valószínűségi súlyozás segítségével. Minél kevesebb nyelvet szeretnénk használni, és minél kevesebb módszertant. Ez utóbbi szemponttól azt várjuk, hogy könnyebb lesz a kódot karbantartani és bővíteni további interpretációkkal. Mind ezt hatékonyan is szeretnénk csinálni. E téren az alap célunk egy négyzetes lefutási idő a bemenet méretének függvényében. Ezeknek a megkötéseknek az IRTG nyelv eleget tesz. Hiszen nagyon eltérő interpretációk esetében is legfeljebb interpretációk száma plusz egy nyelvet kell használnunk. Ugyanakkor, ha kiegészítjük az IRTG-t egy összetettebb gráf algebrával, az képes lehet minden interpretáció hatékony leírására és mindehhez két nyelvet kellene használnia.

2.5 Az IRTG és az alárendelt algebrák

Ebben a fejezetben a kutatás során használt környezetről(ALTO), formalizmusról(IRTG), algebrákról(SA,TTA,SGA), hátrányaikról és az ezeket kiküszöbölő ideiglenes megoldásainkról lesz szó.

2.5.1 Az ALTO

A kutatás során a kódot az Algebraic Language Toolkit-tel, avagy ALTO-val fordítjuk és futtatjuk. Az ALTO egy nyílt forrású parszer, ami többféle algebrát is megvalósít, ami IRTG-be ágyazva használható. Ezek egyike az s-graph és a tag tree algebra. Ezen kívül is szabadon bővíthető új algebrákkal. Már korábban is használták gráf transzformációra és szemantikai feldolgozásra is. Nagy előnyt jelent, hogy Java-ban lett implementálva , így szinte bármely platformon futtatható. Ezen kívül rendelkezik grafikus és konzolos felhasználói felülettel is.

2.5.2 Az IRTG

Az IRTG (Interpreted Regular Tree Grammar, Interpretált Reguláris Fa Nyelvtan) egy kontextusfüggetlen nyelvtan, ami egy vagy több algebrába beágyazott újraíró szabályokból áll.

```
NP -> _NP2_amod_JJ_NN(JJ, NN)
[string] *(?1,?2)
[tree] NP2(?1, ?2)
[ud] merge(f_dep(merge("(r<root> :amod (d<dep>))", r_dep(?1))),?2)
[fourlang] merge(f_dep(merge("(r<root> :0 (d<dep>))", r_dep(?1))),?2)
```

1. ábra: Egy, a amod relációt leíró IRTG-szabály négy algebrával

A szabálysorok egy környezetfüggetlen nyelvtan átírási szabályait adják meg. A szabályok feldolgozásakor először egy levezetési fa (derivation tree) épül, amelyek a nonterminálisokat lecserélő szabályokat tartalmazzák. Egy szabályt a következő módon lehet definiálni (a sablonban a változó részeket {$ $}zárójellel jelöltem, ahol a $ a Slot kifejezésből ered, a {} pedig a nem szöveg elemeket jelöli):

```
[{$ interpretáció neve $}] {$ interpretáció lépése $}
```

A interpretációk nyelve és kimenete többféle is lehet. Választható például a szöveg kimenetű String Algebra, a csúcs sorrendet tartó fa gráf kimenetű Tag Tree Algebra, vagy az irányított gráf kimenetű S-graphAlgebra. Az algebrák és az interpretációk között egy a többhöz kapcsolat van. Az interpretációk egymástól teljesen függetlenek. Egy szabályban minden interpretációt meg kell adni. Minden átírás során, minden interpretációra vonatkozó derivációba beszúródnak az alkalmazott szabályban az interpretációhoz tartozó lépések. A beszúrás helyét ?*szám*-ként jelölik minden interpretációban. Itt a szám annak a jobb oldali nemterminálisnak a sorszáma, amely az interpretációhoz tartozó lépése szúródik be a jelölt helyre az átírása során. Az IRTG futtatásakor bármelyik interpretáció lehet a bemenet. Az ALTO a bemeneti interpretáció algebrájának megfelelő formátumú bemenetet vár. A futás során az ALTO keres a bemenethez egy olyan levezetést, ami megfelel a Környezet Független Nyelvtannak és a bemenetet adja eredményül.

Ezt követően a levezetési fa szerint felépíti a többi interpretációt is és végrehajtva őket előállítja a kimeneteket.

Tekintsük az 1.ábrán bemutatott szabályt. Ez a szabály egy Melléknévből (Adjective,JJ) és egy Főnévből(Noun,NN) készít egy két gyermekű Főnévi Kifejezést(Noun Phrase, NP). A két szó között Melléknévi Módosítói(Adjectival Modifier, amod) viszony áll fenn az UD gráfban. Ennek megfelelően neveztük el a szabályt "_ NP2_ amod_ JJ_ NN"-nek. Az első reprezentációk sorban a Nyers Szöveg, a Szintaktikai Fa, az UD Gráf és a 4lang Gráf. Itt minden reprezentációban a "?1" és "?2" az ahova a "JJ" és "NN" jobb oldali nemterminális szimbólumok átírásánál keletkező kifejezések kerülnek. Ezek az interpretációk bemenetei. Jelen esetben a "?1" a "JJ" és a "?2" az "NN" szimbólum interpretációinak a helye. A két jobb oldali Nemterminális kiértékelése után minden interpretációba a vele azonos interpretáció kimenete kerül, string-é a stringbe stb..

A string interpretációhoz a String Algebra tartozik. Jelen esetben a bemeneti két szót fűzi össze. A tree interpretációhoz a Tag Tree Algebra tartozik, ami egy "NP2" címkéjű csúcs alá szúrja be a két bemenetet. Ez az algebra állítja elő a Szintaktikai Fát. A bemenete két-két csúcs. A negyedik és ötödik sorhoz is az S-Graph Algebra tartozik. Mindkettő irányított éllel köti össze a két bemenetet. Az UD egy amod, a 4lang pedig egy 0 címkéjű éllel. Itt mindkét esetben mindkét bemenet csak egy címkézett csúcs. A nevüknek megfelelő gráfokat állítják elő.. Magasabb szintű szabályoknál már a ud és 4lang bemenetek összetett irányított gráfok lesznek. Mi elsősorban az előbb említett három algebrát használjuk, de ezeken kívül más algebra is használható az IRTG nyelvben, mint például... [alto wiki]. Ezek nem részei a dolgozat fókuszának.

2.5.3 Az SA

Az SA(String Algebra) kétségtelenül az összes IRTG alatt elérhető algebra közül a legegyszerűbb. Itt csak szövegek konkatenációjára(összefűzésére) van lehetőség. A bemenet és kimenet nem tartalmaz slotokat vagy egyéb nyelvi elemeket. A műveletet a következő formátumban lehet megadni:

```
*( {$  szöveg1 $}, {$  szöveg2 $} )
```

Egymásba is ágyazható több konkatenáció:

```
*( {$  szöveg1 $}, *( {$  szöveg2 $}, {$  szöveg3 $} ) )
```

Például a *("Every mouse", *("loves", "cheese.")) kifejezés az "Every mouse loves cheese.", avagy "Minden egér szereti a sajtot." szöveget adja vissza. Ez a konkatenáció kommutatív és asszociatív.

2.5.4 A TTA

A TTA(Tag Tree Algebra) nak, mint az SA-nak, csak egy művelete van, az merge(egyesítés). A TTA esetében viszont már a bemenetek nem nyers szövegek, hanem csúcs sorrend tartó fa gráfok. Tartalmazhatnak slot-okat, amiket TTA esetében Hole-nak(lyuk) nevezünk. A Hole-okat '*'-al jelöljük. Nem lehet őket címkékkel vagy más módon megkülönböztetni sem eltörölni. A gráf csúcsaiba helyezhetőek. Az ilyen slot-okat tartalmazó fákat nevezzük Tag Tree-nek. A TTA merge műveletének két operandusa van. Mindkét operandus egy Tag Tree. A jobb oldali fát szúrjuk be a bal oldali fa minden

hole-jába a merge során. A merge jele a '@' szimbólum, és se nem kommutatív, se nem asszociatív. A fákat zárójelekkel adjuk meg. Egy csúcs közvetlen gyermekeit és az azok alatti részfát a tőle jobb oldali zárójelben kell megadni vesszőkkel elválasztva. Például így írható le jól bevált "John loves Mary." ("János szereti Marit.") mondat Szintaktikai Fája:

S3(NP1(NNP(John)), VP2(VBZ(loves), NP1(NNP(Mary))), .(.))

, aminek ez a fa felel meg:

Ugyanez az igés kifejezés, avagy a VP(Verbal Phrase) helyén hole-lal:

S3(NP1(NNP(John)), *, .(.))

A fenti gráfba a VP2 beszúrása merge-dzsel:

@(S3(NP1(NNP(John)), *, .(.)), VP2(VBZ(loves), NP1(NNP(Mary))))

Ennek a műveleti fája:

A TTA egy egyszerű, de átlátható és jól kezelhető nyelv. Ugyanakkor nincs arra lehetőség, hogy a jobb oldali gráfot a baloldali gráfnak csak adott csúcsába szúrjuk be. A bal oldali gráf minden lyukát felhasználjuk a merge művelet során. Ez már a három gyermekű csúcsok esetében is komoly nehézséget jelentett számunkra. Négy gyermekű csúcsok esetében egyenesen ellehetetleníti a két bemenetű szabályok használatát, amikre a hatékonyság végett törekszünk.

Például ha van már egy szavak nélküli:

S3(NP1(*), VP2(*, NP1(*)), *)

fánk, akkor abból sose leszünk képesek az eredeti:

S3(NP1(NNP(John)), VP2(VBZ(loves), NP1(NNP(Mary))), .(.))

fát előállítani, mert már az első szóhoz tartozó csúcsok, az "NNP(John)" beszúrása esetén az:

S3(NP1(NNP(John)), VP2(NNP(John), NP1(NNP(John))), NNP(John))

fa gráfot kapjuk. Éppen ezért a interpretációs lépések segítségével kell összerakni szabályról szabályra ezt a fát. Lásd [Példa 1].

Ez a példa is jól mutatja, hogy egy két gyerekű csúcsot, mint a VP2 össze tudunk rakni egy két bemenetű szabályban. Egy három gyerekűt, mint az S3 már nem, hiszen a három gyereket nem tudja mind megkapni egyszerre. Ekkor kénytelenek vagyunk két szabály alatt előállítani a szerkezetet merge használatával. Az ilyen három gyermekű csúcsok gyakoriak a Penn Treebank Szintaktikai Fáiban. Ilyen a "the black cat" ("a fekete macska") Főnévi Kifejezés is, aminek a Szintaktikai Fája az:

NP3(DT(the), JJ(black), NN(cat)).

A fát Merge nélkül praktikusan csak három bemenetű szabállyal lehetséges implementálni. Lásd [Példa 2].

Merge segítségével sokkal optimálisabban is megoldható. Lásd [Példa 3].

Négy gyermekű főneves kifejezések esetében ez már nem lehetséges. Ilyen például a "this British industrial conglomerate"(ez a brit ipari összetömörülés), amihez a:

NP4(DT(this), JJ(British), JJ(industrial), NN(conglomerate))

fa tartozik. Ezt a fenti logikával nem tudjuk helyesen megoldani. Lásd [Példa 4].

Itt a második

NP_BAR

elkészítésekor, a merge művelet során a JJ mindkét lyukba beszúródik, így a: NP4(DT(this), JJ(British), JJ(British), NN(conglomerate))

Ezt a TTA korlátai miatt nem tudjuk megkerülni.

Négy gyermekű csúcs egy helyes szintaktikai fában nem fordul elő. A középső JJ-k helyes esetben egy ADJP-t alkotnának. Mi a Stanford parser kimenete alapján dolgoztunk, ami sok esetben ilyen fát adott eredményül.

2.5.5 Az SGA

Az SGA(S-Graph Algebra) a legbonyolultabb algebra, amit használunk. Több műveletet és összetett gráf nyelvet használ. Az egyetlen hiányossága, hogy nem képes a csúcsok sorrendjét kezelni. Ezért szorulunk a TTA használatára csúcs sorrend tartó fák esetén. Az SGA-ban egy csúcsnak három attribútuma van, name(név), tag(címke) és mark(megjelölés). Ezeket a következő szintaxissal tudjuk megadni: {$ name $}/{$ tag $}<{$ mark $}>. Az attribútumok közül a tag és a mark elhagyható. A név azonosítja a csúcsot adott környezetben, a címke jelenik meg a gráf kirajzolásakor és a jelöléssel hivatkozhatunk a csúcsokra egyes műveletek során. Az éleket :-tal jelölik, és címkézhetőek. Alapesetben az él balról jobbra mutat, de van mód jobbról balra mutató él definiálására is. A gráfokat string ként kell megadni Például a:

“(ROOT :root (loves/loves :nsubj (John/John) :dobj (Mary/Mary)))”

a “John Loves Mary.” mondat UD Gráfját írja le, ami így néz ki:

A nyelvtan legfontosabb három művelete a forget(elfelejt), rename(átnevez) és a merge(egyesít). A forget és a rename a jelölések manipulációjára való. A forget művelettel lehet egy jelölést az összes vele megjelölt csúcsról törölni. A rename művelettel egy adott jelölés minden megjelölt csúcson lecserélhető egy másik jelölésre. A merge művelet bemenete az előzőekkel ellentétben két gráf. A bal oldali gráf minden megjelölt csúcsába beszúrja a jobb oldali gráf minden ugyanazzal a jelöléssel megjelölt csúcsát. A műveletek egymásba ágyazhatóak. Csak az egymást követő forget műveletek cserélhetőek fel minden esetben. Forget-et a “f_ {$ jelölés_ neve $}({$ gráf $})” formában lehet megadni, ahol természetesen a gráf helyén állhat újabb művelet is. Rename-t pedig a:

```
r_{$ régi_jelölés $}_{$ új_jelölés $}( {$ gráf $} )
```

formátumban lehet megadni. A Rename esetén a leváltandó jelölés neve elhagyható és abban az esetben a “root” jelölésű csúcsokon fogja végrehajtani. A merge formátuma hasonlít leginkább a közismert programozási nyelvek függvényhívásaira.: merge({$ gráf1 $}, {$ gráf2 $}) Lásd [Példa 5].

A példa sok szabálya megfeleltethető a tree reprezentáció ese tében használttal. Ennek oka az, hogy a nyelvtanokat úgy írtuk meg, hogy azokat könnyű legyen összeilleszteni. Általában a Szintaktikai fa egy csúcsát vagy egy három gyermekű csúcsának a felét rakjuk össze egy szabály alatt, és az UD Gráf-ba kerülő élek és a jobb oldali nemterminális szimbólumok szerint nevezzük el a szabályokat. Mivel az UD Gráfban és 4lang Gráfban sok a hasonlóság, azt is hozzá adom az egyesített nyelvtan példában. Lásd [Példa 6]

Az S-Graph algebra egy jól használható nyelv, de módosítani és javítani is nehéz. Ennek elsődleges oka a bonyolult szintaxis.

A műveletek nehezen átláthatóak. A legtöbb művelet jele egy-egy betű, és így se nem olyan beszédes, mint az aritmetikai operátorok, se nem olyan felismerhető, mint az SGA merge vagy a TTA '@' jelölése. Az operandusok egy részét is a függvény nevében jelölik. Ez mind sokat ront az átláthatóságon. Sok esetben fölösleges, hogy külön művelet a forget és a rename a merge-től. Egyszerűbb lenne a merge attribútuma ként megadni, hogy a bal oldali gráfból milyen jelölést párosítunk a jobb oldali gráf melyik jelölésével. Azt is lehetne opcionális operandus, hogy a merge után melyik mark legyen elfelejtve.

Ha már a nyelv célja egyértelműen a tömörség, akkor a merge lehetne '@ ' a TTA-hoz hasonlóan. A forget és a rename is lehetne pl. '&', mint reference, '#', '%' vagy egyszerűen egy 'R', ahol a forget egyszerűen egy olyan rename, ahol az új jelölés hiányzik és nem a régi.

A ":" semmilyen asszociatív viszonyban nincsen az irányított élekkel. A ':'-ot lehetne például "-{$ él_ címke $}-"re cserélni irányfüggően kacsacsőrökkel a végén. A csúcsok deklarációjában az attribútumok egymástól teljesen eltérő szintaxissal vannak jelölve teljesen feleslegesen. A csúcsok adatai is mehetnének egy szögletes zárójelbe. A zárójelen belül [n={$ name $} , t={$ tag $} , m={$ mark $}]formában egyértelmű is lenne, hogy melyik melyik. Így a sorrendjük lehet cserélhető. Így bármelyikük elhagyható anélkül, hogy zavaróvá válna. Itt attribútumok felsorolásáról van szó, így a kis betű sem zavaró. Adott sorrend mellett a kis betűk is elhagyhatóak:

```
[{$ name $} | {$ tag $} | {$ mark $}].
```

Lásd [Példa 7]

Véleményem szerint ez sokkal tisztább módja lenne és tördelni is sokkal egyszerűbb. Ha az egyik forget vagy rename műveletetre nincs szükség, akkor csak elhagyjuk a hozzá tartozó attribútumokat . Még ennél is szebb lenne, ha a műveleteket operációs jelek jelölnék. Lásd [Példa 8]

Ugyanakkor ez már túlzottan eltér a gráfleíró nyelvek hagyományos stílusától, és tördelni is nehézkesebb. Operátoros esetben pedig szükség lenne egy sorrendiségre is, ami nem lenne mindenki számára triviális. Arról nem is beszélve, hogy a rename művelet három bemenetű, így nem lehet egy operátorral elvégezni, ahogy az összetett merge négy operátorát is erőltetetten hat.

2.6 Az IRTG és az ALTO hiányosságai

[Ide is kéne egy bevezető]

2.6.1 Az IRTG hiányosságai

Az IRTG nem egy programozási nyelv, hanem egy nyelvtan. A szabályok egymás alatt adhatóak meg és csak backspace karakterekkel és kommentekkel tagolhatóak. Egy nyelvtan több fájlra nem szedhető szét. A fájl elején fel kell sorolni az interpretációkat, és egyik interpretáció sem hagyható el egyik szabályból sem, még akkor sem, ha semmit sem adnak vissza. A kutatás során egy olyan nyelvtant készítettünk, ami 2700 szabályból áll és négy interpretációból. Ez ha a szabályokat üres sorokkal választjuk el, akkor 16204 sor kódot jelent egyetlen fájlban. Egy ekkora kódot nehéz karbantartani, átlátni, fejleszteni. Tehát szükség van IRTG-ben az importálásra

Rengeteg esetben a szabályok alig egy-egy szóban térnek el. Lásd [Példa 6.] Itt a “John” és a “Mary” szóhoz tartozó terminális szabályok között csupán maguk a szavak jelentik a különbséget. A String interpretációban is csak maga a szó jelenik meg. Más szófajú szavak esetében is csak a szabály sort és tree reprezentációt kell még módosítani. Az ilyen esetek miatt nagy kár, hogy az IRTG-ben nem lehet szabályokat egymásból vagy egy közös ősből származtatni.

Sokszor egy szó szófaja egy másik szófaj alkategóriája. Ilyen például az NNP(), ami az N() alketegórája. Az egy főkategóriába tartozó szófajok sok esetben ugyanúgy viselkednek. Például egy NP-ben, ami egy N-ből és egy JJ-ből áll, minden esetben a két szó között egy amod él lesz az UD gráfban. Ugyanakkor vannak olyan esetek is, amikor az alkategóriáknak csak egy részhalmaza viselkedik hasonlóan. Egy alkategória esetében két féle képpen lehet megoldani ezt. Az egyik verzióban általánosítunk, avagy az alkategóriákat ugyanúgy kezeljük. Ekkor jelentős áldozatot hozunk a pontosság terén. Más esetben minden egyes alfajra előállítunk minden szabályt, amivel előfordul. Ekkor jelentős áldozatot hozunk hatékonyság terén, mivel sokkal több szabályon kell végig iterálnia az ALTO-nak az IRTG futtatásakor. Erre jó megoldás lenne, ha egy szabálysorban minden olyan nemterminális szimbólumot meg tudnánk adni, amire a szabálynak minden interpretációhoz ugyanaz a lépés tartozik. Erre jó megoldás lenne, ha a szabálysorban regex-xel lehetne megadni mindegyik jobb oldali nemterminális szimbólumot.

Az IRTG szabályokban mindenre más és más algebrát használunk más és más nyelvvel. Ennek az az oka, hogy mint már az algebráknál részleteztem, minden algebrának vannak hiányosságai. Az IRTG maga lehet önmagában átlátható, de ha az algebrái sokfélék, és némelyiket önmagában is nehéz átlátni, akkor az az IRTG-re is közvetlen hatással lesz (lásd s-graph algebra). Hiába a sokféleség, ha egyes feladatokra csak egy olyan algebra létezik, ami megfelel a célnak és az is erősen korlátozott. Egy algebrára van csak szükség. Egy olyan algebrára, ami az s-graph algebra szintjén van, de átlátható és képes mint a csúcsok sorrendjét kezelni, mint a címkék konkatenációját. Elvégre a szöveg is kezelhető egyetlen csúcs ként. Egy ilyen algebrában az is fontos, hogy ne kelljen az egyszerű műveleteket sokkal terjedelmesebben megoldani, mint ha azt az egyszerű algebrákkal végeznénk. Az s-graph javítására az algebra kapcsán mutattam példát, de egy ilyen univerzális algebra már szemantikai változtatásokat is igényelne.

Gyakori az IRTG-ben az, hogy kulcsszavak ütköznek pár változónak a nevével, ami felesleges errort okoz a feldolgozás során. Például az “interpretation” szóhoz generált terminális szabályunk ütközött az interpretation szóval, amivel kezdődnek az IRTG fájlok elején az interpretációkat definiáló sorok. Ha a szabálynak így kezdődik a neve, akkor már nem fogadja el a futtató környezet. Erre a legegyszerűbb megoldás az, ha a nyelvben megjelenik a szöktetés, vagy a kulcsszavakat úgy módosítjuk, hogy ritka speciális karaktereket is tartalmazzanak. Például, ha az “interpretation” helyett “<interpretation>” lenne a kulcsszó.

2.6.2 Az ALTO hiányosságai

Az ALTO működése iteratív. Végig iterál az összes lehetséges levezetési fán a környezetfüggetlen nyelvtanban, és kiválogatja a bemenetre illeszkedőeket. Ez alatt persze különböző módszerekkel igyekszik elkerülni azokat a lehetséges levezetéseket, amiről már tudja, hogy biztos rosszak lesznek. Erre az egyik legegyszerűbb módszer az, ha a levezetési fákat fentről lefelé generáljuk és kihagyjuk azokat a lehetőségeket, amik már magasabb szinten olyan élet vagy csúcsot vagy azok olyan halmazát vesznek be a derivációba, ami a bemenetben nem szerepel. [Ennek neve is van, erről kaptam forrást is.]

Az elvi működés és a megvalósítás következtében az ALTO rendkívül lassú. A feljebb említett 2700 szabályos és négy interpretációs nyelvtan 10^5 nagyságrendű adaton nagyjából 30 óráig futott, pedig a rendelkezésünkre álló tanszéki szerveren futtattuk. Ez a mi feladatunk esetében elfogadhatatlan. Ezért nem ártana egy hatékonyabb c++ verzió az ilyen méretű munkákhoz. Ma már a C++17 és a hamarosan érkező C++20 is kényelmes programozási nyelv, és nem nagy ördöngösség az alap platformokon futtatható verziókat legenerálni a nyílt forráskódból. Egy könyvtár formájában pythonból is elérhető lenne ez a verzió.

Az ALTO rengeteg felesleges munkát végez, mivel azokat a megoldásokat is ki generálja a bemenetből, ami más interpretációk derivációjában errorhoz vezet. Az ilyen eseteket is ki lehetne szűrni már a derivációs fa magasabb szintjének az iterációja során. Persze ez debuggolás szempontjából káros, mivel az ilyen esetek kiszámítása nélkül nem tudjuk megfejteni, hogy mi okozta a hibát, de mérvadó méretű adatot nem is akkor fogunk futtatni. Jó lenne, ha ez opció ként lenne elérhető.

Az ALTO mindennek ellenére a jelenlegi legfejlettebb program az IRTG futtatására, és állandó fejlesztés alatt áll. A pontos működését még nem ismerjük. A módosítások többsége viszont a belső működés alapos átdolgozását igényli. Erre jelenleg még nincs erőforrásunk.

2.7 Ideiglenes megoldások

A kutatás során az IRTG-t illető problémák többségét kód generálás segítségével küszöböltük ki. Két implementáció készült python nyelven, amik bemeneti adatokból generáltak irtg szabályokat. Kezdetben csak NP-k feldolgozása volt a cél, majd onnan terjesztettük ki az implementációkat VP-k és ADJP-k re. A generátor kódos megoldások legfőbb hátránya a fejlesztőhöz kötöttség volt. Nem tudtunk többen párhuzamosan fejleszteni, mert egyikünk sem értette a másik megoldását kellő mélységben. Mindkét megoldás a Stanford Parser a Penn Tree Bank ből kapott kimenetét használta fel bemenetnek, amiket python scriptekkel készítettünk elő . Ezek a scriptek végezték a fákból a megfelelő kifejezések részfáinak a kiszedését, a részfák rendezését típus szerint és a fák UD gráfjainak a generálását a Stanford Parser-rel.

2.7.1 első generátor

Az első generátort én készítettem el 2018 augusztusában. Ez egy objektum orientált megoldás volt. Képes volt kommenteket is generálni, kezelni a rövidítéseket és könnyen bővíthető volt. Ugyanakkor rush development során keletkezett a szakmai gyakorlatom végén, így a kód még nincsen se letisztázva, se alaposan dokumentálva.

Külön osztályban tárolta a nyers szöveg szavait , a fát és az UD-t, majd ezeket rendezte egy közös osztályba. A közös osztály a Data, a nyers szövegé a Terminal, fáé a Tree, a UD-é és a 4lang-é a Dependency. Az összes osztály rendelkezett saját szöveg bemenetű konstruktorral, ami közvetlen a bemeneti adatok formátumából inicializálja az osztájt.

A Terminal egy szót tárolt és annak a szófaját. A Tree tartalmazta a részfát a Stanford parser formátumában, a TTA formátumában, a TTA formátumában levelek nélkül és külön a szavakat. A Tree legtöbb formátuma csak a kommentek generálásához volt hasznos, de ideiglenesen mindegyiket generálta a program a kommentek végleges formátumától függetlenül. A Dependency tárolta a fej szavat, a függő szavat, az él UD típusát és az él 4lang típusát. Az UD él 4lang-ra vetítését a Fourlang enum és egy dictionary segítségével végezte. Mint korábban említettem, a szavak és 4lang fogalmak megfeleltetésével

még mai napig sem foglalkozunk, de az is ebbe az osztályba került volna és feltételezhetően szintén egy dictionary és enum segítségével.

A generálás innentől szöveg konkatenációból áll az adatok függvényében. A kifejezés generátor függvények először betöltötték az adatokat a Data osztályokba, majd kiszűrte az egyedi adatokat a levél nélküli fák és UD élek alapján. Az komment első sorát a szerint generálta, hogy van e kapcsolat a szavak között és hogy a fej vagy a függő van előrébb. Ez a sor azt írta le, hogy milyen kifejezést milyen típusú szavakból épít a szabály és, hogy a dependens milyen szerepet lát el a mondatban. Ehhez az adatok rövidítéseit a Short2Long osztály segítségével oldotta fel, aminek a betöltendő adatai a viszonylag kevés féle szófaj és él típus okán kézzel készültek. A következő két komment az első példány TT-jét és Stanford tree-jét tartalmazta. A szabály sort az adatokból nem volt nehéz legenerálni, mivel a szabályok neveit eredetileg is a bemenetek TT interpretációjának gyökere és az UD élek alapján képeztük. A string és tree sor csak a szabály bal oldali nemterminális szimbólumától függ, így azok konstans szövegek voltak. Az UD-nél a szövegbe csak az él nevét kellett beszúrni attól függően más sablonba, hogy a csúcsok között van e kapcsolat, megjelenik e mindkettő az UD gráfban(a “the”,”a” és “an” szavak például nem), és hogy ha van köztük kapcsolat, akkor melyik a fej. A fourlangnak ezek a tulajdonságok már bele voltak kódolva az enum alapú típusába, így itt csak egy else-if ágazaton kellett végigmenni (mivel pythonban nincsen switch case).

Már itt is látszott, hogy számos lehetőség van optimalizációra. Például a kódba ágyazott adatokat is ki lehetett volna szervezni txt fájlokba. Így a fourlang if-else ágai is megoldhatóak lettek volna egy dictionary-vel és karbantarthatóbbak lettek volna az adatok. A kettő magas kifejezések részfái között rengeteg volt a hasonlóság, így könnyedén össze lehetett volna a generáló függvényeiket vonni. A magasabb kifejezés részfákat se lett volna sokkal nehezebb ugyanabban a függvényben generálni. Az adatok feldolgozását és tárolását egy osztályban is meg lehetett volna oldani, ha a bemeneti adatok egy közös fájlban vannak. A nagyon sokadlagos adatokat, mint a fák stanford és 4lang formátuma, nem kellett volna eltárolni. Elég lett volna, ha a többi adatból származtató függvényeket is tartalmazza az egy darab Data osztály. Ezzel azt a duplikációt is elkerülhettük volna hátrány nélkül, hogy a terminálisokat majdnem mindegyik adatstruktúra eltárolta.

Végeredményben viszont még ekkor is egy olyan programunk lett volna, ami az IRTG kódot csak generálja. Végső soron templatelést használt volna, de az IRTG nyelvhez kötött szinten. Az új szabály egyre több kódot jelentett volna, és nehéz lett volna a hibákat meghatározni bárkinek, aki nem dolgozott a program elkészítésén és nem tanulmányozta hosszabb távon.

2.7.2 második generátor

A második generátort Ács Evelin kezdte el fejleszti 2018 őszén, mivel bár értette az első generátor működését, nem látta át a teljes kódot, és én már nem dolgoztam a projektben, hogy letisztázzam és alaposan ledokumentáljam azt. A generátor vívmánya, hogy az adatokat struktúra ként kezeli.

A bemenet ennél a generátornál a Stanford fák és a hozzájuk tartozó UD gráfok, amiket fa ként és gráf ként kezel. A többi adat többségét, mint a templateket és UD-4lang párokat, külső fájlokban tárolja. Ennek a generátornak is a template-lés az alapelve, ugyanakkor egy kevésbé objektum orientált megközelítést követ. Itt egy szabály adatai egy Dictionary-ben vannak tárolva, amiket egy Dictionary-be gyűjt, amiben a szabály neve a kulcs.

A terminálisokat egy külön script generálja. Külön kezeli az unáris, bináris és ternáris szabályokat.

Az unáris szabályokhoz egyszerűen végig iterál a fákon és kikeresi az egy gyermekű csúcsokat, amiknek a gyermeke nem levél. Kigyűjti a csúcs és a gyermeke címkéjét, és beilleszti egy ilyen sablonba:

```
{$ szülő címke $} -> _{$ szülő címke$}_unary_{$ gyerek címke $} ({$ gyerek címke $} )
[string] ?1
[tree] {$ szülő címke $}(?1)
[ud] ?1
[fourlang] ?1
```

A binárisokat és trenárisokat egy algoritmus kezeli. Ez az algoritmus először az UD gráf élein iterál végig. Veszi minden élet, a fejét és a dependensét. A két csúcs címkéjéhez tartozó szavakat és azok közös ősét kikeresi a szintaktikai fából. Ez után összegyűjti belőle a következő adatokat:

- *ős :*Mi az ős címkéje?
- *darab :*Hány gyermekű az ős?
- *gyermek1, gyermek2(, gyermek3):*Az ős-nek mik a gyermekei?
- *ős1, ős2:*Az ős melyik gyermekeinek leszármazottja a két szó?
- *szó1,szó2 :*Mi a két szó?
- *sorrend:*Mi a sorrendjük a fában?
- *ud:* mi az UD él címkéje?
- *4lang:*Mi az UD-nek megfelelő 4lang struktúra? (ezt az UD adatokból származtatja)

Bizonyos esetekben, mint egy case struktúra, a 4lang szabály egy harmadik csúcstól is függhet. Case esetében ezt kihagyjuk a 4lang-ból és egy HAS csúccsal helyettesítünk.

Az "ősgyermek"-ekre csak a trenáris szabályok generálása esetén van szükség és a szabály sorhoz. Az is fontos merge szabályok esetében, hogy az ős gyermekek szomszédosak-e. Mivel a két bemenetű szabályokra törekszünk, a trenáris szabályok két szabályból állnak, egy BAR és egy MERGE szabályból. A BAR generál egy TT-t két részfát és egy lyukat egy megfelelő címkéjű csúcs alá rendezve. A MERGE szabályok merge-lik a harmadik részfát ebbe a fába. Ezért a két ősgyermek pozíciójától függ, hogy a generált szabály egy BAR vagy MERGE szabály.

A szükséges adatokat ebbe a templatbe illesztjük a fejléc sorok generálásához:

```
{$ős$} -> _{$ős$}{$darab$}_{$ud$}_{$ős1$}_{$ős2$}({$ős1$},{$ős2$})"
```

Itt az őshöz BAR esetben a "_ BAR" is hozzákonkatenálódik, és merge esetben a szabály neve a "_ MERGE" kifejezést is tartalmazza.

Ebben a verzióban a kommentelés és tagolás a legnagyobb hiányosság. Bár a projekt jelenlegi tagjai jól átlátják a szabályokat, a szükséges háttér nélkül nehéz a 2700 szabály között eligazodni. Ezt persze nem lenne nehéz megoldani, mivel a szükséges adatok már

rendelkezésre állnak. Ha a szabályokat a baloldali szimbólum szerint és a közös ős fokszáma szerint csoportosítjuk, akkor a tagolás sem nehézkes. Persze itt is a Merge és a BAR szabályok külön kategóriát képeznek. A szabályok jelenleg gyakoriság szerint vannak sorba rendezve de a név szerinti sorba rendezés is sokat segítene. A kategóriákat mélység szerint is több főkategóriába lehet rendezni.

A kódot legtöbben itt sem látjuk át, mivel még részletes dokumentáció nem készült. A Slime első verziója nem lesz képes ilyen gráf műveletekre, de könnyű lesz benne olyan kódot készíteni, ami mindenki számára érthető, aki az IRTG-t és a Slime-t ismeri. Mindezt pedig minimális overhead-del, így csak a bemeneti adatokat szükséges generálni hozzá. Mivel az alkategóriákon belül a legtöbb szabály összevonható lesz REGEX és templatelés segítségével, a sorrend sem jelent majd problémát.

Chapter 3

A Slime nyelv

3.1 Bemutatkozás

A szakdolgozat keretében kifejlesztett nyelv a Slime nevet kapta. A Slime egy UTLE (Universal Templater Language Extension), avagy egy más nyelvek fölé szánt univerzális bővítmény, ami template-elést használ.

A fejlesztés során az IRTG kiegészítése az elsődleges cél, de ezt a sok iteráció alatt kinőtte a koncepció. Ennek több oka volt. Elősször is az IRTG-nek a legtöbb hiányossága későbbi fejlesztés során megszűnhet a nyelv kiegészítésével (Lásd fejezet ...). A kiegészítéshez a belső működés módosítására nincs feltétlen szükség. Az ALTO-t ugyan viszonylag könnyű kiegészíteni és módosítani, mivel kellően objektum orientált és nyílt forráskódú. Ugyanakkor állandó fejlesztés alatt áll, ezért a forráskód módosítása és kiegészítése is verzió követést igényel. A függőséggel arányosan nő a karbantartási költség. Az ALTO fejlődésével pedig sok kód feleslegessé válik. Ezért külső megoldás kell. Minden nyelv fejlődik, de mindig lesznek fejletlen nyelvek is, amik felett a Slime hasznos lesz.

Programozási és adat leíró nyelveknek széles rétege nem haladják meg azt a szintet, ami a praktikus használatukhoz szükséges. Ennek egyik válfaját alkotják azok a nyelvek, amik még fiatalok, és egyszerűen nem jutottak még el a kellő érettségig. A másik válfaj pedig azok a nyelvek, amiknek a fókusza túl keskeny a bennük rejlő lehetőségekhez képest. A fejlesztőknek sokszor nem éri meg az összes alap funkcionalitást implementálni. Jó kérdés, hogy az IRTG a két kategóriából melyikbe esik vagy nem esik. A Slime küldetése az, hogy ezeket a nyelveket felemelje egy magasabb szintre azok módosítása nélül. Növelje a kódok átláthatóságát, struktúráltságát, és kiírtsa a repetitív kódrészeket.

Az univerzalitást szem előtt tartva egy olyan módszert kell alkalmazni, ami független a kiegészített nyelv fordítójától és szintaxisától. Erre a legoptimálisabb megoldás a template-elés. Először a kiegészített nyelvre generáljuk a kódot és azt futtatjuk a nyelv saját fordítójával. Ez ugyan egy új lépést jelent, de Streameléssel áttetszővé válhat a Slime könnyű súlya miatt. Az ötlet persze nem teljesen egyedi. Eddig is sokféle template processzort és enginet használtak(Lásd fejezet ...) adatleíró nyelvekhez. Ugyanakkor ezek inkább könyvtárak, mintsem nyelvek. Legtöbbször egy magasabb szintű nyelvből lehet kezelni őket, amiknek a támogatása is szükséges. Céljuk a HTML vagy épp CSS kódok aktív manipulációja. A Slimeból sem nehéz könyvtárat készíteni, de hosszú távú célja az abszolút önállóság. Kerüli a függőséget minden felette lévő rétegtől is.

A réteges architektúra és egyirányú viszony okán a kiegészített nyelvre alárendelt nyelv(subordinate language) ként fogok hivatkozni. A templatek esetében megszokott

terminológia “master dokument” ként hivatkozik az alárendelt nyelv sablonjaira. Ezzel ellentétben alárendelt dokumentum(subordinate document) ként fogok hivatkozni rá.

A Slime fiatal, így a Slime-ban is megjelennek még azok a jelenségek, amiket elkerülni igyekszik. Nem haladja meg azt a funkcionalitást, amire tervezve lett. Közel sem turing teljes. Csak a templateléshez és struktúráláshoz szükséges alapfunkciókat tartalmazza(Lásd fejezet ...). Nem tartalmazza a legalapvetőbb aritmetikai műveleteket sem. A kódjában legtöbbször könnyű, de nem mindig elkerülhető a repetitivitás. Még hosszú út áll előtte. Ezekre a problémákra és tervezett megoldásaikra a későbbiekben fogok kitérni. A Slime így is jelentős fejlődést jelent olyan nyelvek számára, amiken a tünetek sokkal súlyosabban jelentkeznek. Ilyen nyelv az IRTG, HTML, XML és irónikusan annak az ANTLR-nek a lexer és parser nyelvtan leíró nyelve, amit a Slime jelenlegi fordítója használ(lást Példák ...). Tehát végső soron a Slime egy olyan nyelv, ami saját magának az implementálását is határozottan könnyebbé tette volna.

3.2 Tervezési szempontjai

A Slime a következőkre összpontosít:

- alárendelt és fölérendelt nyelvtől való függetlenség
- könnyű súly és sebesség
- platform függetlenség
- könnyű bővíthetőség
- esztétikum és egyéniség
- kód átláthatósága és könnyen érthetősége
- szabad tördelhetőség akár több fájlba
- meredek tanulási görbe
- hatékonyan programozás az alárendelt nyelven

Eddig a nyelvektől való függetlenség volt a legalaposabban tárgyalva. A nyelv külön fordítót, templatelést és szöktetést használ e célból.

A könnyű súlyt könnyen parszolható szintaxissal és egyszerű műveletekkel támogatja. Később a robosztus külső könyvtárak elhagyása is sokat fog segíteni.

A platform függetlenség okán Kotlinban készült az első fordító, ami JVM-en fut. Igaz egy C++ 20 verzió is tervben van a könnyű súly miatt. A jelenlegi verzió még a karbantarthatóság miatt OOP alapokon nyugszik, de a C++ 20 már DOP(data oriented programming) alapú lesz.

A könnyen bővíthetőséget modularitással és egységes szintaxissal támogatjuk.

Az esztétikum és felismerhetőség is első sorban a szintaxisra vonatkozik. Itt a legfontosabb az átláthatóság. Fontos, hogy szép kódunk legyen, amiben a nyelvi elemek jól felismerhetőek. Ugyanakkor az is fontos, hogy a Slime kódja kitűnjön az alárendelt nyelv kódjából. (lásd fejezet ...)

A szabad tördelhetőségben segítenek sokat a zárójelek, változók és az importálás megvalósítása. (lásd fejezet ...) A könnyen tanulhatóságot a beszédes, de tömör jelölések, kevés, de sok oldalú műveletek és következetes működés és a kódok átláthatósága is segíti.

A hatékony programozást a tömörített kifejezések és a szükségtelen jelölések elhagyása segítik.

Sokszor kellett kompromisszumokat kötni a szempontok között. Ezeket a döntéseket ugyanakkor legtöbbször inkább elkerültük. Fontos, hogy rugalmasak lenni, és engedni a programozót a saját stílusát követni. A nyelv hemzseg a zárójelektől a könnyű parszolás és átláthatóság érdekében. Ugyanakkor vannak biztosítva módszerek a csukó zárójelek esetenkénti elhagyására. Sok nyelv az adatot elkülöníti a kódtól. Slimeban a kód tartalmazza a templatet, de külön fájlba szervezéssel elkülöníthető. Nem a Slime-ba ágyazzuk az alárendelt nyelv kódját. Az alárendelt nyelv kódjába ágyazzuk a Slime-ot. Így a Slime használatának mennyisége teljesen opcionális. A felhasználó hagyatkozhat teljesen a Slime-ra, vagy használhatja csak a legrepetitivebb kód részek esetén. A Slime esetén akár saját maga is könnyen lehet az alárendelt nyelv. Ezt a szöktető zárójelek nagyban segítik. A REGEX-ek esetén például csak egy karaktert lehet szöktetni, így REGEX-ek között keresni nagyon nehézkes. Slime esetében az alárendelt Slime kód részeket egyszerűen {" "} szöktető zárójelek közé rakjuk.

3.3 Alternatívák template-elésre

3.4 Típusok

Alapjáraton 11 példányosítható típus léteznek a Slime-ban:

- *név:*Name
- *egyszerűek:*Text, Spec, Slot
- *több eleműek:*Temp, List, Type, File, Refe
- *iterátor:*Iter

Ezeknek közös absztrakt őstípusa a Vari. Minden változónak lehet akár több, akár nulla neve. Ezeket jelenleg a szövegtől megkülönböztetve Name típusú változókkal kezeljük. Egy változót vagy a nevein vagy őt birtokló más változókon keresztül érjük el. Az egyszerűekkel kezeljük a szövegeket(Text), speciális karaktereket(Spec) és a sablonok mezőit(Slot). Ezek közül csak a Slot tartalmazhat másik változót, de az is legfeljebb egyet, ami sablon típusú kell legyen. A listák típusosak, de lehet a típusuk Vari is. A Type-pal hozhatók létre új típusok. Ezek a típusok a C struktúráihoz hasonlóan viselkednek. Függvényeket ugyan nem tartalmazhatnak, de névvel és típussal rendelkező attribútumaik vannak. Ezen típusok példányaira összefoglalóan Inst ként hivatkozunk. Inst típusú változót nem hozhatunk létre. Slimeban az importálás File típusú változókkal van megvalósítva. Inicializálás után ezek tartalmazzák az importált Slime fájl kimenetét és elérhető változóit. A Refe valósítja meg a szakdoga egyik főkövetelményét. Rajta keresztül lehet ugyanis a változók között REGEX-szel keresni. Egy Refe egy típus megkötésből és egy REGEX mintából áll. Mindig abban a namespace-ben keres, ahol létre lett hozva. A találatok mindeig elérhetőek rajta keresztül, mint ha az elemei lennének. Minden több elemű változónak van iterátora, amin keresztül for each jellegű viselkedés érhető el.

3.5 Zárójelek

A nyelvben tíz zárójel létezik.

- *komment:*COMM
- *típus jelölő:*TEXT, SPEC, SLOT, TEMP, REFE
- *művelet jelölő:*DECL, DELE, PLUS, EXTE

Ebből csak a komment zárójele nem sorolható be nagyobb csoportba. Az ugyanis a fordításból teljesen kimaradó megjegyzéseket jelöli.

3.5.1 A típus jelölő zárójelek

Minden típus jelölő zárójelhez egy típus tartozik. A hozzárendelt típusokat képesek példányosítani. A példányoknak nem lesz neve Három ezek közül az egyszerű típusokhoz tartozik: TEXT, SPEC, SLOT. A másik kettő a template-hez és a reference-hez. Mindegyik zárójel sajátos szintaxissal is támogatja a változók értékének könnyebb megadását (lásd fejezet ...). A szöveg zárójelet használja a nyelv szöktetésre is. A tartalmát teljes mértékben visszaadja szöveg ként egészen a csukó zárójelig. A SLOT zárójelben csak a slot címkéjét kell megadni, amin keresztül később könnyen hivatkozható a template-en keresztül. A SPEC segítségével több soros templateket is lehet egy sorban deklarálni. Jelenleg még csak a következő karaktereket lehet vele megadni 3-3 féle képpen:

- ‘$\backslash n'$: e, ent, enter,
- ‘$\backslash r'$: r, ren, renter,
- ‘$\backslash t'$: t, tab, tabulator,
- *‘ ’:* s, spa, space
- *‘.’:* pe, per, period
- *‘?’:* qm, qum, question_ mark
- *‘!’:* em, exm, exclamation_ mark
- *‘,’:* co, com, comma
- *‘:’:* cl, col, colon
- *‘;’:* sc, sec, semicolon

Később ki lesz bővítve a teljes unicode karakter készletre, és a karaktereket az unikódjukkal is el lehet majd érni. A TEXT és a REFE kivételével mindegyik többet is képes egyszerre deklarálni. Ekkor a példányokat listában adják vissza.

3.5.2 művelet jelölő zárójelek

A művelet jelölő zárójelekkel érhetőek el a Slime alapműveletei, a deklaráció, törlés, hozzáadás és kiterjesztés.

A deklarációval lehet bármilyen típusú új változót megadni és a törléssel felszabadítani. Deklarációnál adhatunk meg több vagy akár nulla nevet is. A deklaráció összes lehetséges formáját később tárgyaljuk. Egy változó felszabadításához az összes nevén keresztül törölni kell. A felszabadított változói egy importált Slime fájlnak nem elérhetőek.

A hozzáadás a legsokoldalúbb művelet. Lehet vele Slot-ba beszúrni Temp-et, Text, Spec vagy Slot változót. Utóbbi három ilyenkor template-té konvertálódik. Lehet vele egy Temp-et bővíteni Text-tel, Spec-cel, Slot-tal. Lehet vele listába elemet szúrni. Akár adott indexen is. Lehet vele Inst attribútumának értéket adni. Slot-ba Temp tartalmú Listát szúrva annak minden elemére elvégzi a beszúrást, és az eredményül kapott Temp-eket egy listában téríti vissza. Egy több elemű változót egy másikba szúrva, specifikálható, hogy az egyik elemeit a másik melyik elemeibe szúrja bele. Ekkor az előbbi változót téríti vissza. Több elemű változóhoz egy másikat adva az előbbihez hozzáfűzi az utóbbi elemeit. Több elemű változóhoz egy másik iterátorát adva az előbbihez hozzáadja az utóbbi elemeit. Egy több elemű változó iterátorához egy másikét adva előbbi minden eleméhez hozzáadja az utóbbit. Egy több elemű változó iterátorához egy másik több eleműt adva az előbbi minden eleméhez hozzáadnja az utóbbi azonos indexű elemét. Lehetséges több elemű változóhoz adott indexen is hozzáadni a másik elemeit Ha egy iterátorba szúrjuk bele, akkor mindegyik elemébe külön-külön. Ha pedig iterátorhoz adunk hozzás iterátort, akkor páronként végezzük a hozzáadást.

A kiterjesztés a változókat szöveggé alakítja és visszatéríti. Ez az alárendelt kódba ágyazva kiírást jelent. Text esetében annak tartalmát írja ki. Spec esetében a jelölt speciális karaktert írja ki. Slot esetében vagy a Slot tartalmának kiterjesztését, vagy ha üres, a Slot zárójeles formátumát írja ki. Minden több elemű változó esetén (a Temp is ilyen), az elemek kiterjesztését egymás után. Itt specializálni lehet elválasztó karaktersorozatot egy Spec zárójelben. Ha egy több elemű változó tartalmaz egy másikat, akkor az utóbbi elemeit is ugyanazzal az elválasztó jellel fogja kiírni. (ez később módosítva lesz) Fontos az is, hogy a többváltozós változók indexelhetőek. A templateknek a slotjai pedig elérhetőek a templaten keresztül akkor is, ha már tartalmaznak elemet.

3.6 Szintaxis

Tizenegy típus és négy fő művelet. Ennyit könnyű megtanulni. Persze ezek többsége még érthetőbb lesz példákon keresztül. Azok átlátásához előbb ismertetem a pontos szintaxist.

A szintaxis terén volt a legnehezebb megtalálni a hangsúlyt a tömörség az átláthatóság és a tanulhatóság között. Fontos volt az, hogy a jelölések már bevettek legyenek vagy következetesek, hogy könnyű legyen tanulni. De az átláthatósághoz az is fontos volt, hogy a jelölések egyediek legyenek

A bevett jelölésekhez a c alapú nyelvek szintaxisa szolgált alapul. Mint azok a nyelvek, a Slime is a kapcsos zárójelekkel jelöli alapvetően a blokkokat. Mivel a műveleteket az alárendelt nyelv kódjába ágyazzuk, minden művelet egy újabb blokkba kerül. Ez rendes tördelés mellett az átláthatóságot is segíti, és használatával a műveleti sorrend is egyértelmű. A C nyelvekkel ellentétben viszont a blokkok meta adatait a blokkon belül jelöli. Mivel a zárójelek attribútumai és metaadatai is belül vannak, a metaadatokat, mint például a művelet típusa, a lehető legtömörebben egy-két karakterrel jelöljük. Ez a

zárójelek hierarchiájának átlátásában és a tördelésben is sokat segít. A zárójeleken belül az attribútumokat vesszővel és kettősponttal választjuk el. A kettőspont választja el a függőségben lévő attribútumokat és a vessző a függetleneket. Ha az attribútumok között összetettebb hierarchia van, azt a kettőspont mellett kerek zárójelek is jelölik. Egyes műveletek esetén a főbb attribútum halmazokat saját operátorral is elválasztjuk. Például deklaráció során a létrehozott változó metaadatait(név és típus) az attribútumaitól egy ":=" operátor választja el.

Persze a zárójelezésnek a legnagyobb költsége a programozás hatékonyságában van. Nagyon macerás minden műveletnél egy nyitó és egy csukó zárójelet is rakni. Éppen ezért két lehetőség is van tömörítésre.

Az első lehetőség az egysoros és kompakt zárójelek használata. Az egysoros zárójelek a sor végéig tartanak és a kompakt zárójelek pedig az első whitespace karakterig. Így ezeknek végén nem kell csukó zárójelet rakni. Az egy soros zárójeleket kapcsos helyett szögletes zárójellel jelöljük; a kompakt zárójeleket pedig kacsacsőr zárójellel. Így például a deklaráció {= ... =} zárójelének az egysoros párja a [= ... és a kompakt pedig a <=... zárójelek.

A második megoldás az, hogy egy zárójelben többet is végezhetünk az annak megfelelő műveletből pontosvesszővel elválasztva. Ez a jelölés is természetesen a C alapú nyelvekből származik. Persze az egysoros műveletek miatt legtöbbször a sorok elejére kerül a pontosvessző. Így egyébként elfelejteni is nehezebb és egybevonható a következő sort nyitó zárójellel, ha az azzal kezdődik. Sok nyelvben megoldották már a pontosvesszők elhagyását, de többek között a sablonoknál is problémát jelentett volna, ha több sorors szövegeket soronként lehet csak létrehozni. (későbbiekben persze egyes műveleteknél erre is sor kerülhet)

A nyelvben létező műveleteket a következőképpen jelöljük:

művelet	kódnév	blokkos	egysoros	kompakt
deklaráció	DECL	{= ... =}	[= ...	<= ...
törlés	DELE	{x ... x}	[x ...	<x ...
hozzáadás	PLUS	{+ ... +}	[+ ...	<+ ...
kiterjesztés	EXTE	{* ... *}	[* ...	<* ...
szöveg	TEXT	{" ... "}	[" ...	<" ...
speciális	SPEC	{@ ... @}	[@ ...	<@ ...
mező	SLOT	{$ ... $}	[$ ...	<$ ...
sablon	TEMP	{| ... |}	[| ...	<| ...
komment	COMM	{# ... #}	[# ...	<# ...
referencia	REFE	{& ... &}	[& ...	<& ...

Érdekesség:A nyelv neve onnan ered, hogy a műveletek blokkos jelölése a fantasy irodalom slime fajának példányaira emlékeztetnek, amiknek az operátor típust jelölő karakterek a szemei. Az egysoros zárójelekre half slime és a kompaktra pedig slime eye ként is szoktunk hivatkozni szlengesen.

Azt is jó kihasználni, hogy egymásba ágyazott zárójeleknél a beágyazottak nem függenek a befoglaló zárójel típusától. Így egy kompakt zárójelbe ágyazott egysoros zárójeltovábbra is csak az első sortörés zár, ahogy a blokkosat is csak a csukó zárójel. Ez persze az egysoros zárójelbe ágyazott blokkos zárójelre is teljesül. Pélául ez mezőes: <|[@ enter; enter; tabulator

A whitespace-k kezelése másban is eltérhetnek zárójel típusonként. Például az egysoros és kompakt zárójeleknek legtöbbször nem része az őket záró whitespace. Ezért zár több egymásba ágyazott egysor zárójelet is a sort záró sortörés. A kompakt Spec és Slot zárójeleknek viszont része az őket záró whitesspace (Ezt még be kell vezetni ...). Általánosságban a zárójelek minden belső whitespace karaktert figyelmen kívül hagynak. Az alárendelt nyelvbe ágyazott minden speciális zárójel előtti whitespace is kimarad a kimenetből. Text zárójelben minden whitespace megmarad. Temp esetében a sor törő, végi és eleji whitespace-k maradnak csak ki. Temp-ben ezen kívül nincs minden szöveg Text zárójelben. A legtöbb a beágyazott zárójeleken kívül mezőezkedik el. Éppen ezért van szükség kompakt Slot és Spec esetén, arra hogy a záró whitespace ne a környezet része legyen. Különben a kompakt zárójelek nagyon beleolvadnának (Refe : Ezt még be kell vezetni ...)

A zárójelek tartalmának szintaxisa műveletenként egységes. A blokkos zárójeleken szemléltetem:

Comm(comment): {# This is a comment #}

A Comm zárójelek belsejére nincs megkötés.

Refe(reference): {& Text ,t[123] &}

A Refe zárójelekben a típus kritériumot és a regexet vesszővel választjuk el

Text: {" This is a text "}

A Text zárójelek belsejére nincs megkötés.

Spec: {@ enter; tabulator @}

A Spec zárójelben a speciális karakterek kódneveit kettősponttal elválasztva soroljuk.

Slot: {$ slot1 $}

A Slot zárójelben a slotok címkéit kettősponttal elválasztva soroljuk.

Temp: {| text {$ slot1 $}{@ e;t @} text; {"text"} {$ slot2 $}{@ e;t @} |}

A Temp zárójelben minden a sablonok adatait pontosvesszővel választjuk el. Egy sablon adatait elválasztás nélkül soroljuk. A speciális zárójeleken kívüli elemek szövegnek minősülnek.

Decl(declaration): {= name1, name2, name3 :List:Text := {"text1"}, {"text2"} =}

Decl zárójelben a változó neveinek vesszővel való felsorolását egy : és a változó típusa követi. A metaadatokat a := választja el a változó értékétől. A nevek tartalmazhatnak betűt, számot és alulvonást, de nem kezdődhetnek számmal. A nyelvben az egymásba ágyazott típusokat kettősponttal tagoljuk, így: List:List:Temp

Ez C nyelvekben így néz ki: List<List<Temp> >

A zárójelezés viszont csak akkor praktikus, ha egy típusba egyszerre több típus is ágyazható. A Slime jelölése viszont kiegészíthető zárójelezéssel. Például, ha bevezetjük majd a dictionary-ket Dict néven: List:Dict:(Text, List:Temp)

Itt azért nem a kacsacsőr zárójeleket használjuk, mert azokkal már a kompakt zárójelezést jelöljük, amire sokkal alkalmasabb a kerek zárójelnél. Könnyebb észrevenni az alárendelt kódrészletekben.

A változónak nem vagyunk kötelesek nevet adni. Későbbi verziókban a típus paraméter is elhagyható lesz, ha egyértelmű a változó értékéből. Az értékeket a változó típusától függően sokféleképpen megadhatjuk. Minden változónak lehet értéket adni egy path-

szal, ami egy azonos típusú változó elérési útvonalát adja meg ponttal tagolva. Például az inst1 nevű Inst attr3 nevű List típusú attribútumának a harmadik eleméhez vezető path: inst1.attr3.2 (mivel a lista első elemének 0 az indexe). Ha az adott Inst adott atribútumának harmadik eleme egy Temp:

{= temp1 : Temp := inst1.attr3.2 =}

A típus zárójellel rendelkező típusoknak lehet típus zárójellel is értéket adni:

{= temp1 : Temp := {| text {$ slot1 $}{@ e;t @} text {"text"} {$ slot2 $}{@ e;t @} |} =}

A List-eknek lehet egy a beágyazottal azonos típusú zárójellel is értéket adni:

{= specL1 : List:Spec := {@ e;t @} =}

és ilyenkor lehet használni reference-t is, ha már léteznek a megfelelő változók akármilyen néven:

{= specL1 : List:Spec := {&Spec ,∧ spec[123]$&} =}

A Type-oknak elég az attribútumaik típusát azonos típusú attribútumon ként egyszer jelölni:

{= tempi : Type := te1, te2, te3 : Text, sp1, sp2, sp3: Spec, sl1, sl2, sl3: Slot =}

Az Inst-eknél definiálhatjuk, hogy melyik attribútumoknak adunk értéket:

{= t1 : tempi := te1 := {"text"}, sl3:={$slotty$}, sp2:={@e@} =}

de erre sem vagyunk kötelesek, ha az értékeket sorrendben adjuk meg:

{= t1 : tempi := {"text"}, {"text"}, {"text"}, {@e@}, {@e@}, {@e@}, {$slotty$}, {$slotty$} =}

A File egy text-tel hozható létre, ami az elérési útvonalát tartalmazza:

{= f1 :File := "/home/boss/Documents/test_complex_long"} =}

Ekkor a file le is fut és a generált kimenetet is fogja tartalmazni a változó outp néven (pl.: f1.outp), de csak a nem törölt változói elérhetőek.

Name teljes mértékben a Text módjára viselkedik. ugyanúgy is deklarálható, de meg kell adni a típusát. Ugyanakkor ha egy változóba beleszúrunk egy nevekből álló listát, akkor az a neveihez adódik hozzá. (Későbbiekben ez meg fog szűnni. A változók name attribútumába kell majd Text listát szúrni)

Dele(delete): {x temp1 ; specL1; { & Spec ,∧ spec[123]$ x}

Törlés esetén a változókat megadhatjuk path-szal vagy referenciával

Plus: {+ temp1.slot1 :+ temp2 +}

Plus zárójelben a két változót megadhatjuk path-szal, de akármilyen típus és műveleti zárójellel is (a törlés kivételével, mert az nem térít vissza értéket). A bal oldaliba szúrjuk be a jobb oldalit. A két változót :+-szal választjuk el. Több elemű változók esetén azt is megadhatjuk path-szal, hogy a baloldali melyik eleméhez melyik elemét addjuk a jobb oldalinak. Ekkor ar elemeknek a birtokosukon belüli mezőt kell megadni path-szal. A listát vesszővel tagoljuk, kettősponttal választjuk el a Plus többi attribútumától és a párokat :+-szal választjuk el, így:

{+ temp1 :+ inst1 : sl1 :+ at1 , sl2 :+ at2+}

Ahogy korábban említettem az iterátorokkal lehet különböző hozzá adásokat végezni több elemű változók között:

{+ list1 :+ list2 +} {# adds all elements of list2 to list1 #}

{+ list1.iter :+ list2 +} {# plus list2 to all elements in list1 #}

{+ list1 :+ list2.iter +} {# plus all elements of list2 to list1 #}

{+ list1.iter :+ list2.iter +} {# plus all elements of list2 to the elements of list1 on the same index #}

Exte: {* variable1; variable2* {# returns it as text#}

Exte zárójelben a kiírni kívánt elemeket pontosvesszűvel elválaszva soroljuk.

{* list1 : {@ e;e;e@ } *} {# You can add divider character by special character # }

{* list1 : {|This is a temp, witch can include special character too. Like {@ e@ } for example|} *} {# or template# }

A több elemű változóknak megadhatunk elválasztó speciális karakter sorozatot és templatet is.

3.7 Implementáció

A fordító jelenlegi implementációja Kotlinban készült (link ...).

Az implementáció ANTLR-t (link ...) használ a parszoláshoz. Az ANTLR a szintaktikai elemzés során feltokenezi a szöveget és szintaktikai fát épít. Az tokenezéshez szüksége van egy lexer grammar-re (SlimeLexer.g4), és a fa felépítéséhez egy parser grammarre(SlimeParser.g4). Ezek tartalmazzák a nyelv átírási szabályait, és a tokenezéshez szükséges REGEX szerű, de annál fejlettebb mintákat. Az ANTLR ezekből a nyelvtanokból osztályokat generál. Sok nyelvben képes erre. Ugyanakkor Kotlinban még nem képes, így Java osztályokat generáltatunk vele. Ezek az osztályok végzik a tokenezést (SlimeLexer) és a parszolást(SlimeParser). Ezen kívül olyan osztályokat is generál, amik alapul szolgálnak a fa listeneres(SlimeParserBaseListener) és visitoros(SlimeParserBaseVisitor) bejárásához, és a szemantika megvalósításához. Én a visitoros bejárást választottam, mivel a

Az ANTLR két féle nyelvtant használ, lexer és parser nyelvtant. Előbbi felelős a tokenezésért és az utóbbi a parszerelésért. Ezek össze is vonhatóak, de ez nem felelt meg a célnak, mert vegyes nyelvtanban nem lehet a lexernek több módja.

A Slime esetén a tokeneket az alárendelt nyelv kód részei, a nyitó és csukó zárójelek, elválasztó karakterek és változó nevek teszik ki.

A lexer nyelvtan fragment-ekből és lexer rule-okból áll. A fragmentek önmagukban nem képeznek tokent, de részei lehetnek lexer szabályoknak. Többek között arra jók, hogy tömörebben írhassunk le hasonló szabályokat, vagy olyan szabályokat, amikben sok az ismétlés. Ugyanakkor nem mindig képes feladatát teljesen ellátni.

A lexer képes váltogatni módok között, amikben más szabályok lesznek érvényesek. A vermeket verem módjára kezeli az ANTLR. Megadható, hogy mely tokenek esetén menjen be egy adott módba és az is, hogy annak a módnak mely tokenjei esetén térjen vissza. Például ilyen, hogy "|" esetén belépünk a Temp belsejében érvényes szintaxis lexer módjába és "|" esetén pedig visszatérünk. Nagy gyengesége a nyelvnek, hogy nem tartozhatnak a szabályok több módhoz is. Így a Slot nyitó zárójelét is hét különböző token kezeli attól függően, hogy melyik módból akarjuk elérni. Ez alól a fragmentek kivételt jelentenek,

amik minden mode alatt azonosak. Használhatjuk is őket, de nem jelent tömörítést, ha a "|" zárójelet egy minimum két karakteres nevű fragment-be csomagoljuk. Minden speciális zárójelnek külön módja van, kivéve a 4 operátor zárójelnek (OPER) és a Slot-Spec párosának (SLSP). Ezen kívül minden módnak be kellett vezetni egy egy soros és kompakt változatát is, mert másképp nem megoldható a whitespace-k egyedi kezelése.

Erre is jó megoldást nyújt a Slime. Van a kódban 27 nem üres sor a nyitó zárójelek kezelésére, ami négyszer is előfordul a szabályok nevének pár karakternyi változtatásával, és önmagában is sok ismétlést tartalmaz. Ennek karbantartása már template-eléssel nem jelentene problémát. Sőt template-eléssel a fragmentek is teljes mértékben kiválthatóak.

A parser szabályok már a lexer és más parser szabályokból építkeznek. Ezek feladata felismerni az egyes műveleteket és beazonosítani az attribútumokat. Az egységesség céljából minden művelet parser szabálya egy head body és tail részből áll, ahol a head és a tail csak a nyitó és csukó zárójelek és csukó whitespacek. Igaz, még itt is jól látszanak az ANTLR hiányosságai, mivel a head-ek némeike 21 token-ből áll (3 féle nyitó zárójel 7 módban). Az összetettebb zárójeleknél, mint Decl, Exte vagy Plus, a body további bodyPart-okra van bontva. Az ANTLR minden parser szabályra generál egy leszármazottat a parsz fája csúcsából. Ezekre az osztályokra tartalmaz a visitor osztály visit függvényeket. A Slime esetében ez a visitor alap a SlimeParserBaseVisitor és a belőle származtatott osztályom a MySlimeParserVisitor, ami a fordító legfőbb funkcionalitását valósítja meg. minden zárójel parszer szabályának a body-ja állítja elő a valódi kimenetet, amit ő csak továbbít. Összetettebb esetekben a body is csak a bodyPart-ok végeredményének az összegzését végzi. Ez a kimenet pedig mindig valamilyen a Slime-ban létező típussal rendelkezik.

A Slime típusainak mind létezik a háttérben megfelelő osztály, ami S*typeName* módra van elnevezve. Mindegyik az SVari absztrakt ős osztályból származik.

A lexer nyelvtan szabályaira alap

agramatikus

6-os a maiból

dávid email

szöveg tisztázása

bevezetők

latexbe

ábrák

Slime

egyszerű kétemeletű példa

10 darabra

3.8 A fejlesztés fázisai

3.9 Fejlett megoldásaink

Acknowledgements

Köszönöm mindenkinek, aki munkásságával és vagy jelenlétével a szakdolgozat elkészüléséhez és vagy minőségéhez hozzá járult! Köszönet elsősorban konzulenseimnek, Ács Evelinnek és Nemeskey Dávidnak, akik mindig rendelkezésemre álltak, ha forrásokra, segítségre, jó tanácsokra vagy csak motívációra volt szükségem! Köszönet Recski Gábornak és Kornai Andrásnak, akik elindítottak a Szintaktikai és Szemantikai elemzés területén. Köszönet az AUT tanszéknek, amiért támogattak abban, hogy eljussak az MSZNYK-ra, ahol megismerhettem a számítógépes nyelvészet magyar vonatkozású eredményeit. Végezetül, de nem utolsó sorban pedig köszönet szüleimnek és Vágó Nórának, amiért biztosították a lelki és anyagi hátteret a munka folyamán is.

Appendix

A.1 A TeXstudio felülete

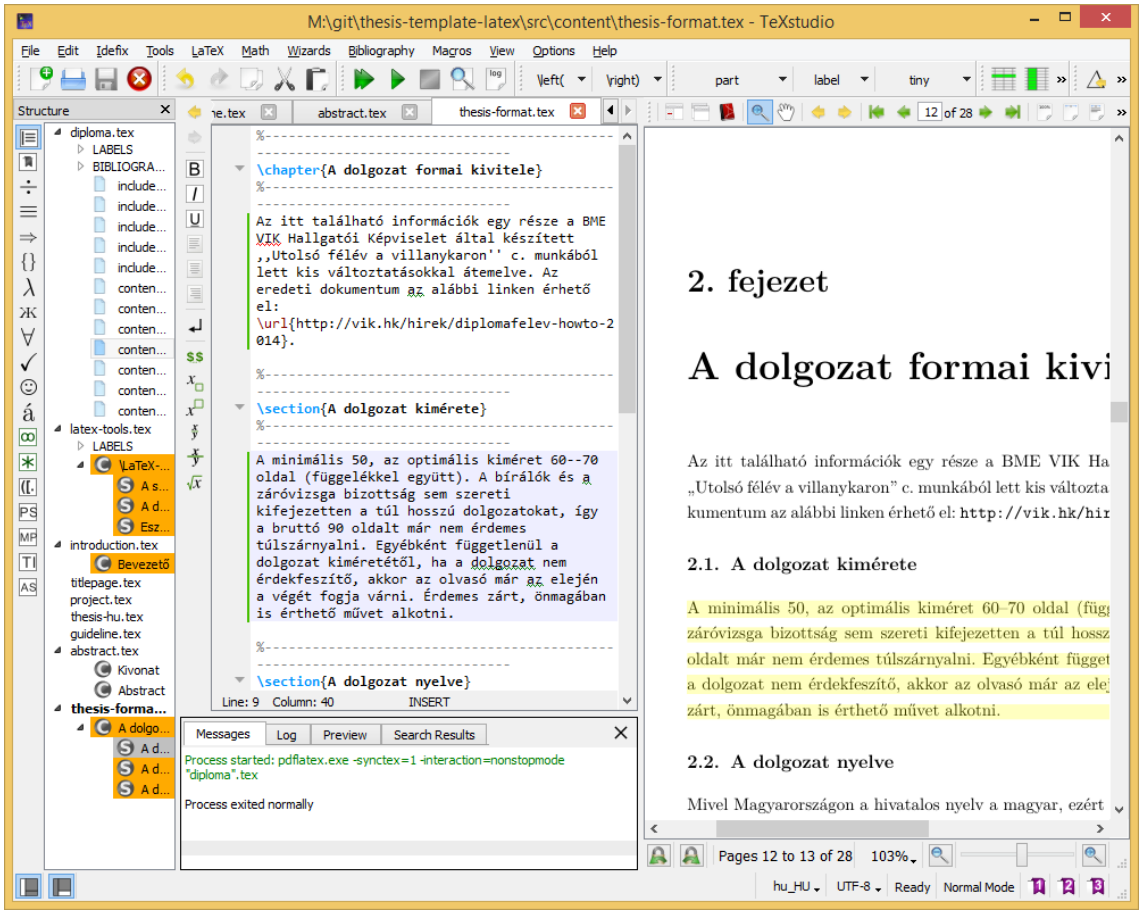

Figure A.1.1: A TeXstudio LaTeX-szerkesztő.

A.2 Válasz az „Élet, a világmindenség, meg minden” kérdésére

A Pitagorasz-tételből levezetve

$$c^2 = a^2 + b^2 = 42. \tag{A.2.1}$$

A Faraday-indukciós törvényből levezetve

$$\operatorname{rot} E = -\frac{dB}{dt} \qquad \longrightarrow \qquad U_i = \oint_{\mathbf{L}} \mathbf{Edl} = -\frac{d}{dt} \int_{A} \mathbf{Bda} = 42. \tag{A.2.2}$$