Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék

Nyelvfüggetlen könnyű bővítmény sablon műveletekkel

TDK dolgozat

Készítette:

Holló-Szabó Ákos

Konzulens:

Dr. Mezei Gergely

2019

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

Alulírott *Holló-Szabó Ákos*, MSc-s hallgató kijelentem, hogy ezt a Tdk dolgozatot 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. október 28.

Holló-Szabó Ákos
hallgató

Kivonat

Napjainkra bebizonyosodott, hogy közel lehetetlen lenne az informatika szerteágazó tudományában minden kihívást csupán néhány programozási nyelvvel hatékonyan megoldani. Mai napig születnek új nyelvek több-kevesebb létjogosultsággal, pedig már eddig is létezik több tízezer. Számos olyan adatleíró-, programozási nyelv és nyelvtan létezik, amit csak egy-egy hasznos funkcionalitás miatt fejlesztettek ki. Ezekből a nyelvekből gyakran több, más nyelveknél megszokott, népszerű nyelvi elem hiányzik, ami azonban nem kizáró ok, mivel a speciális célterületen nagy hatékonysággal használhatóak. Ilyen célterület például a Natura Language Processing (NLP) területén a mondatok feldolgozása, fákká és egyéb szintaktikai és szemantikai reprezentációkká alakítása. Az IRTG (Interpreted Regular Tree Grammer) nyelvtan a szabályok deklarációja után számos formalizmus közötti kölcsönös konverzióra képes. Ez egy egyedülálló funkcionalitás, ami új reményt ad az NLP analitikus megközelítéséhez. Ugyanakkor ez a nyelvtan nem valósít meg ennél a funkcionalitásnál többet. Hiányzik az importálás, ciklusok, feltételes elágazások, de még a hagyományos értelemben vett változók sem léteznek benne. A korlátozás a nyelv széles körű, hétköznapi használatát teljesen ellehetetleníti. Egy másik példa az Android layoutok esetén használt XML. Az XML-nél hiányzik a sablon használat, az include-olás és a hagyományos értelemben vett változó használat.

A korlátozott funkcionalitásból fakadó problémák az érintett nyelvek bővítésével könnyen megszüntethetőek. Nincs értelme azonban ezen általános problémák esetén nyelvspecifikus megoldásoknak, mivel a kibővített nyelvek fejlődésével a problémák megszűnhetnek, a kiegészítést feleslegessé téve. Az általános megoldáshoz sablonok segítségével jutunk el. A kiegészített nyelv jellegzetes, ismétlődő részeit sablonosítjuk úgy, hogy csak az alapértelmezettől eltérő adatokat kelljen újradefiniálni. Már eddig is számos hatékony megoldás létezik sablonkezelésre, de ezek többsége, egy adott nyelvre összpontosít. Éppen ezért a szintaxisukat is úgy alakítják, hogy ezt az egy nyelvet minél hatékonyabban tudják kezelni. Általában nem is önálló megoldások, hanem szükségük van egy magasabb szintű nyelvre, ami a sablonműveleteket vezérli. A kettős nyelvi megoldás miatt külön kell fordítani a sablonokat és az azokat kezelő kódot. Az időveszteség jelentősebb mennyiségű adaton már nem elhanyagolható.

Dolgozatomban egy általam kidolgozott új megoldást, a Language Independent Lite Extension by Templating Operationt (LILETO) mutatom be. A megoldás lényege, hogy egy olyan réteget képzünk az "alárendelt" nyelv fölé, ami lehetővé teszi az alapvető programozási módszertan használatát, miközben megtartja az eredeti nyelv hatékonyságát. Elérhetővé teszi a változók, a struktúraszintű típusok, a kollekciók, az importálás, a ciklusok és a feltételes elágazások használatát. A kibővített nyelv az "alárendelt" nyelvvé fordul, amit a saját fordítója fordít gépi kóddá, ami által a bővítmény fordítója független marad az "alárendelt" nyelv saját fordítójától.

A dolgozatomban bemutatom a LILETO koncepciót, a Lileto nyelv jelenlegi fázisát, közeli fázisait, felhasználását és párhuzamot vonok a Lileto és más hasonló megoldások között.

Abstract

Already today, it is proven that it's almost impossible to solve every challenge in the broad fields of computer science efficiently with just a few programming languages. Until this day new languages are created, even though there are already tens of thousands of them. There are many data description and programming languages and grammars, which were developed for some particular useful functionality. Often times some elements which are present in other popular languages are missing from these languages, which is not a reason for exclusion because these languages perform well for their intended use. One such use is for example the process of transforming sentences into trees and other syntactical and semantic representations in the field of Natural Language Processing (NLP). The IRTG (Interpreted Regular Tree Grammar) is capable of inter-conversion between several formalism right after the declarations of its rules. This is a unique functionality, which gives a new hope for the analytic approach to NLP. At the same time, it does not implement more functionality than this. Importing, cycles, conditional junctions are missing, even variables in the traditional sense are non-existent in it. This restriction makes the extensive, everyday use of the language near impossible. Another example is XML when defining Android layouts. XML has no support for templating, including and variable usage in the traditional sense. The problems emerging from the restricted functionality can be eliminated by extending the affected languages. However, there is no point of language specific solutions for such general problems, since the development of the extended languages can eliminate the problems, making the extension unnecessary. We reach the general solution with the help of templates.

The problems caused by the limited functionality can be easily solved by the extension of the involved languages. But there is no point in language specific solutions in case of general problems, since the development of the extended languages can eliminate the problems making the extension unnecessary. We can reach the general solution by the help of templates. We should make templates out of the typical and repeated parts of the subordinate language, so only the data different from default must be redefined. There are plenty of efficient solutions for templating these days, but the most of them concentrate on one specific language. Therefore, their syntax is designed for the efficient handling of that one specific language. Usually they are not standalone solutions, they need a higher-level language that controls the templating operations. Due to the dual-language solution, the templates and their manager code must be compiled separately. The time overhead is not negligible in case of significant volume of data. In my thesis I am going to present a new solution that I developed, called the Language Independent Lite Extension by Templating Operations (LILETO). The essence of the method is to build a layer above the subordinate language that makes the basic methodology of programming possible while keeping the efficiency of the original language. It makes possible the usage of variables, structure level types, collections, importing, cycles and conditional junctions. The extended language compiles to the subordinate language, that is compiled to machine code by its own com-

piler, so the extensions compiler stays independent from the subordinate language's own compiler.

Within my TDK thesis I am going to present the concept of LILETO, the present and upcoming phases of the Lileto language, its usage and I am going to establish a parallel between Lileto and other similar solutions.

1. fejezet

Bevezetés

A dolgozatom keretén belül a `LILETO` koncepcio kidolgozásával és annak implementációjával foglalkoztam. A `LILETO` (Language Independent Lite Extensions by Templating Operations) kifejezést rövidíti. A konkrét implementáció pedig a fejlesztés alatt lévő `Lileto` nyelv, amin keresztül a koncepciót is szemléltetni fogom. A `LILETO` lényege, hogy:

1. *Language Independent:* A kibővített nyelv fordítójától és szintaxisától függetlenek maradunk.
2. *Lite Extensions:* A kibővített nyelvet úgy egészítjük ki általános hasznú funkcionalitásokkal, hogy fordítási teljesítményigénye csak elhanyagolható mértékben nő.
3. *By Templating Operations:* Mindezt egy sablon műveletekre optimalizált szkript nyelvvel valósítjuk meg, ami saját fordítójával fordul tovább a kibővített nyelv forráskódjára.

Egy programozási nyelv kibővítésére számos okunk lehet:

- A nyelvet túl kötöttnek tartjuk, ami a fejlesztést lassítja.
- A nyelvet túl kötetlennek tartjuk, ami a számtalan hibalehetőséget teremt.
- A nyelvből hiányoznak magasabb szintű nyelvi elemek, amik egy OOP programhoz elengedhetetlenek.
- A nyelv kódja túlságosan repetitív.
- A nyelv kódja nem elég átlátható.
- A nyelv nem Turing teljes.
- …

Ezek többsége a nyelv szintaxisának és funkcióinak hiányosságából fakad, ezért jó megoldás a bővítés. A hiányosságnak is számos oka lehet:

- A nyelvet nem arra tervezték, mint amire használjuk.
- A nyelv csak ”proof of concept” céllal lett implementálva.
- A nyelv túl fiatal, így nem tartalmazza még a megoldásokat az összes problémára.
- A nyelvet éppen mi fejlesztjük, de még tervbe sem vettük a funkciókat.
- A nyelvet adat tárolásra fejlesztették és így könnyebb feldolgozni.

- A nyelv olyan régi, hogy a funkciók túl megterhelőek lettek volna az akkori számítógépeknek.
- A nyelv fejlesztői nem rendelkeztek a kellő tapasztalattal.
- …

Ugyanakkor nem is érdemes egy darab nyelvhez bővítményt készíteni. Ha a megoldásaink nem hordozhatók, akkor kódunk nagy része idővel feleslegessé válhat. A legtöbb programozási nyelv állandó fejlődés alatt áll. Az adott nyelv számunkra megterhelő hiányosságai annak fejlődésével megszűnhetnek. A fejlesztésünket nem szeretnénk kitenni a verziókövetés állandó költségeinek. Főleg nem érdemes egy olyan bővítményt készíteni, amit közvetlen a kiegészített nyelv fordítójához illesztünk. A fordító ugyanis akkor is változhat, ha maga a nyelv nem. A fordító fejlesztőinek a visszafele kompatibilitás csak a nyelvük forrás kódjai esetén fontos. Az számukra a legfontosabb szempont, hogy a nyelv felhasználóinak ne kelljen a régi kódjaikat módosítani minden új verzió kiadásánál. A kódjaink nem hogy feleslegessé válhatnak, hanem megeshet, hogy nem működnek egy adott alverzión túl tovább, mert megváltozott egy darab interfész a fordítóban, amire építettünk. Éppen ezért mindenképpen egy olyan megoldást keresünk, ami a kiegészített nyelvre fordul és onnan fordul csak tovább annak a saját fordítójával. (Ezt a megoldást alkalmazza például a `typescript` nyelv fordítója is, ami a `javascript` nyelvet bővíti ki erősebb típusossággal és további OOP-t segítő megoldásokkal.) Ugyanakkor kijöhet a kiegészített nyelvnek egy olyan főverziója, ami már nem kompatibilis a régi verziókkal. Ekkor vagy követjük a verziót, vagy megragadunk a kiegészített nyelv utolsó általunk támogatott verziójánál. Utóbbival le is mondunk az új verziókról és esetleges új megoldásaikról. Éppen ezért fontos, hogy a megoldásunk rugalmasan kezelje a kiegészített nyelv szintaxisát. Ha viszont a fordítótól és a kiegészített nyelv szintaxisától már nem függünk, akkor mennyire kifizetődő magától a nyelvtől függeni?

Ez már nem egy egyértelműen eldönthető kérdés. Én azt az utat vizsgálom, amikor a kérdésre nem a válasz. Megoldásom kulcsa pedig a sablon műveletek. A sablon nem más, mint egy hiányos szöveg, amiben a hiányzó szöveg helye meg van jelölve. A megjelölt helyeket hívjuk a sablonok mezőinek, amiknek jele általában egy-egy szöveges név. Ezekre azért van szükség, hogy később a mezőkre könnyen tudjuk hivatkozni. Sablonokba csomagolva a kiegészített nyelv jellegzetes szerkezeteit (bennük helyet hagyva a változó részeknek) teljesen eltűnik előlünk az adott nyelv. Ha a sablonokat elkészítettük, akkor olyan mint ha csak a sablon nyelvben programoznánk onnantól. A szerkezeteket annyiszor, ott és úgy alkalmazzuk, ahányszor, ahol és ahogy csak szeretnénk. A kiegészített nyelv kódját a sablonokba adatokat illesztve generáljuk. A fordítótól függetlenek maradunk, ha a nyelv szintaxisa változik, akkor csak a megfelelő sablonokon kell módosítani. Egy új nyelvi elem esetén csak létrehozunk egy-két új sablont. Ha átakarunk térni egy teljesen másik nyelvre, akkor semmi más dolgunk nincs, mint megírni az új sablonokat és újra visszatérhetünk a bővítmény nyelvéhez.

A réteges architektúra és egyirányú viszony miatt a kiegészített nyelvre `alárendelt nyelv`ként (subordinate language) fogok hivatkozni. A template-ek esetében megszokott terminológia `master document`ként hivatkozik az alárendelt nyelv sablonjaira. Ezzel ellentétben én `alárendelt dokumentum`ként (subordinate document) fogok hivatkozni rájuk. A `Lileto` az alap koncepción túlmutató céljai:

- Növelje az alárendelt nyelv kódjának átláthatóságát
- Segítse az alárendelt nyelv kódjának strukturálását.
- Kiiktassa az alárendelt nyelvből a repetitív kódrészeket.

- Behozza azokat az alapvető megoldásokat, amik egy nyelv alapműködéséhez szükségesek.

A `Lileto` mindezt minimalista, letisztult szintaxissal is támogatja. A `Lileto` hosszú távon egy Turing teljes nyelv lesz, ami a szövegkezelést, számkezelést és gráfkezelést is meg fogja valósítani. Sőt hosszú távon egy teljes értékű objektumorientált nyelv a cél. Az objektum orientáltsághoz viszont már egy sajátos ön sablonosító megoldást fog használni. Hosszú út vezet a teljes megvalósulásig.

Az ötlet nem teljesen egyedi, eddig is sokféle template processzort és engine-t használtak adatleíró nyelvekhez az ismétlődések és strukturálatlanság kezelésére. Ugyanakkor ezek inkább könyvtárak, mintsem nyelvek: legtöbbször egy magasabb szintű nyelvből lehet kezelni őket, amiknek a támogatása is szükséges. Céljuk tipikusan HTML vagy CSS kódok folytonos manipulációja. A `Lileto`-ból sem nehéz könyvtárat készíteni, de hosszú távú célja az abszolút önállóság. Kerüli a függőséget minden felette lévő rétegtől is.

A `Lileto` még fejlesztés alatt ál, így benne is megjelennek természetesen az elkerülni kívánt jelenségek. Nem mindig elkerülhető a kódok repetitivitása. Nem haladja meg azt a funkcionalitást, amire tervezve lett, közel sem Turing-teljes. Nem tartalmazza a legalapvetőbb aritmetikai műveleteket sem. (lásd 4) Csak a template-eléshez és strukturáláshoz szükséges funkciókat implementálja . Ezekre a problémákra és tervezett megoldásaikra a későbbiekben fogok kitérni. A `Lileto` így is jelentős fejlődést jelent olyan nyelvek számára, amelyekben ezek a problémák sokkal nagyobb mértékben jelentkeznek(lásd 5. Ilyen nyelvek és nyelvtanok az IRTG, HTML, XML és ironikusan annak az ANTLR-nek a `lexer` és `parser` nyelvtan leíró nyelve, amit a `Lileto` jelenlegi fordítója használ. Tehát végső soron a `Lileto` egy olyan nyelv, ami saját magának az implementálását is határozottan könnyebbé tette volna.

2. fejezet

Alternatívák sablon kezelésre

Ebben a fejezetben azt mutatom be, hogy a ma fellelhető megoldások miben térnek el a `Lileto`-tól. Nem kívánom a nyelveket minősíteni. Más a céljuk, így természetes, hogy nem tudnak a `Lileto` célkitűzéseinek eleget tenni. Részletesen kifejteni sem kívánom a nyelvek részleteit, mert az meghaladná a dolgozat kapacitását. Rengeteg alternatíva létezik template-elés terén az itt felsoroltakon kívül is. Ezekkel a példákkal csupán egy áttekintést kívánok nyújtani, mert fontos átlátni a környezetet és nyelvközösséget, aminek a `Lileto` is része lesz.

2.1. Jinja2[1]

A Jinja 2 egy Python alatt futó sablonkezelő nyelv. A sablonokba építhető benne a beszúrt adatokat kezelő logika, de az adatok előállítását már egy magasabb szintű nyelv végzi. Főleg HTML és XML kódok kezelésére használják, de tud bármilyen szöveg alapú nyelvet generálni(HTML, XML, CSV, LaTeX, stb.). Egy sablon tartalmaz változókat és/-vagy kifejezéseket, illetve a tag -eket. A változók/kifejezések lesznek kicserélve a megfelelő értékekkel, amikor lefut a render, a tagek pedig a logikáját irányítják a sablonnak. A nyelv jelölései:

- *{% ... %}:*az állítások
- *{{ ... }}:*a kifejezések amik ki lesznek írva a sablon kimenetén
- *{# ... #}:*a komment
- *# ... ##:* a sorműveleteknek

A változókat filterekkel módosíthatjuk a legkönnyebben. A filterek el vannak választva a változótól egy '|' szimbólummal. Lehetnek argumentumaik, mint a függvényeknek. Például ezekkel végezhető HTML szöktetés is.

A változókat és kifejezéseket lehet tesztelni. A felső nyelvből vezérelhető a whitespace -ek kezelése Szöktetni kétféle képen lehet:

- '..' karakterek használatával
- %...% blokk használatával

A vezérlő nyelvtől függően lehet egy sort sorműveletté konvertálni. A nyelvben van sablon öröklés.

Ezzel jól definiálhatók olyan nyelvi szerkezetek, amik később újra felhasználhatók és specifikálhatók. Példa:

[1] `https://pypi.org/project/Jinja2/`

```
<!DOCTYPE html>
<html lang="en">
<head>
{% block head %}
<link rel="stylesheet" href="style.css" />
<title>{% block title %}{% endblock %} - My Webpage</title>
{% endblock %}
</head>
<body>
<div id="content">{% block content %}{% endblock %}</div>
<div id="footer">
{% block footer %}
&copy; Copyright 2008 by <a href="http://domain.invalid/">you</a>.
{% endblock %}
</div>
</body>
</html>
```

Beszúrandó

```
{% extends "base.html" %}
{% block title %}Index{% endblock %}
{% block head %}
{{ super() }}
<style type="text/css">
.important { color: #336699; }
</style>
{% endblock %}
{% block content %}
<h1>Index</h1>
<p class="important">
Welcome to my awesome homepage.
</p>
{% endblock %}
```

Ennél a példánál az alap sablonban meg vannak jelölve blokkok, amikre nevükkel lehet hivatkozni, majd a gyerek sablonban felüldefiniáltuk a blokkok tartalmát. Mivel a footer blokk nem volt definiálva a gyerekben, így az a szülőben definiáltakkal egyezik. El lehet érni más sablonokat a fájlnevük megadásával Nem tudsz több blokkot azonos névvel definiálni ugyanabban sablonban. Egy blokkot többször a a *self* speciális változón keresztül lehet felhasználni.

```
<title>{% block title %}{% endblock %}</title>
<h1>{{ self.title() }}</h1>
{% block body %}{% endblock %}
```

El lehet érni a szülő blokkot is a super változón keresztül Meg lehet adni a blokkok zárásánál a nevet a jobb olvashatóság érdekében

Lehet egymásba ágyazni a blokkokat. Alapértelmezetten a blokkok változói nem érhetők el kívülről. Elérhetőségüket a scoped kulcsszóval lehet módosítani. Ha a sablon kontextusában benne van a sablon kitöltésére szánt objektum, akkor nem kell string-ként írni az extends kulcsszó után az objektumot.

Alapvető nyelvi szerkezetek (blokkokba kell rakni):

- *ciklus:* for: break, continue
- *elágazások:* if/else / elif
- *makrók:* macros
- *függvény hívások:*call
- *szűrők:* filters
- *hozzárendelés:* assignment
- *származtatás:* extend
- *blokkok:* blokk
- *beemelés:* include: ignore missing
- *importálás:* import

Kontextusok: prózai, matematikai, összehasonlítási, logikai
További operátorok:

- in : benne van e egy adott halmazban
- is : igaz e rá, hogy
- | : csővezeték
- ~: to string and concatenate
- () : függvény hívás
- / [] : objektum attribútumának lekérdezése

2.2. Smarty[2]

A Smarty egy rendkívül fejlett templater engine. Mint a legtöbb templater, PHP-ra és HTML-re szakosodott. A blokkjait kapcsos zárójellel határolja.

Számos funkcionalitást lehet vele a template-en belül jelölni. Ezeknek a részletezése nem fér bele a szakdolgozat kereteibe, ezért csak a szintaxis szemléltetésének céljával sorolok fel alapvető nyelvi elemeket:

- *variable: {$ name }* A Smarty mezői, ők maguk is tartalmazhatnak más változókat.
- *functions: {funcname attr1="val1" attr2="val2"}*
- *komment: {* This is a comment *}*
- *import: {include 'page_footer.tpl'}*
- *cycle:*

```
<tr class="{cycle values="odd,even"}">
<td>{$data[rows]}</td>
</tr>
```

[2] `https://www.smarty.net`

A ciklus megvalósítása. Végigiterál az értékeken.

- *upper: {$name|upper}* Nagy betűkkel szúrja be a szöveget.
- *foreach:*

```
{foreach $nameList as $name}
<li>{$name}</li>
{/foreach}
```

  Loops throw the whole array. Egy fejlett megoldás, ami sokféleképpen használható, nem részletezem a szakdolgozat kereteiben.

- *display:* Kiír konzolra egy template-et.

Összefoglalva a Smarty minden műveletet kapcsos zárójelekbe zár. Ezeket figyelmen kívül hagyja, ha whitespace van előttük és utánuk is. A függvények nevét a kapcsos zárójelekkel jelöli. Igyekszik mindent saját nyelvi elemekkel megoldani. A szintaxisa viszont körülményes és funkcióit nehéz rendesen kitanulni.

2.3. T4[3]

C#-ban és Visual Basic-ben elérhető szöveg sablon nyelv. A generált fájl lehet akármilyen szöveg típus. Kétfajta T4 van: a futás idejű és a design idejű.

Egy sablon három részből áll:

- Irányelvek

  Elemek, amik irányítják a sablon vezérlését

- Szöveg blokkok

  A tartalom, ami közvetlenül másolódik a kimenetre

- Vezérlő blokkok

Vezérlő blokkok:

- <# ... #> Általános
- <#= ... #> Kifejező (elvégzi és stringé konvertálja)
- <#+ ... #> Osztályok elemei, amiknek nem szabad megjelennie a kimeneten.

Importálható nyelvek:

- leg gyakrabb .NET asembly-k.
- más névterek változói és metódusai.
- más sablonok szövegek és kódja.
- alapvető utility metódusok.

Lehet változókat és függvényeket létrehozni/meghívni a blokkon belül. Lehet egymásba ágyazás.

[3] https://docs.microsoft.com/en-us/visualstudio/modeling/code-generation-and-t4-text-templates?view=vs-2019

2.4. Acceleo[4]

Eclipse plugin-ként érhető el. .mtl a kiterjesztés. Az Acceleo 3 az OMG által definiált MOFM2T szabvány implementációja. Az Acceleo nyelv, amit az OMG MTL-ként hivatkozik, egy modulja két főbb típusba tartozó struktúrából (sablonok és lekérdezések). Az Acceleo-ban, az OCL részhalmazát képző kifejezésekkel készíthetsz lekérdezéseket.

Modulok:

- *általános szintaxis:*

  [module <module_name>('metamodel_URI_1', 'metamodel_URI_2')]

- *Import:* import qualified::name::of::imported::module
- Statikus fölülírás

Sablonok fejléce

- *általános szintaxis:* [template...][/template]
- *Előfeltételek:* () ? (...)
- *Utó-kezelés:* () post()
- Változó inicializálás

lekérdezések: [query ... /]
A nyelv elemei:

- *Fájl tag:* [file(<uri_expression>, <append_mode>, <output_encoding>)]
  (...) [/file]
- *For ciklus:* [for (i : E | e)]...[/for] vagy [for (<iterable_expression>)]...[/for]
- *if:* [if (condition)]...[/if]
- *let:* [let (variableName : VariableType = expression)]...[/let]
- *comment:* [comment/]

2.5. Extend[5]

Javara fordul. Sokban azonos a szintaxisa a Java -val. Egy bővítmény, aminek a célja a Java megkötéseinek lazítása.

Különbségek:

- Nem kötelező a ; írása
- Konstruktorok

  Nem muszáj kiírni az osztály nevét hanem csak egy new -t elegendő írni

- A változókat (fields) lehet val és var kifejezéssel deklarálni.

  val hogyha final

[4] `https://wiki.eclipse.org/Acceleo/User_Guide#Behavior`

[5] `https://www.eclipse.org/xtend/documentation/202_xtend_classes_members.html#extension-methods`

- Ha a függvény kimenetét ki lehet találni akk nem kell deklarálni.
- Van operátor felülírás.
- Dispatch írása engedi h két azonos nevű, de különböző bemenetű függvény legyen.
- Metódus készítésekor az xtend megengedi, hogy egy lépésbe végezzük el a gráfműveleteket.
- Kiegészítő (extensions) függvények osztályokhoz. (szerintem előre kész lévő függv.)

  Lehet őket külső helyről beimportálni.
- Annotációs típusokat lehet definiálni.

Önálló nyelv, de közvetlen java fölé lett készítve.

2.6. Freemaker[6]

Apache által készített sablon motor. Ez egy Java -hoz írt könyvtár, amely szövegeket generál(HTML, email, fájl konfiguráció, forráskód stb.)

Támogatott típusok: Skalárok:

- String (szöveg)
- Number (szám)
- Boolean (bool változó)
- Date-like (dátum, avagy date, time és date-time)

Tárolók:

- Hash (szótár)
- Sequence (sorozat)
- Collection (egyszerű kollekció)

Szubrutinok:

- Methods and functions
- User-defined directives (macros)

Miscellaneous/seldom used:

- Node
- Markup output

A sablon tartalmazhat:

- *Szöveget*
- *Interpolációt (mezőket):* ${ ... }
- *Tag-eket: <# .../#>*

[6] `https://freemarker.apache.org/docs/index.html`

- *Kommenteket:* <#–

Flag nem lehet flagben és interpolációban sem. Ingorálja a fölösleges white space -eket a tagban. Vannak benne rangek. Lehet benne a stringet darabolni.

Konkatenálás + jellel.

Van benne if:

- *or:* ||
- *and:* &&
- *not:* !

Null esetén lehet default értékeket beállítani ezek segítségével: unsafe_expr!default_expr or unsafe_expr! or (unsafe_expr)!default_expr or (unsafe_expr)! Tesztelő operátorok unsafe_expr?? or (unsafe_expr)?? boolean -t ad vissza.

Assign <#assign x + = y>

2.7. Mustache[7]

A Mustache az egyik legtöbb forrással rendelkező templater könyvtár. Tömbnyire HTML manipulációra használják. Elérhető többek között Ruby, JavaScript, Python, PHP, Perl, Objective-C, Java, C# /.NET, Android, C++, Go, Lua, Scala, Delphi, R, C nyelveken. A 46 nyelv teljes listája elérhető a hivatalos GitHub oldalukon. A Kotlin ugyan még nem szerepel benne, de ami Java alatt elérhető, az Kotlin alatt is. Jól működik többek között a TextMate, Vim és Emacs szövegszerkesztőkkel.

Ún. `logic-less template`-eket használ, vagyis elkülöníti a logikát és a a sablonokat. A mezőket dupla kapcsos zárójelekkel jelöli, például *Dear {{ name }}*. A mezőkre `tag`-ként hivatkozunk esetében. A `name` a tag kulcsa. Ezt a kulcsot fogom használni a későbbi példáimban is.

A template-be kulcs-érték párokat lehet beszúrni. Minden tag-be az ő nevéhez tartozó adat szúródik be.

Több fajtája létezik a tag-eknek:

- *Variables:* Ezek a legalapvetőbb tag-ek. Amikor a template `name` kulcsú tag-jébe szúrunk be, akkor az összesbe szúrunk be. Alapvetően szöktetve vannak a HTML-ből. Például a *GitHub* helyett *& lt;b& gt;GitHub& lt;/b& gt;* kerül a kódba. Ha szeretnénk, hogy ne legyenek szöktetve, akkor *{{{ name }}}* vagy *{{& name }}* módra kell megadnunk őket. Például ha a *{{{name}}}* vagy *{{& name}}* mezőbe szúrjuk be a *GitHub* szöveget, akkor az *GitHub*ként is lesz beszúrva.

- *Sections:* Egy section tartalma egyszer vagy többször is megjelenhet. Elejét *{{# name }}* és végét *{{/ name }}* módon jelöljük. A kettő között egy sablonrészlet található. Egy boolean értéket vagy adatlistát lehet a `section` mezőbe szúrni. Ha false-t vagy üres listát kap, akkor nem jelenik meg. True esetén megjelenik egyszer, és a tagjeibe a kulcsukhoz tartozó adatot szúrja be. Nem üres lista esetén a belső sablonba a lista minden elemét beszúrja és az eredményeket egymás alá írja ki. Ha egy paraméter nélkül hívható objektumot adunk át neki, akkor azt meghívja és a kimenetét szúrja be a Section helyére. Ez lehet lambda kifejezés, funktor vagy function is.

[7] `https://Mustache.github.io`

- *Inverted Sections:* Elejét *{{∩ name }}* és végét *{{/ name }}* módon jelöljük. Akkor jelennek meg, ha a kulcsuk a kapott adatban nem létezik, false vagy üres lista. Ez például hiányzó kulcsok esetén alkalmas hibaüzenet vagy más visszajelzés megjelenítésére.

- *Comments:* Mustache-ben kommentet *{{! ignore me }}* formátumban írhatunk. Ezeket bárhol kezdhetjük a szövegben, de nem kerülhetnek tag-be.

- *Partials:* A *{{> name}}* formátumban adhatóak meg. Rekurzívak is lehetnek, ha nem tartalmaznak végtelen ciklust. Ezzel valósítja meg a Mustache az importálást. A fenti példa a name nevű Mustache fájl tartalmát jelöli.

- *Set Delimiter:* Ezzel a funkcióval új jelölés vezethető be a dupla kapcsos zárójelek helyett. Például a *{{=<% %>=}}* a *{{ name }}* formátumot *<% name %>*-re cseréli. Ez hasonló módon vissza is állítható: *<%={{ }}=%>* Ez a Mustache megoldása arra, hogy váratlan szintaxisú környezetben is jól használható legyen.

Összegezve, a Mustache egy egyszerű, kreatív és egyszerű templater engine. Ugyanakkor külső vezérlést igényel működéséhez és HTML-ekre van szakosodva. Nincs szöktetés, csak a tag-ek jelölése állítható dinamikusan. Ez ugyanakkor egy olyan képesség, amivel a `Lileto` nem rendelkezik (még).

3. fejezet

A Lileto nyelv[1]

A `Lileto` egy letisztult nyelv, összesen öt példányosítható típust és nyolc műveletet tartalmaz. Egységes és csak erősen indokolt esetekben tér el a természetes, bevett megoldásoktól. A továbbiakban a szintaxist, típusokat és műveleteket összetettség szerinti sorrendben adom meg.

- Az első alfejezetben a szintaxis alapjait mutatom be.
- A második alfejezetben a típusok közös tulajdonságait mutatom be.
- A harmadik alfejezetben a sablonok szintaxisáról és alkotóelemeiről írok.
- A negyedik alfejezetben bemutatom a műveleteket és egyszerű eseteiket.
- Az ötödik alfejezetben mutatom be a tároló típust és a műveletek komplexebb eseteit.
- Végezetül pedig a hatodik alfejezetben bemutatom az importálást.

3.1. Alapvető szintaxis

A szintaxis terén volt a legnehezebb megtalálni az egyensúlyt a tömörség, az átláthatóság és a tanulhatóság között. Fontos volt, hogy bevett és vagy következetes jelöléseket használjunk, hogy könnyű legyen őket megtanulni. De az átláthatósághoz az is fontos volt, hogy a jelölések egyediek legyenek és így felismerhetőek az alárendelt nyelv kódjában. Éppen ezért a legtöbb sablon leíró és kezelő nyelvhez hasonlóan a `Lileto`-ban is mindent, ami nem az alárendelt nyelv kódja, azt speciális zárójelekbe zárjuk. Minden, ami azon kívül van az az alárendelt nyelv része. A Lileto nyelvben minden, ami a nyelv speciális zárójelein kívül van, az az alárendelt nyelv része. Minden mást speciális zárójelekbe kell zárni, amikhez egy általános szintaxis is tartozik. Persze ettől a szintaxistól minden konkrét zárójel típus eltér a maga módján. Minden speciális zárójelet kapcsos zárójel nyit és zár. A folyó szövegben ezeket a kapcsos zárójeleket lehet szöktetni “${“ és “$}” módokon. A nyelvben jelenleg a ‘$’ karakterrel lehet szöktetni, amit “$$” módon szöktetünk.

Általános szintaxis:

```
{M =name head | body | tail outp= M}
```

- *M:* A zárójel egy karakteres jele, ami a zárójel típusát jelöli. A zárójel eleji jel kötelező, de a zárójel végi sok zárójel típus esetében elhagyható.

[1] `https://github.com/Hollo1996/LILETO`

- *name:* A zárójeleknek lehet nevet adni, ami alapján bárhol meghivatkozhatók. Ez a név latin betűvel vagy alulvonással kell kezdődjön, és latin betűket, alulvonást és számokat tartalmazhat. Egyes zárójelek ezután a fájlon belül globálisan elérhetőek lesznek az adott névvel. Más zárójelek az őket körülfogó zárójelhez fognak tartozni. A name elhagyása nem minősül szintaktikai hibának.
- *head:* A zárójel fejléce. Itt lehet a zárójel tartalmának formátumára vonatkozó és más a zárójelre vonatkozó metaadatokat megadni. Elhagyása nem minősül szintaktikai hibának.
- *body:* Itt lehet a zárójel tartalmát megadni. Ennek szintaxisa tér el a legjobban az egyes zárójel típusok között.
- *tail:* Még nem használjuk! A zárójel kimenetére vonatkozó metaadatokat lehet majd benne definiálni.
- *outp:* A zárójel kimenetének neve. Érvényesek rá a name-re vonatkozó formai megkötések. Változó létrehozására való zárójelek esetén a name határozza meg a változó visszatérési értékének nevét, így nincs erre a változóra szükség. Minden zárójel esetén elhagyható.

head, body vagy tail elhagyható, ha nem tartalmazna értéket:

```
{M =name head | body | tail outp= M}  {. tail-t még nem használt .}
{M =name | body | tail outp= M}  {. tail-t még nem használt .}
{M =name head | | tail outp= M}  {. tail-t még nem használt .}
{M =name head | body outp= M}
{M =name | tail outp= M}  {. tail-t még nem használt .}
{M =name body outp= M}
{M =name head | outp= M}
{M =name outp= M}
```

Zárójel típusok:

- *TEXT(szöveg):*

  ```
  {" text "}
  ```

  - *M:* ”
  - *name:* nincs
  - *head:* nincs
  - *body:* maga a szöveg
  - *tail:* nincs
  - *outp:* nincs

- *SPEC(speciális karakter):*

  ```
  {$ code code … $}
  ```

  - *M:* $ a zárójel végi elhagyható
  - *name:* nincs
  - *head:* nincs

Név	Kód	Karakter
enter	e	
tabulátor	t	
szóköz	s	
nyitó kerek zárójel	orb	(
csukó kerek zárójel	crb	)
egyenlőségjel	eq	=
pluszjel	pl	+
nyitó kapcsos zárójel	ocb	{
záró kapcsos zárójel	ccb	}
pont	pe	.
vessző	co	,
pontosvessző	sc	;
kettőspont	cl	:
függőleges vonás	pi	|
kisebb	ls	<
nagyobb	gt	>
kérdőjel	qm	?
felkiáltó jel	dm	!

3.1. ábra. speciális karakterek Lileto kódjai

  - *body:* A karakterek nyelv specifikus kódjainak felsorolása. Lehet őket vesszővel tagolni, de nem kötelező. A kódok elé számot írva egyszerre több jön létre ugyanabból a karakterből. A megadott karaktereket szöveggé fűzve adja vissza. A whitespace karakterek így kezelhetőek a leginkább átláthatóan. Az utf8 karakterek többsége elérhető a kódjukkal. Jelenleg még a következő kódok működnek: **??**
  - *tail:* nincs
  - *outp:* nincs

- *TEMP(sablon):*

```
{< =name text {" ... "} {$ … $} {< ... >} … >}
```

  - *M:* < and >
  - *name:* adható, de legtöbb esetben nem kötelező
  - *head:* nincs
  - *body:* Tartalmazhat szöveget, TEXT, SPEC és további TEMP zárójeleket. Ekkor a beágyazott TEMP-ek jelölik a sablon mezőit. Tartalmuk nem jelenik meg a Temp típusú változó kiíratásánál, ahogy ők maguk sem. Ha viszont adatot szúrunk beléjük, akkor abból szöveg generálódik, ami mögéjük kerül az adott sablonban. A beágyazott TEMP-eknek kötelező nevet adni. Ezek a nevek a körbefogó Temp zárójelhez fognak tartozni. Az egymást követő szövegeket (ebbe beleértve a SPEC zárójelek kimenetét is) egy szöveggé fogja olvasztani, hogy csökkentse a kimenet méretét.
  - *tail:* nincs
  - *outp:* nincs

- *CONT(tároló):*

  ```
  {[ =name path :type = value… | path :type = value ... ]}
  ```

  - *M:* [and]
  - *name:* Adható, de nem kötelező
  - *head:* Adható. Azonos a formátuma a body-val. Definiálásával egy táblázat fejlécét adjuk meg.
  - *body:* Egy mezőnek 3 tulajdonság adható meg:
    - *name:* A mező kulcsa, ami csak a name formátumának megfelelő szöveg lehet. Head esetén az oszlop nevét jelöli. Ha az érték mezőnek és oszlopának is van neve, akkor azok nem különbözhetnek. Ugyanakkor az eltérés jelezheti azt is, hogy az érték mezőnek nem az az oszlopa.
    - *value:* A változó értéke. Akármilyen értéket visszaadó zárójel vagy valamilyen érték állhat a helyén. Head esetén az oszlop default értékét adja meg.
    - *type:* Megköthető a mező típusa. Ekkor az értéket le ellenőrzi, hogy megfelele a megadott típus megkötésének. Táblázat típus esetén megadható a táblázat fejléce is, így annak újradefiniálására nincs szükség az érték mezőben. Head esetén itt adható meg az oszlopnak típus megkötés.
    - Egyik definiálása sem kötelező! Nem vagyunk kötelesek semelyik mezőhöz vagy oszlophoz sem nevet, típus megkötést vagy értéket rendelni. Ha viszont egy oszlopnak már adunk nevet, akkor minden elemnek is kell adnunk, ami név nélküli oszlophoz tartozik. Ha nem kapnak értéket, akkor a nyelv null értékét (jele: -) vagy ha van definiálva, akkor az oszlop default értékét veszi fel. Az egyes mezők vesszővel tagolhatók.
  - *tail:* nincs
  - *outp:* nincs

- *COMM(parancs):*

  ```
  {! =name expr expr … outp= !}
  ```

  - *M:* !, A zárójel végi elhagyható
  - *name:* A parancs neve. Később ezen keresztül lesz elérhető mint függvény. Jelenleg, ha az alárendelt nyelvbe van ágyazva, akkor nem írja ki a kimeneti értékét, hanem változót készít belőle ezen a néven.
  - *head:* nincs
  - *body:* Ide kerülnek maguk a parancsok. A parancsokat nem vagyunk kötelesek, de elválaszthatjuk pontosvesszővel vagy vesszővel. Parancs lehet:
    - *Lileto bracket:* Akármilyen másik Lileto speciális zárójel.
    - *declaration(változó létrehozás): name :type = expr* Új változó létrehozása értékadással. Az értéket vagy a várt típusra konvertálja vagy hibát dob. Ha a típus nincs definiálva, akkor átveszi a változó típusát. A type hiánya esetén a := jól megkülönbözteti a mezei értékadástól.
    - *bracket(zárójel): (expr)* A kifejezés visszatérési értékét módosítás nélkül téríti vissza. Csak a műveleti sorrend egyértelmű megadására használjuk.

        * *insert(beszúrás): expr < expr* A bal és jobboldali változó a kifejezések visszatérített változóinak mély másolata. Lecseréli a baloldali változó megfelelő változóit a jobboldali változó megfelelő változóira.
        * *addition(összeadás): expr + expr* A bal és jobboldali változó a kifejezések visszatérített változóinak mély másolata. Hozzáfűzi a baloldali változó megfelelő értékeihez a jobboldali változó megfelelő értékeit.
        * *equation(értékadás): variable = expr* A jobboldali változó a kifejezések visszatérített változóinak mély másolata, de a baloldali nem másolódik. Lecseréli a baloldali változó megfelelő értékeit a jobboldali változó megfelelő értékeit. Az outp nevű változónak értéket adva is megadhatjuk a visszatérített értéket.
    - *tail:* nincs
    - *outp:* A visszatérített változó neve. Bármilyen már a fájl-hoz tartozó változó neve megadható. Ha ez definiálva van, akkor ez lesz a változó lesz a visszatérési érték.

- *COMT(komment):*

  ```
  {. text .}
  ```

    - *M:* .
    - *name:* nincs
    - *head:* nincs
    - *body:* folyó szöveg, de nincs semmilyen hatása
    - *tail:* nincs
    - *outp:* nincs

3.2. Típusok közös tulajdonságai

A `Lileto` nyelvben nincsen öröklődés. Nincs rá szükség, mivel még függvények és típus deklarációk sincsenek. Minden egyes változó rendelkezik a következő attribútumokkal:

- *cont(tartalom):* A változó nem primitív tartalma. Egy lista, ami más változókat tárol.
- *rest(megkötés):* A változó nem primitív tartalmára vonatkozó megkötések. A változó kötelező attribútum neveit és típus megkötéseit és default értékeit tárolja.
- *text(szöveg):* A változó által közvetlen tárolt egybefüggő szöveg. Ebben tárolják a változók közvetlen szöveges tartalmukat.

Ezeket ugyanakkor közvetlen sosem érjük el. Fontos tudnunk, hogy indexelésnél valójában mindig a cont listát indexeljük, és a rest felelős azért, hogy a változónk jól formázott legyen. A text attribútum a nagy kivétel, ami minden változóból elérhető, de ha a változó nem használja, akkor valójában a cont tartalmát fogjuk visszakapni szöveggé alakítva. A legprimitívebb típus a szöveg és minden más közvetlen abból épül fel:

- *Text (szöveg):* Folyó szöveg. A TEXT és SPEC zárójelek is ilyen típusú változót térítenek vissza. Ez a típus tekinthető a nyelv primitív típusának.
    - *cont:* Mindig üres.
    - *rest:* Mindig üres.

  - *text:* Ebben tárolja a változó az értékét.

- *Temp (sablon):* Folyó szöveg. A TEMP zárójelek ilyen típusú változót térítenek vissza. Felfogható egy Text-eket tároló speciális faként, ami minden csúcsban eggyel több adatot tárolhat, mint ahány gyereke van az adott csúcsnak. Ezeket a szövegeket ugyanis be is rendezi a gyerekei elé és mögé.
  - *cont:* A változó tartalma. Text és Temp változókat tárolhat benne. Minden Temp-nek neve is kell legyen!
  - *rest:* Mindig üres, mert a megkötéseket a típus maga már meghatározza.
  - *text:* Mindig üres.

- *Cont (tároló):* A nyelv tároló osztályai. A CONT zárójelek ilyen típusú változót térítenek vissza.
  - *cont:* A változó tartalma. Akármilyen típust tárolhat, de meg kell feleljen a megkötéseknek. Ha vannak megkötések, akkor a rest hosszával osztható sok elemet kell tárolnia. Ekkor annyi sorból áll, ahányszor osztható a hossza a rest hosszával. Minden sorában csak egyedi nevek lehetnek.
  - *rest:* A változó tartalmára vonatkozó megkötések. Akármilyen változót tárolhat. Ezeknek neve és típus megkötése vonatkozik majd a cont megfelelő változóira is és értéke pedig a cont megfelelő változóinak default értéke.
  - *text:* Mindig üres

- *Comm (parancs):* A parancsok kimenete. Később ez foglya helyettesíteni a függvényeket. A COMM zárójelek ilyen típusú változót térítenek vissza.
  - *cont:* A változó tartalma. A parancs zárójelben létrehozott változókat tárolja el. Akármilyen típust eltárolhat.
  - *rest:* Mindig üres.
  - *text:* Mindig üres

- *File (fájl):* Az importált fájlok adatai. Nincs zárójele. A COMM zárójelben lehet példányosítani a fájlhoz vezető path szerint.
  - *cont:* A változó tartalma. Ide kerülnek a fájlban létrehozott globális változók.
  - *rest:* Mindig üres.
  - *text:* A fájl által generált szöveg.

- *Null (null):* Nem példányosítható típus. A nyelv null értéke. Ha egy változó értéket kap, akkor automatikusan felülíródik. Éppen ezért csak érték nélküli változók vehetik föl.
  - *cont:* Mindig üres.
  - *rest:* Mindig üres.
  - *text:* Mindig üres.

Struktúrák helyett a tároló típust használjuk, ami sokoldalú működésével könnyen kivált minden standard tároló típust. (lásd 3.5) A tároló sokoldalúságának köszönhető az is, hogy csupán öt példányosítható típusra van szükségünk. Ennek köszönhetően lesz a fordítás is gyors.

3.3. Sablonok

Más alapvető sablon nyelvekben a sablonok mezőkből és szövegből állnak. A `Lilelto`-ban viszont ennél összetettebb a helyzet. Nem csak, hogy a sablon deklarációnál speciális karaktereket is könnyen definiálhatunk, de a mezők maguk is sablonok, ezzel számtalan praktikus megoldást lehetővé téve.

Először tekintsük a szintaxist:

```
{<
text {$ … $} {" ... "} {< =slot1 ... >} …
text {$ … $} {" ... "} {< =slot2 ... >} …
text {$ … $} {" ... "} {< =slot3 ... >} …
>}
```

Mint azt már korábban is leírtam, a `Lileto` sablonokban háromféleképpen is lehet szöveget deklarálni. Az első legegyszerűbb módja ennek a külső szöveg, ami minden a sablonban használható `Lileto` speciális zárójelen kívül eső karakter. Fontos tudni, hogy a sor végi és eleji és sortörő white space karakterek ebbe nem tartoznak bele! Ezeket a fordító figyelmen kívül hagyja, hogy a zárójeleket szabadon lehessen tördelni. A második a SPEC zárójelek, amikkel a sor eleji és végi white space karakterek is könnyen deklarálhatók kódneveikkel. A SPEC zárójelekben a fordító minden a white space karaktert figyelmen kívül hagy. A harmadik pedig a TEXT zárójelek, amik tartalmának első és utolsó white space karakterét leszámítva minden karaktere az általa deklarált szöveg részét képezi. A sor elválasztó White space karakterek ezzel is kezelhetőek. Mind a külső szövegben, mind a szöveg zárójelben használhatók a ”${”, ”$}” és ”$$” szöktetett karakterek. A deklaráció után a szomszédosan deklarált szövegeket egyesíti (konkatenálja) a fordító, ahogy minden egyes sablonművelet után is. Például tekintsük a fenti példa egy sorában a sor elején háromféleképpen definiált három szöveget. Ezeket a létrejövő Temp változó egyetlen egy szövegként fogja eltárolni. A sablon mezőinek pedig kötelező nevet adni, hogy később könnyen lehessen rájuk hivatkozni.

Ezek a mezők egy egyszerű logikát követnek, ami a mustache nyelv (lásd 2.7) megoldásának továbbgondolása: Vegyük például a

```
{< {< =greeting Dear {< =title >} {< =first >} {< =second >} >} ... >}
```

sablont, amibe a `Lileto` speciális zárójelét ágyaztuk.

- Ha a `greeting` mezőbe nem szúrunk adatot, akkor tartalma meg sem jelenik a kimeneten:

  ```
  input:
  output: " ... "
  ```

- Ha a `greeting` mezőbe szöveget vagy sablont szúrunk, akkor a mező helyén a beszúrt szöveg vagy sablon tartalma fog megjelenni:

  ```
  input: {" text "} v. {< text >}
  output: " text ... "
  ```

- Ha a `greeting` mezőbe null értéket szúrunk, akkor a mező tartalma fog megjelenni, aminek mezői nem fognak megjelenni a kimeneten:

  ```
  input: null
  output: " Dear    ... "
  ```

- Ha a `greeting` mezőbe egy kulcsokkal rendelkező tárolót szúrunk, akkor az egyes mezőibe be fogja szúrni a mezők nevével mint kulccsal rendelkező adatokat:

```
input: {[ first:={" Béla "} second:={"Kovács"} ]}
output: " Dear  Béla Kovács ... "
```

- Ha a `greeting` mezőbe egy kulcs nélküli tárolót szúrunk, akkor minden elemét külön külön be fogja szúrni az eredeti mezőbe, azok kimenetét közvetlen egymás után illesztve:

```
input:
{[
{[ first:={" Béla "} second:={" Kovács "} ]}
{[ title:={" Mr. "} first:={" Attila "} second:={" Hollósi "} ]}
{[ title:={" Dr. "} second:={" Egresi "} ]}
]}
output: " Dear  Béla KovácsDear Mr. Attila HollósiDear Dr.  Egresi ... "
```

- Utóbbi három esethez azt is fontos megjegyeznem, hogy a beszúrás kimenete is sablon, tehát Temp típusú. A feljebb leírt kimenetek viszont a valódi kimenetek szöveggé alakítva. Így hát az utóbbi három eset valós kimenete:

```
output1: " Dear {< =title >} {< =first >} {< =second >} ... "
output2: " Dear {< =title >} Béla Kovács ... "
output3: " Dear {< =title >} Béla KovácsDear Mr. Attila HollósiDear Dr. {< =first
```

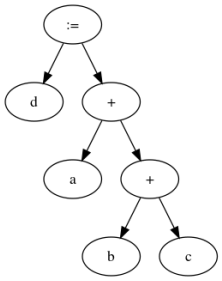
Ezek a sokoldalúság adják a `Lileto` egyik fő erősségét, mivel a sablon és az adat deklarációja után egyetlen egy beszúrással megkaphatjuk a kimenetet.

3.4. Parancsok

A legtöbb sablonokkal foglalkozó nyelvben csak a sablonok szintaxisára van szükség, mert a sablonok kezelését már egy magasabb rendű nyelv végzi. A `Lileto`-nak viszont szüksége van egy saját szintaxisra sablon kezelésre. A sokoldalú sablonoknak és tárolóknak köszönhetően ez már egyszerű. Négy műveletünk van összesen.

- Egy változó létrehozás, avagy `deklaráció`, amivel új néven lehet változót létrehozni;
- egy érték módosítás, avagy `értékadás`, amivel egy már létező változó értékét lehet módosítani;
- egy érték egyesítés, avagy `összeadás`, amivel két változó tartalmát lehet egymás mögé fűzni egy új változóba
- és egy érték összefűzés, avagy `beszúrás`, amivel egy változó értékei közé lehet beszúrni egy másik változó értékeit egy új változóban.

A `Lileto`-ban ezeket a COMM zárójelekben tehetjük meg. Mivel a nyelvben az összeadás és beszúrás is új változót hoz létre, nem pedig a baloldali változót módosítja, kénytelenek vagyunk a két művelet kimenetét egy új vagy régi változóba elmenteni. (a += b helyett a = a + b vagy c:= a + b) Ebből adódóan a COMM zárójel sorainak deklarációval vagy értékadással kell kezdődnie. A sorokat elválaszthatjuk pontosvesszővel, de ha attól függetlenül is egyértelmű a szintaxis, akkor erre nincs szükség.

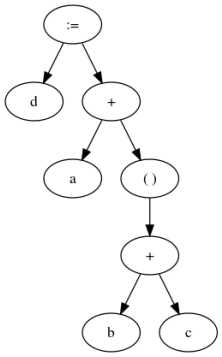


3.2. ábra. Két összeadás műveleti sorrendje.

A műveletek mindig jobbról balra értékelődnek ki. A műveleti sorrendet zárójelezéssel lehet egyértelműsíteni:

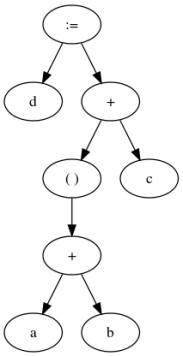
d := a + b + c Műveleti sorrendet lásd: 3.2

d := a + (b + c) Műveleti sorrendet lásd: 3.3



3.3. ábra. Ugyanaz a végrehajtási sorrend.

d := (a + b) + c Műveleti sorrendet lásd: 3.4

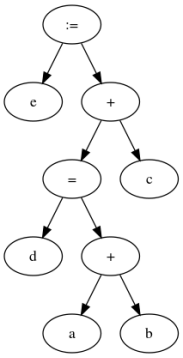


3.4. ábra. Módosított műveleti sorrend.

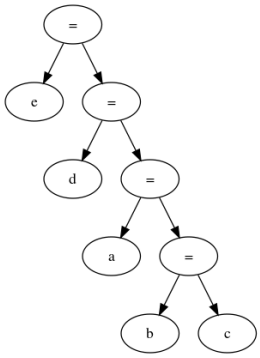
A sor elejét nem zárójelezhetjük, mert annak kell utoljára kiértékelődni, hogy a művelet kimenete mentésre kerüljön. Deklaráció csak a sor legelején szerepelhet, de értékadás szerepelhet a sorban bárhol.

e := (d = a + b) + c Műveleti sorrendet lásd: 3.5 e = d = a = b = c Műveleti sorrendet lásd: 3.6 Ekkor a értékéről minden értékadásnál másolat készül

A COMM zárójeleknek van visszatérési értéke. Ha nem adtunk nevet a zárójelnek, akkor az outp-ben megadott változót fogja visszatéríteni. Ha itt nem definiáltunk semmit, akkor null értéket ad vissza. Ha adtunk nevet a zárójelnek, akkor magát téríti vissza egy Comm típusú változó ként.



3.5. ábra. Az értékadás nem kell, hogy az egyenlet bal oldalán legyen.



3.6. ábra. A négyszeres egyenlőség végrehajtási sorrendje.

3.4.1. Deklaráció

`Lileto`-ban egy változót így deklarálhatunk:

```
name :type = value
```

Itt az egyes adatok:

- *name:* A változó neve. Vonatkozik rá minden a zárójelek nevére vonatkozó megkötés. (lásd 3.1)
- *type:* A változó típusa. Cont változó esetén itt lehet a tároló fejlécére is megkötést tenni. (lásd 3.5.1)
- *value:* A változó értéke. Értéket bármivel megadhatunk, ami térít vissza értéket. Itt lehet szó műveletről, `Lileto` speciális zárójelről vagy egy már létrehozott változó elérési útjáról. Egy változó elérési útját mindig ponttal tagoljuk. A változókat így lehet indexelni is. Ekkor az indexet is úgy kezeljük, mint ha egy attribútum név lenne. Azok a változók is indexelhetőek, amik nem használják a cont (content, lásd 3.2) tulajdonságukat, csak ilyenkor üresnek számítanak. A változók cont-jából minden névvel rendelkező változó elérhető a változóból név szerint is és index szerint is. Így például a ”temp” sablon ”slot” mezőjét is elérjük ”temp.slot”-ként. A változók szövege elérhető a változóból text néven. Ilyen attribútummal nem rendelkező változók esetén a cont tartalma alakul szöveggé.

COMM zárójelben a típus megkötést el lehet hagyni, de egyes zárójelekben (pl. a CONT zárójelekben) a többi sem kötelező. Ha a típust nem definiáltuk, akkor az az értékből lesz meghatározva.

A típus megkötés hiánya esetén a szintaxis így módosul:

```
name := value
```

Egyszerre több változót is létrehozhatunk ugyanazzal a típus megkötéssel, így:

```
( name:=value, name:=value, name:=value, name:=value ):type
```

Egyszerre több változót is létrehozhatunk ugyanazzal az értékkel, így:

```
( name:type, name:type, name:type, name:type ) = value
```

Ekkor minden változó egy másolatot fog eltárolni az értékről. Mindkét esetben a vessző elhagyható.

3.4.2. Értékadás

Az értékadást eleinte könnyű összetéveszteni a deklarációval, pedig közel sem ugyanarról van szó. A deklaráció új változót hoz létre, míg az értékadás egy már létező változót módosít. Ha értékadásnál a baloldali változót a nevével adjuk meg, mert közvetlen elérjük, akkor a szintaxisa is nagyon hasonló a típus megkötés nélküli deklarációhoz. Ekkor leginkább a hiányzó ':'-ból lehet megkülönböztetni az értékadást a deklarációtól, ami a rejtett típus meghatározásra utal.

```
c := (a + b) {. deklaráció .}
c = (a + b)  {. értékadás .}
```

Az értékadás a baloldali változóba másolja a jobboldali változó cont és text értékét. Mivel ekkor már a baloldali változó típussal rendelkezik, a típus formai megkötéseinek meg kell felelnie a jobboldali értéknek. A deklarációval ellentétben az értékadás nem kell, hogy a sor legelején legyen. A kapott érték másolatát vissza is téríti.

Mint minden műveletnek, ennek is vannak speciális esetei.

Például sablonnak adhatunk értékül szöveget.

```
temp = {" ... "}
```

Ekkor a sablon egész tartalmát annak az egy szövegnek a másolatára cseréli. Ez a sablon mezőkre is működik.

```
temp.slot = {" ... "}
```

Ilyenkor viszont az egész sablon másolatát téríti vissza.

Szövegnek ugyanakkor nem adhatunk értékül sablont. Kötelesek vagyunk a sablont szöveggé alakítani.

```
text = temp.text
```

3.4.3. Hozzáadás

A hozzáadás jele természetesen a '+' karakter.

```
value1 + value2
```

A hozzáadás két azonos típusú változó esetén a két változó tartalmának és szövegének másolatát egyesíti egy új változóban. Viszont két különböző típusú változó esetén is vannak elfogadott esetek.

Ha sablonhoz adunk szöveget, akkor egy másolat keletkezik a végén a szöveggel.

```
temp2 := temp1 + text
```

Ugyanígy sablon mező esetén is, de ekkor az egész sablonról készül a másolat.

Sablon végére hiába tartanánk logikusnak, nem így fűzünk mezőt:

```
temp2 := temp1 + slot {. a slot egy névvel rendelkező Temp típusú változó .}
```

A mezőt egy sablonba csomagolva kell hozzáadnunk a sablonhoz:

```
temp2 := temp1 + temp3 {. a temp3 tartalmazza a slot mezőt .}
```

e

3.4.4. Beszúrás

A beszúrás jele a '<' karakter, ami egy nyílra emlékeztet.

```
value1 < value2
```

Ez az a beszúrás, amiről a sablonok esetében már szó esett(lásd 3). A jobboldali változó tartalmát és szövegét a baloldali másolata tartalmának és szövegének helyére másolja. Ennek alapvetően nincs sok haszna, de speciális esetei adják a `Lileto` egyik erősségét.

- Ha egy sablon mezőbe szúrunk szöveget, akkor a mező a szövegre cserélődik a mezőt tartalmazó sablon másolatában.
- Ha sablon mezőbe szúrunk egy sablont, akkor a mező a másik sablon tartalmára cserélődik a mezőt tartalmazó sablon másolatában.

Ha egy mezőbe ágyazott mezőbe szúrunk szöveget, akkor több lépésben hajtódik végre a művelet. Először a belső mező cserélődik le a beszúrt szövegre. Másodszor pedig a külső mező lecserélődik a saját tartalmára.

Ha ennél többszörös a mezőbe ágyazás (pl.: mezőben mező, amiben mező), akkor hasonló a működés. Először a belső mező cserélődik le a beszúrt szövegre. Ezt követően pedig belülről kifelé minden mező lecserélődik a saját tartalmára.

A műveletet követően minden szomszédos szöveg egyesül.

A művelet azért fejti ki minden mező tartalmát, mert másképp nem jelenne meg a tartalmuk, amikor a sablont szöveggé alakítjuk. Azt feltételezi a fordító, hogy ha a programozó értéket szúr a sablon egy mezőjébe, akkor annak kimenetét meg is szeretné jeleníteni.

Ha viszont igazán komoly eredményeket szeretnénk elérni, akkor nem egy adatot szeretnénk egyszerre beszúrni. Itt lesznek segítségünkre a tárolók, amik segítségével felsorolásokat és sablon tömböt is tudunk készíteni egyetlen beszúrással!

3.5. Tárolók

A tárolók legbővebb szintaxisa:

```
{[
name :type = value , name :type = value … {. header .}
|
name :type = value , name :type = value … ; {. line1 .}
name :type = value , name :type = value … ; {. line2 .}
name :type = value , name :type = value ... {. line3 .}
]}
```

A tároló legbővebb formája egy táblázat. Itt a '|' karakter előtti sor a táblázat fejléce. Ennek elemei a ','-vel elválasztott fejléc mezők. Minden mező az adott oszlop nevét, típusát és default értékét határozza meg. A '|' karakter utáni elemek a táblázat értéksorai. Ennek elemei a ','-vel elválasztott értékmezők. Az értékmezők a táblázat értékeit, azok típusát és nevét határozzák meg. Soronként a neveknek egyedinek kell lennie (a fejlécben is). A név a mezőnek a soron belüli kulcsa. Egy mezőnek sem vagyunk kötelesek se nevet se típust se értéket adni. A név, a típus és az érték hiánya esetén a szintaxis így módosul:

```
name :type = value
name:type
name:
:type
name := value
:type = value
value
```

Ha se az értéket se a típust nem definiáltuk, akkor a változó null értékű és Null típusú lesz. Ha egy teljesen üres mezőt kívánunk definiálni, azt a '-' karakterrel tehetjük meg. Ha egy oszlopnak van neve, akkor az oszlophoz tartozó minden értékmezőnek az a neve. A név ütközés szemantikai hibának számít! Ha egy oszlopnak van típus megkötése, akkor az oszlophoz tartozó minden mezőnek csak olyan típusú vagy null értéke lehet és csak olyan típusú lehet a típus megkötése. Ha egy oszlopnak adtunk nevet, akkor minden név nélküli oszlophoz tartozó érték mezőnek kötelesek vagyunk nevet adni. Egy oszlophoz viszont nem mindig az az értékmező tartozik egy sorból, ami alatta van. Ha vannak nevek a táblázatban, akkor egy értékmező pozícióját azzal is meghatározhatjuk, hogy az adott oszlop nevét adjuk neki. Az ilyen névvel rendelkező érték mezők a soruk végére kell kerüljenek. Egy sornak nem kell minden oszlopban értéket tartalmaznia. Azok az értékmezők, amik nem lettek definiálva egyszerűen az oszlop default értékét, nevét és típus megkötését öröklik. Az érték sorokat egy oszlop esetén ','-vel választjuk el. A ',' karakterek mindig elhagyhatók.

Ez a bonyolult szintaxis teszi lehetővé, hogy minden standard tárolót definiálni lehessen a CONT zárójellel:

A generikus lista csupán egy fejléc nélküli egysoros táblázat kulcs nélküli mezőkkel:

```
{[ :type = value , :type = value … ]}
```

A típusos lista egy egy oszlopos táblázat, ahol a táblázat egy mezőjének sincsen kulcsa:

```
{[ :type | value , value … ]}
```

A generikus map(dictionary) csupán egy fejléc nélküli egysoros táblázat kulcsos mezőkkel:

```
{[ name = value , name = value … ]}
```

A típusos map(dictionary) egy egy oszlopos táblázat kulcsos mezőkkel:

```
{[ :type | name = value , name = value … ]}
```

A mátrix csupán egy egy típusú kulcsok nélküli táblázat:

```
{[
( - - … ):type {. header .}
|
value value …  ; {. line1 .}
value value …  ; {. line2 .}
value value …  ; {. line3 .}
]}
```

Ebből is definiálhatunk generikusat:

```
{[
value value …  ; {. line1 .}
value value …  ; {. line2 .}
value value …  ; {. line3 .}
]}
```

A struktúra típus deklaráció csupán egy egy soros fejléc nélküli táblázat értékek nélkül:

```
{[ name:type , name:type … ]}
```

Ez utóbbit használhatjuk, amikor egy tároló megkötést definiálunk:

```
name :Cont{[ name:type , name:type … ]} = value
```

Ekkor a jobboldali érték egy tároló, ami fejlécének, ha van, akkor egyeznie kell a definiált fejléccel. Így olyan, mint ha egy struktúra példányokból álló listát definiáltunk volna.

A tárolók haszna, hogy műveletek esetén sokszor iterátorként kezeljük őket. Ezeket a helyzeteket többnyire az operátor típusa határozza meg és hogy annak melyik oldalán áll a tároló és annak nevekkel rendelkező, nevekkel nem rendelkező vagy nevekkel opcionálisan rendelkező változatárol van e szó.

3.5.1. Deklaráció Tárolókkal

Változó létrehozásánál érték ugyan csak az '=' karakter egyik oldalán állhat, de nem mindegy, hogy milyen típust várunk. Ennek alapján három esetünk van.

A legegyszerűbb eset, ha tároló az érték és tárolót várunk:

```
cont :Cont = {[ ... ]} {. CONT zárojel .}
cont :Cont = {! ... !} {. Cont típusú változót eredményül adó COMM zárojel .}
cont :Cont = ( a + b ) {. Cont típusú változót eredménül adó művelet .}
cont :Cont = c {. Cont típusú változó .}
```

Ekkor alapvetően nincs gond, de a típus megkötés megszabhatja a várt tároló fejlécét is:

```
cont :Cont{[ ... ]} = {[ ... ]}
```

Ekkor a jobboldali tároló nem tartalmazhat olyan oszlopot, ami nem része a megkötött fejlécnek. Még azt sem teheti meg, hogy a várt fejlécben definiálttal megegyező nevű és típusú mezőt vesz fel, de más default értékkel. Megeshet, hogy ekkor deklarálunk az értékben is oszlopokat, mert más oszlopnak nem tervezünk értéket adni. A legegyszerűbb persze az, ha a jobboldali zárójelnek nem adunk értéket, csak tagoljuk a sorait pontosvesszőkkel.

Ha tárolót várunk, de az érték nem tároló, akkor egy olyan tároló jön létre, aminek az az egy definiált (nem default) értéke.

```
cont :Cont = {< ... >} {. TEMP zárojel .}
cont :Cont = {" ... "} {. TEXT zárojel .}
cont :Cont = {$ ... $} {. SPEC zárojel .}
cont :Cont = {! ... !} {. nem Cont típusú változót eredményül adó COMM zárojel .}
cont :Cont = ( a + b ) {. nem Cont típusú változót eredménül adó művelet .}
cont :Cont = c     {. nem Cont típusú változó .}
```

Ha ez nem lehetséges a megkötött fejléc miatt, akkor a fordító hibát dob. Például, ha olyan tárolót várunk, aminek az első oszlopa nem egyezik az értékül adott típussal.

```
cont :Cont{[ ... ]} = {< ... >} {. TEMP zárojel .}
cont :Cont{[ ... ]} = {" ... "} {. TEXT zárojel .}
cont :Cont{[ ... ]} = {$ ... $} {. SPEC zárojel .}
cont :Cont{[ ... ]} = {! ... !} {. nem Cont típusú változót eredményül adó COMM zárojel
cont :Cont{[ ... ]} = ( a + b ) {. nem Cont típusú változót eredméynül adó művelet .}
cont :Cont{[ ... ]} = c  {. nem Cont típusú változó .}
```

Az utolsó eset pedig ha nem Cont típusú értéket vár, de Cont típusú a baloldali érték.

```
temp :Temp = {[ ... ]} {. Temp típus megkötés .}
text :Text = {[ ... ]} {. Text típus megkötés .}
comm :Comm = {[ ... ]} {. Comm típus megkötés .}
null :Null = {[ ... ]} {. Null típus megkötés .}
cont :File = {[ ... ]} {. File típus megkötés .}
```

Ekkor a tárolónak meg kell felelnie az egyes típusok formai követelményeinek:

- *Temp:* Nem lehet fejléce. Csak név nélküli szöveget tartalmazhat és névvel rendelkező Sablont tartalmazhat.
- *Text:* Nem lehet fejléce. Üres kell legyen.
- *Comm:* Nem lehet fejléce. Minden értékének kell legyen neve. A névnek egyedinek kell lennie.
- *Null:* Nem lehet fejléce és üres kell legyen.
- *File:* Nem lehet fejléce. Minden értékének kell legyen neve. A névnek egyedinek kell lennie.

3.5.2. Értékadás Tárolókkal

Az értékadás és a deklaráció közötti főbb különbségeket már tárgyaltuk. (lásd 3.5.2) Itt ezért már csak a tárolókat is érintő speciális esetekre térek ki.

Ha Cont típusú a jobboldali érték és a baloldali is, akkor nem kell felelni a baloldali érték fejlécének. Maga a fejléc ugyanis nem a tároló típusának a része, hanem az adatának. Másolásnál nem csak az értékek másolódnak, hanem a fejléc is felülíródik!

```
left: {[ :Text | value1 value2 value3 ]}
input: {[ :Temp | value4 value5 value6 ]}
output: {[ :Temp | value4 value5 value6 ]}
```

Ha Cont típusú a jobboldali változó, de a baloldali nem, akkor nem történik automatikus típus konverzió. Pár kivétellel a fordító hibát dob. Például sablonok esetén a névvel rendelkező értékmezőket értékül adja a mezőknek. Több azonos kulcsú érték esetén és több azonos nevű mező esetén minden mezőre és azzal egyező nevű adatra elvégzi az értékadást azok sorrendjében. Éppen ezért nem lehetséges egy sablon értékeit egy sablonnak megfelelően formázott tárolóval újradefiniálni.

```
sablon: {< Dear {< =first Béla >} {< =last Tóth>} >}
input: {[ {" My dearest  "} {< =first Lajos >} {< =last Szabó>} ]}
output: {< Dear {< =first Lajos >} {< =last Szabó>} >}
```

Ha a tárolóban nincsenek névvel rendelkező adatok, akkor a művelet nem módosítja a baloldali változót.

Ha Cont típusú a baloldali változó, de a jobb oldali nem, akkor a változó tartalma és megkötése fog másolódni.

```
bal: {[ {" My dearest  "} {< =first Lajos >} {< =last Szabó>} ]}
jobb: {< Dear {< =first Béla >} {< =last Tóth>} >}
output: {[ {" Dear "} first:={< Béla >} last:={< Tóth>} ]}

bal: {[ {" My dearest  "} {< =first Lajos >} {< =last Szabó>} ]}
jobb: {" Dear "}
output: {[  ]}
```

3.5.3. Hozzáadás Tárolókkal

- Ha a baloldali és jobboldali változó is Cont típusú, akkor az új tároló a baloldali táblázat fejlécét örökli.
  - Az új tároló a baloldali tároló adataival fog kezdődni. Ezt követően a jobb oldali tároló minden sorából kiszűrjük a fejlécnek megfelelő értékmezőket, és azokat egyesével az új tároló végére fűzzük.

    ```
    bal: {[ a:Text b:Temp | v1 v2; v3 v4 ]}
    jobb: {[ a:Text c:Temp | v3 v4; v5 v6 ]}
    output: {[  a:Text b:Temp | v1 v2; v3 v4; v3 null; v5 null ]}
    ```
  - Ha a jobboldali tároló egy sora nem tartalmaz megfelelő értéket, akkor a sort kihagyjuk.

    ```
    bal: {[ a:Text b:Temp | v1 v2; v3 v4 ]}
    jobb: {[ a: c:Temp | {""} v4; {<>} v6 ]}
    output: {[  a:Text b:Temp | v1 v2; v3 v4; {""} null; {<>} null ]}
    ```
  - Ha a jobb oldali tárolónak nincsen fejléce, akkor egy sornak tekintjük.

    ```
    bal: {[ a:Text b:Temp | v1 v2; v3 v4 ]}
    jobb: {[ a:=v5 v6 c:=v7 v8  ]}
    output: {[  a:Text b:Temp | v1 v2; v3 v4; v5 null ]}
    ```
  - Ebben az esetben, ha a baloldali tároló egyoszlopos, akkor érdemes a jobboldalit is azzá tenni egy egy széles ”null fejléccel”.

    ```
    bal: {[ :Text | v1 v2 v3 v4 ]}
    jobb: {[ {"a"} {<>} {"b"} {<>} {"c"} ]}
    output: {[  :Text | v1 v2 v3 v4 {"a"} ]}

    bal: {[ :Text | v1 v2 v3 v4 ]}
    jobb: {[ - | {"a"} {<>} {"b"} {<>} {"c"} ]}
    output: {[ :Text | v1 v2 v3 v4 {"a"} {"b"} {"c"} ]}
    ```
  - Ha a baloldali tárolónak nincs fejléce, akkor a jobboldalinak egyszerűen hozzáfűzzük az értékeit.

    ```
    bal: {[ v1 v2 v3 v4 ]}
    jobb: {[ a:Text b:Temp | v5 v6; v7 v8 ]}
    output: {[ v1 v2 v3 v4 a:Text=v5 b:Temp=v6 a:Text=v7 b:Temp=v8 ]}
    ```
- Ha a baloldali változó Cont típusú, de a jobboldali nem, akkor megpróbálja az adott változó másolatát a tároló másolata mögé fűzni.

  ```
  bal: {[ v1 v2 v3 v4 ]}
  jobb: v5
  output: {[ v1 v2 v3 v4 v5 ]}
  ```

```
bal: {[ :Temp | v1 v2 v3 v4 ]}
jobb: {< >}
output: {[ :Temp | v1 v2 v3 v4 {< >} ]}
```

Ha ez nem lehetséges, mert a jobboldali érték önmagában nem lehet a tároló egy sora, akkor hibát dob.

- Ha a baloldali változó nem tároló, de a jobb oldali igen, akkor a baloldali változó típusától függ a működés.
  - Ha a baloldali változó szöveg, akkor a tároló minden sora után készül egy másolat belőle. A tároló minden sorának minden Text típusú mezőjét hozzáfűzi a sorhoz tartozó másolathoz. A másolatokat egy fejléc nélküli tárolóban téríti vissza.

    ```
    bal: {"  "}
    jobb: {[ {"a"} {"b"} ; {"c"} {<  >} ]}
    output: {[ {" ab "} {" c "} ]}
    ```

  - Ha a baloldali változó egy sablon, akkor is minden sor után másolatokat készít. A sablonmásolatok minden mezőjére az adott névvel rendelkező szövegeket és sablonokat kigyűjti az ahhoz tartozó sorból. A szövegeket és a sablonok tartalmát sorban a mező tartalmának végére fűzi.

    ```
    bal: {< {< =a >} {< =b >} >}
    jobb: {[ a:={"a"} a:={"b"} ; {"c"} b:={< d{< e >} >} ]}
    output:
    {[
    {< {< =a ab >} {< =b >} >}
    {< {< =a >} {< =b d{< e >}>} >}
    ]}
    ```

3.5.4. Beszúrás Tárolókkal

A tárolók beszúrása a nyelv legfontosabb művelete. Ezzel készíthetjük el egy sorban a jól formált sablonból és tárolóból a kívánt kimentet.

- Ha bal és jobboldali változó is Cont típusú, akkor a jobboldali minden sora után másolatot készítünk a baloldaliból. A másolatok értékeit töröljük és hozzájuk adjuk a jobboldali megfelelő sorát. A másolatokat egy fejléc nélküli tömbben téríti vissza.

  ```
  left: {[ v1, v2 ]}
  right: {[ v3; v4 ]}
  output: {[ {[ v3 ]} {[ v4 ]} ]}
  ```

- Ha a bal oldali tároló, de a jobb oldali nem, akkor a baloldaliból másolatot készít. A másolat tartalmát törli és hozzáadja a jobboldalit.

  ```
  left: {[ a:Temp b:Text | v1, v2 ]}
  right: {< a >}
  output: {[ a:Temp b:Text | a:={< a >} b:=null ]}
  ```

  Ha ez nem lehetséges a tároló fejléce miatt, akkor hibát dob a fordító.

- Ha a baloldali változó egy sablon és a jobboldali egy tároló, akkor a tároló minden sora után másolat készül a sablonból. Ez alól kivételt jelentenek az egy oszlopos tárolók, amiket ebből a szempontból egysorosnak tekintünk. A sablon minden mezőjére kikeresi a sor utolsó a névvel rendelkező elemét és beszúrja a mezőbe, majd a mezőt lecseréli a mező tartalmára.

```
left: {< {< =a {< =b >}>} >}
right:
{[ a: b: |
{[ a: b: | {" a2 "} {" b2 "} ]}, {" b1 "} ;
{" a3 "} {" b3 "}
]}
output: {[ {< b2 >} {< a3 >} ]}
```

- Ha a baloldali változó egy sablon mezője, akkor is a sablonból készülnek másolatok. Ilyenkor úgy vesszük, mint ha sablonba szúrtunk volna a megfelelő fejlécű táblázatokba ágyazva. Ha a mező egyik ősében (őt tartalmazó mezőjében) a mezőn vagy annak ősén kívül más mező is áll, akkor az megszakítja a kifejtést.

```
left:
{< =temp {<=slot1
{<=slot2 {<=slot3 {<=slot4  >} >} >}
{<=slot5 >}
>} >}

right: {[ =data érték1 érték2 érték3]}
operation:  temp.slot1.slot2.slot3.slot4 < data
output: {< {<  = slot1 ertek1ertek2ertek3 {< =slot5 >} >} >}
```

A mellékletben látható egy példa, amiben egy egyszerű xml layout-ot tudtunk sokkal egyszerűbben kezelni a `Lileto` segítségével.

3.6. Importálás

`Lileto` nyelvben az importálás nem más, mint File típusú változók létrehozása. A File típusú változóknak nincs saját zárójele. COMM zárójelben lehet csak példányosítani a `Lileto` fájl elérési útjának szöveges megadásával.

```
file2 :File = {"C:\Users\Me\Documents\ExampleX.lileto"}
```

Ekkor először lefuttatja a fájlt, majd a publikus tagjainak másolatát a File típusú változó cont-jában, a szöveges kimenetet pedig a változó text-jében.

A `Lileto` tiltja a körkörös importálást! Jegyzi a fordítás alatt lévő fájlokat, és azok hierarchiáját, egy az elérési utakat tároló fába, és ha olyan gyermeket talál, aminek van vele egyező őse, akkor hibát dob.

A File típusú változó nem módosítható! Nem vesz részt értékadásban!

Ugyanakkor művelet bal oldalán sem állhat! Ahhoz, hogy az egész fájlt egy változó ként tudjuk kezelni, más típusú változóvá kell alakítanunk deklarációval.

```
cm :Cont = file2 {. fájl publikus változói .}
tx :Text = file2 {. fájl kimenete .}
tm :Temp =file2
```

Ekkor úgy veszi, mint ha egy tároló lenne vagy egy szöveg attól függően, hogy a létrehozott változó milyen típusú. Tároló esetén nem tudunk itt fejlécet megkötni! A fájl publikus változói viszont módosíthatók, de ezek csak másolatai az eredeti változóknak.

Egy változó, amint egy COMM zárójelben létrehozunk alapból nem publikus. A COMM zárójelnek nevet kell adni, hogy a változói kintről is elérhetőek legyenek.

```
egyik.lileto:
outerText
{! =c data = {[ ... ]} !}
outerText

masik.lileto:
outerText
{!
temp =  {< ... >}
egyik :File = {"C:\Users\Me\Documents\egyik.lileto"}
outp = temp + egyik.c.data
!}
outerText
```

4. fejezet

Turing teljesség

Egy új nyelv esetében az első kérdés mindig az, hogy mennyire komplex algoritmusok implementálhatók benne. Ennek a kérdésnek a megválaszolásához pedig az algoritmus elmélet alapfogalmait kell felhasználnunk. Éppen ezért a fejezet elején a szükséges alapfogalmakat ismertetem. Ezt követően megállapítom a nyelv jelenlegi fejlettségi szintjét. A fejezet végén pedig arra térek ki, hogy milyen megoldásokkal tervezem egy magasabb szintre emelni a nyelvet.

4.1. Fogalmak

Először a formális nyelvek leírására térek ki. [2] Ezt követően az automatákra, azokon belül is a turing-gépre. [3] Végezetül a polinomiális fogalmát fogom kifejteni.

- Egy tetszőleges nem üres, véges halmazt ábécének hívunk. Jelölése : Σ .
- A Σ Ábécé elemeit betűknek vagy karaktereknek hívjuk. Egy szó a Σ elemeiből képzett tetszőleges véges hosszú sorozat. Az y szóhosszát | y | jelöli. A Σ ábécéből képezhető összes szóhalmazának jelölése $\Sigma *$.
- A Σ ábécé feletti nyelvnek hívjuk a Σ elemeiből képezhető szavak egy tetszőleges (nem feltétlenül véges) részhalmazát, azaz L $\subseteq \Sigma *$.
- Két tetszőleges szó x=a1a2...an $\in \Sigma$ n$\subset \Sigma *$ és y=b1b2...bk $\in \Sigma$ k$\subset \Sigma *$ összefűzésén vagy konkatenációján az a1 a2...an b1 b2...bk $\in \Sigma$ n+k $\subset \Sigma *$ szót értjük.
- Az L1,L2 $\subseteq \Sigma *$ nyelvek konkatenációján (összefűzésén) azt a nyelvet értjük, melynek szavai egy L1-beli és egy L2-beli szó összefűzéséből állnak, jel ölése L1L2:

  L1L2= { w $\in \Sigma *$:w = xy ahol x $\in$ L1 és y $\in$ L2 }
- Az L $\subseteq \Sigma *$ nyelvtranzitív lezártja az L $* =$ { w $\in \Sigma *$: w = x1 x2 ... xk valamely k $\geq$ 0 számra, ahol x1,x2,...,xk $\in$ L }
- Nyelvtan(vagy formális nyelvtan) alatt egy olyan G= (V, Σ ,S,P)rend-szert értünk, ahol
  - V egy véges nem üres halmaz, a változók halmaza,
  - Σ egy ábécé,V $\cap \Sigma = \oslash$,
  - S $\in$ Va kezd ő változó,

  - P egy véges halmaz, az ún. levezetési vagy produkciós szabályok halmaza. P elemei $\alpha \to \beta$ alakúak, αés β tetszőleges,V és Σ elemeiből képzett sorozat, az egyetlen megkötés, hogy α tartalmazzon legalább egy változót is (β lehet akár az üres szó is).

- A G= (V, Σ ,S,P) nyelvtan által generált L(G) nyelv azokból a w $\in \Sigma *$ szavakból áll, melyekhez van egy valahány lépésből álló S $\to \Upsilon 1 \to \Upsilon 2 \to \cdot \cdot \cdot \to$ w levezetés.

- Chomsky-féle nyelvosztályok

  - A G nyelvtan a 3. osztályba tartozik, ha szabályai A $\to$ aB illetve A $\to$ a alakúak (A,B $\in$ V és a $\in \Sigma$ tetszőleges). A fentieken kívül megengedett még a kezdő változóra az S $\to \varepsilon$ szabály, amennyiben S nem fordul elő egyetlen szabály jobb oldalán sem.
  - AG nyelvtan a2. osztályba tartozik, ha szabályai A $\to \alpha$ alakúak, ahol A $\in$ V és $\alpha \in$ (V $\cup \Sigma$) $*$ tetszőleges, de legalább 1 hosszú. A fentieken kívül megengedett még a kezdő változóra az S $\to \varepsilon$ szabály, amennyiben S nem fordul elő egyetlen szabály jobb oldalán sem.
  - A G nyelvtan az 1. osztályba tartozik, ha szabályai β A $\Upsilon \to \beta\, \alpha\, \Upsilon$ alakúak, ahol A $\in$ V és α, β , $\Upsilon \in$ (V $\cup \Sigma$) $*$tetszőleges, de αlegalább 1 hosszú. A fentieken kív ül megengedett még a kezdő változóra az S $\to \varepsilon$szabály, amennyiben S nem fordul el ő egyetlen szabály bal oldalán sem.
  - A 0. osztálynyelvtanaira semmilyen megkötés nincs (a szokásos megkötésen kívül, miszerint a bal oldal legalább egy változót tartalmaz).

- Chomsky-normálforma Definíció Egy CF nyelvtan Chomsky-normálformájú (CNF), ha a szabályai A $\to$ aA $\to$ BC alakúak, ahol a $\in \Sigma$ és A,B,C $\in$ V

- Egy Turing-gépet a következő T= (Q, Σ, Γ, q0, -, F, δ)hetes ír le, ahol:

  - Q egy véges, nem üres halmaz, ez a gép állapotainak halmaza
  - Σ egy véges, nem üres halmaz, ez a bemeneti ábécé
  - Γ egy véges, nem üres halmaz, ez a szalagábécé
  - q0 $\in$ Q a kezdőállapot
  - - $\in \Gamma \ \ \Sigma$, a szalagon az üres jel
  - F $\subseteq$ Q az elfogadó állapotok halmaza
  - δ az átmeneti f üggvény, δ : (q,a) $\to$ (q $\prime$,b,D), ahol q, q $\prime \in$ Q,a,b $\in \Gamma$ ésD $\in$ { B,J,H } (azaz Balra, Jobbra vagy Helyben)

  A Turing-gép egyetlen szalaga van a bemeneti szalag. Ezen tud balra és jobbra mozogni az író fej, ami írni és olvasni tudja azt az egy betűnyi szeletet a szalagon, ami felett ál. A gép úgy működik, hogy az aktuális állapotra és karakterre, ami felett a fej áll, kikeresi a megfelelő szabályt. Ha ilyet nem talál, akkor megáll. Ha talál, akkor ellenőrzi, hogy a szabály jobboldalán definiált irányba tud e a fej mozogni. Ha nem, akkor leáll. Ha tud, akkor frissíti az aktuális állapotot a szabály jobb oldalán lévőre, beírja az aktuális pozícióba a szabály baloldalán definiált értéket és végrehajtja a mozgást. Ha a megállás pillanatában az aktuális állapot elfogadó, akkor a gép elfogadta a bemenetet.

- Az M Turing-gép által elfogadott nyelv:L(M) = { w $\in \Sigma$:M elfogadja w szót }

- k-szalagos Turing-gép egy olyan Turing-gép változat, amely k darabszalagot használ (k $\geq$ 1). A gépet ugyanúgy egy hetes adja meg, mint a korábbi definícióban:T= (Q, Σ , Γ ,q0„F, δ), ahol δ kivételével minden ugyanaz, mint korábban, az átmeneti függvény pedig: δ (q,a1a2...ak) $\rightarrow$ (q ′, b2b3...bk, d1d2...dk), ahol ai, bi $\in$ Γ , q, q′ $\in$ Q és di $\in$ B, J, H. Ez hatékonyabb, de bizonyított, hogy csak olyan nyelvet tud elfogadni, amilyet egy egyszalagos.

- A nemdeterminisztikus Turing-gépeket egy T= (Q, Σ , Γ , q0, -, F, δ)hetessel írhatjuk le, ahol Q, Σ , Γ , q0, -, F szerepe azonos a Turing-gépek esetén definiálttal, valamint: δ (q,a) $\subseteq$ Q $\times$ Γ $\times$ { J, B, H } Akármely nemdeterminisztikus Turing gép szimulálható egy determinisztikus turing géppel.

- A nemdeterminisztikus Turing-gép, akkor fogadja el a bemenetét, ha a számítási fában van elfogadó levél, azaz olyan konfiguráció, amiből nem lehetséges továbblépés és a hozzá tartozó belső állapot elfogadó

- L $\in$ P akkor és csak akkor teljesül, ha található egy olyan c >0 konstans´es egy L1 szópárokból álló nyelv, melyre L1 *in* P és

- Egy programozási nyelv akkor Turing-teljes, ha megoldhatók vele azok a problémák, amelyek egy determinisztikus Turing géppel is.

4.2. Lileto nyelv komplexitása

Egy nyelv akkor lehet csak Turing teljes, ha egy Turing gép működését képes szimulálni. Ehhez két elengedhetetlen nyelvi elemre van szükség: Feltételes elágazásokra és korlátlan mennyiségű adattárolásra. A `Lileto` nyelvben már mindkettő megtalálható, de nagyon korlátozott formában. Feltételes elágazások vannak beépítve minden műveletben, mert a műveletek kimenetele a változók típusától függenek. Ha egy sablonban több mezőnk van, de csak egybe szúrunk adatot, akkor a többi nem jelenik meg. Ezzel már egy feltételes elágazás történt a bemenet függvényében. Hiszen `Lileto`-ban van importálás, így a beimportált adatokat tekinthetjük bemenetnek.

```
data.lileto(1): {!=c data:= {[ a: | {[ x: | - ]} ]} !}
data.lileto(2): {!=c data:= {[ b: | {[ x: | - ]} ]} !}
comm.lileto:
{!
temp:={< {<=a Igaz {<=x >} >}{<=b Hamis {<=x >} >} >}
file :File={"C:\Users\Me\documents\data.lileto"}
input := file.c.data
outp=temp+input
!}
output(1)= {"Igaz"}
output(2)= {"Hamis"}
```

`Lileto`-ban tetszőleges mennyiségű adatot el tudunk tárolni, mivel a tárolók szabadon bővíthetők. Ugyanakkor van egy nagy hiányosság. Nem tudjuk a mutatót mozgatni a szalagon. Meg tudjuk ugyan indexelni a tárolókat, de elmenteni az indexet számként már nem, így újra indexelni sem tudunk vele, vagy a nála eggyel nagyobb vagy egyel kisebb számmal. Ciklusunk sincsenek, és ha egy sornyi adatot szúrunk be, akkor minden mezőbe csak az utolsó a megfelelő névvel rendelkező adat vesz részt a műveletben. Így bár egy lépést tudunk szimulálni, a gép egy lépés után megáll. A nyelv így nem sokkal, de

elmarad a Turing teljességtől. Komplexebb egy csak reguláris nyelvnél, de annál bővebb. Determinisztikus és véges lefutási idejű. Ezt könnyű belátni:

- a nyelv szekvenciális lefutású
- a hivatkozásokban nincsenek körök, hiszen a bal oldali értékről minden műveletnél másolat készül és tiltva van a körhivatkozás importálásban.
- Ha a szintaxisban több értelmezés is lehetséges, akkor mindig van azoknak egy sorrendje, amiből az első lép érvénybe.
- egy bemenetre a szükséges memória jól becsülhető, hiszen nincs a nyelvben rekurzió vagy végtelen ciklus.

A nyelvet az 1-es Chomsky-féle nyelvosztályba lehet sorolni, mert a zárójelek egymásba ágyazhatók, de az egy szinten lévők egymást követik és jól meghatározott, hogy egy zárójelből milyen zárójelek nyithatók. Ugyanakkor nem is 2-es, mert többszörös beágyazásokkal és változó hivatkozással már bővebb egy reguláris nyelvnél.

A nyelv szintaxisának nem is kell megütnie a 3-mas szintet, de a szemantikai megkötéseknek már illene. Jó lenne, ha a `Lileto`-ban lehetne szimulálni egy Turing gépet.

4.3. Lileto nyelv fejlesztése

A `Lileto` nyelvnek szüksége van feltételes elágazásokra, ciklusokra és Bool algebrára!

4.3.1. Feltételes elágazás

A tervek szerint a feltételes elágazások leginkább a Kotlin nyelv When struktúrájára fognak hajazni. Ez alatt azt kell érteni, hogy egyszerre fogja helyettesíteni a switch struktúrát és a soros if-else ágakat.

```
{. switch .}
{?
vari1
|
?=value1 -> {! ... !}
?=value2 -> {! ... !}
?=value3 -> {! ... !}
|
{! ... !}
?}
```

Itt a vari1 egy a when-ből elérhető változó, a "?=" a nyelv ==-je, value azághoz tartozó érték, a nyilak jobb oldalán lévő COMM zárójel az értékhez tartozó ág és a when végén lévő COMM zárójel pedig az else ág. Egy if-else ág sorozathoz csupán összetettebb feltételeket kell írnunk, mivel a zárójelben minden onnan látható változó el is érhető.

```
{. if-else .}
{?
condition1 -> {! ... !}
condition2 -> {! ... !}
condition3 -> {! ... !}
|
{! ... !}
?}
```

A Kotlin when struktúrájának megfelelően itt is lehetne a fejlécben új változót létrehozni

```
{. if-else .}
{?
a:= b<c
|
condition1(a) -> {! ... !}
condition2(a) -> {! ... !}
condition3(a) -> {! ... !}
|
{! ... !}
?}
```

Ekkor feltétel első operátorának nem kell baloldalt definiálni, ha a fejlécben deklarált változó az. A struktúrában az első olyan ág hajtódik végre, aminek a feltétele teljesül. A COMM zárójelek változót is téríthetnek vissza, ami egyben a when visszatérési értéke is lesz.

Ugyanakkor, a parancsokat és a visszatérítést végezhetjük közvetlen a when zárójelben is:

```
{. if-else .}
{?
a:= b<c
|
condition1(a) ->
command1_1
command1_2
command1_3
outp1=
condition2(a) ->
command2_1
command2_2
command2_3
outp2=
|
command3_1
command4_2
command5_3
outp6=
?}
```

4.3.2. Ciklusok

Mint tárolók esetén, itt is szeretnénk egy ciklussal leváltani minden standard ciklus típust. Itt gondolok a for-ra, while, do-while és foreach ciklusokra. alap szintaxis:

```
{@
conditionStart
|
Commands
|
conditionEnd
@}
```

While:

```
{@
conditionStart
|
Commands
@}
```

do-while:

```
{@
|
Commands
|
conditionEnd
@}
```

foreach:

```
{@
next in cont
|
Commands
@}
```

Ekkor az in operátor végzi a következő elem betöltését a next-be, és igazra értékelődik ki, ha ezt sikeresen el tudta végezni. for:

```
{@
i in start : condition -> step
|
Commands
@}
```

Ekkor az in végzi az iterálást. A ”start : condition -> step” pedig egy sorozatot ír le, ahol start a sorozat első eleme akármilyen érték definícióval, a condition egy bool értéket térít vissza és a step pedig a léptető függvény, ami visszaadja a sorozat következő elemét. Így néz ki például benne a 0->10 értékek végig léptetése:

```
{@
i in 0 : i ?< 10 -> i + 1
|
Commands
@}
```

Itt ?< a kisebb-e művelet.

4.3.3. Bool műveletek

A nyelvben az igaz és hamis értéket sokféleképpen meg lehet hivatkozni:

Igaz: true, True, T, 1b, Hamis: false, False, F, 0b

Azért nem adhatók meg kisbetűvel, mert minden a felhasználó által létrehozott változó is kis betűs, így túl vonzó lenne a t-t felüldefiniálni egy sablon létrehozásánál vagy az f-et egy fájl importálásánál.

Minden karakteres bool visszatérésű műveletet ?-lel kezdünk. Ha tagadni szeretnénk, akkor ezt a ’!’ követi.

- *kisebb-nagyobb:* A '<','>','=' tetszőleges kombinációja tetszőleges sorrendben a '?' és opcionális ' ' után.
- *tartalmazza-e:* ?in, ?!in
- *létezik-e:* ?is, ?!is
- *olyan típusú-e:* ?isa, ?!isa
- *osztható-e:* ?%, ?!%
- *és-vagy-nem:* !, &, |, !&, !|, ><, !><

A boolean operátor lusta, avagy, ha a baloldali értéke eldönti az értéket, akkor nem értékeli ki a baloldalát.

Típusuk a Bool lesz és inicializálni lehet majd őket az értékük szöveges megadásával is.

4.3.4. Számok

Az indexek hatékony kezeléséhez elengedhetetlenek a számok. `Lileto`-ban egyetlen számtípus fog kezelni minden létező számot.

A számokat négy byte-tömbben fogják tárolni, amik egész értékeket tárolnak majd: a, b, c, d. Ezekből a szám értéke így számolható: a^b/c^d Ezen kívül egy bool fogja tárolni a helyiértéket. A számokat $-a**b/c**d$, $-a1.a2/b1.b2$ formákban lehet megadni, ahol b és d elhagyható, ha 1 és a1 és b2 is, ha0 az értékük. Még b is elhagyható, ha 1 az ő és d értéke is és a helyi érték is, ha pozitív. Egyenlőség vizsgálat esetén mindkét számot átalakítja olyan formába, hogy csak a és b tartalmazzon értéket, binárisan 1-revégződjön a és relatív prímjei legyenek egymásnak. Minden tárolt számnak pontosan egy ilyen formája lesz! Ez ennek a tárolásnak a normálformája.

- *összeadás/kivonás:* +, -, +=, -=
- *szorzás, osztás:* *, /, *=, /=
- *egész és maradékos osztás:* /., /.=, %, %=
- *hatványozás/gyökvonás:* **,//,**=,//=
- *shift:* «, », «=, »=
- *kerekítés:* up, down, int, roud
- *abszolút érték:* $|value|$
- *és-vagy-nem:* !, &, |, !&, !|, ><, !><

A számok byte-onként indexelhetők és bit-enként két koordinátával.

```
a := 1001.0101b {. a.a=[00001001b,01010000b] a.b=-1 .}
b := a.0 {. b.a=[00001001b] b.b=0 .}
c := a.[1,3] {. c.a=1 c.b=0 .}
d := a.0.0 {. d.a=[00001001b] d.b=0 .}
```

Ezekkel az értékekkel pedig bool változók is inicializálhatók. Az bool műveleteket két szám között csak akkor végzi el, ha azok egészek (b=1,c=1,d=1).

Indexelni tudunk változóval is: a.[b] Indexelni tudunk név szerint: a.["name] Vagy speciális zárójellel is: a.! b:=2 b=!

Tudunk indexbe szúrni: a <[2] b

4.3.5. Determinisztikus Turing gép szimuláció

Az állapotaink legyenek egy tárolóban, ahol a kulcs az állapot neve és az érték egy bool, hogy elfogadó e az állapot. Például:

```
states := {[A:=True,B:=False,C:=True...]}
```

Egy másikban tároljuk a bemenet abc-t

```
inputAbc := {[{"a"}{"b"}{"c"}...]}
```

Egy másikban tároljuk a szalag abc-t

```
stripAbc := {[{"a"}{"b"}{"c"}...]}
```

Egy Text-ben tároljuk a kezdő állapotot és hozzunk létre egy számot az olvasó fejnek.

```
start := {"A"}
head := 0
```

Egy karakter legyen a szalag üres jel:

```
stripEmpty := {"e"}
```

Az átviteli függvényeket tároljuk egy táblázatban:

```
function :=
{[ state:Text, input:Text, nextState:Text, strip:Text, move:Numb
A, a, B, d, 1
...
]}
Itt az egy a jobbra, nulla a helyben maradás és mínuszegy a balra.
```

A bemenetet emeljük be egy külső fájlból:

```
input.lileto:
{!=c inputs:={[d, a, c ... ]} !}
turing.lileto:
f := {"input.lileto"}
inputs = f.c.inputs
```

Egy ciklusban járjunk a bemeneten:

```
b:= T
{@ b
|


...


@}
```

Keressük ki a megfelelő szabályt:

```
nextRule:={""}
{@ rule in function
|
{?
rule.state ?= state & rule.input ?= inputs[head] ->
```

```
nextRule = rule.nextRule
inputs[head] = rule.strip
head += rule.move
?}
@}
```

Ha nem találtunk ilyet, akkor a nextRule üres, álljunk le. Ha az olvasófej negatívba akar lépni, álljunk le. Más esetben frissítsük az állapotot

```
{?
nextRule?={""} -> b=false
head < 0 -> b=false
|
nextRule = rule.nextRule
?}
```

Ha a főciklus véget ért, akkor állapítsuk meg a végeredményt:

```
outp := T
{?
head < 0 -> b=false
|
b = states[state]
?}
b= !}
```

A szimuláció kész! Ha ez már sikeresen lefut, akkor a `Lileto` már egy Turing teljes nyelv! Ha a fordítás lefut polinomiális idő alatt, akkor még egy dologban biztosak lehetünk:

A `Lileto` képes egy olyan Turing gépet szimulálni, ami elfogadja a `Lileto` nyelvet! A `Lileto` akármilyen polinomiális futásidejű algoritmust letud futtatni időben és így az általa kiegészített nyelv is!

5. fejezet

Használata az IRTG-n

Az IRTG egy olyan nyelvtan, amit több gráf nyelv közötti komplex gráftranszformációk leírására fejlesztettek ki. Egy környezet független nyelvtant kell benne definiálni, ami csak az nemterminális szimbólumokból áll és az üres szimbólumból, így csak az üres szó vezethető le benne. Ennek a nyelvnek a szabályait szoktuk az szabály vagy szinkronizációs soroknak is hívni, mert feladatuk a komplex szabályok összefogása. Egy szabálysorhoz lehet társíthatunk formalizmusonként egy szabályt. Ennek a szabálynak a baloldala egyezik a szabálysor baloldalával, így ezeknek a szabályoknak csak jobboldalát kell definiálni. A jobb oldalak csak olyan nemterminális szimbólumokat tartalmazhatnak, ami a korábban említett szabály jobboldalán is szerepel. Ezekkel az egy szabálysorból és formalizmusonként egy további sorból álló szabályokkal kívánjuk összerendelni a formalizmusok egymással egyenértékű részeit. Így megadhatunk a különböző formalizmusok közötti kölcsönös megfeleltetéseket. Az általuk szinkronizált formalizmusokra pedig interpretációk ként is hivatkozhatunk. Minden szabály sor alatt minden interpretációhoz kell definiálni pontosan egy sort, ami csak az interpretáció azonosítóját és a szabály jobb oldalát fogja tartalmazni. Mivel pontosan egy sor mindig tartozik a szabályokhoz, a leírt nyelvtanok determinisztikusak és teljesek.

Az egyes formalizmusok több gráf leíró és szöveg manipulációs nyelven is definiálhatóak. A legegyszerűbb a szövegek konkatenációját leíró String algebra. A TagTree algebra pedig csak fagráfokra használható, de kezeli a csúcsok sorrendjét, így kiváló szintaxis fák leírására. A legfejlettebb viszont az s-graph algebra, ami ugyan nem kezeli a csúcsok sorrendjét, de egészen komplex irányított gráfokat is le tud írni. Ez a formalizmus emberi nyelvű szövegek szemantikai reprezentációs gráfjainak leírására kiváló. Ilyen szemantikát leíró formalizmus például a 4lang algoritmus is.

A nyelvtant az Alto futtató környezettel lehet futtatni. Ekkor, ha az interpretációkat jól szinkronizáltuk össze, akkor akármelyik formalizmus lehet a bemenet. A bemeneti formalizmusnak megkeresi minden lehetséges levezetését és a levezetések mentén legenerálja az összes többi interpretáció összes lehetséges kimenetét. A szabályokat súlyozhatjuk is a szerint, hogy mennyire gyakoriak, és akkor csak a legvalószínűbb levezetést fogja végrehajtani. Így az IRTG-vel képesek lehetünk egy emberi nyelvű szöveg, annak szintaxis fája, egyébb szintaktikai reprezentációja és akárhány szemantikai reprezentációja között kölcsönös konverzióra! Sőt, a szemantikai reprezentáción keresztül képes lehet a szöveg jelentését lefordítani más nyelvekre! Persze ez koránt sem könnyű feladat. Mai napig mennek ennek kapcsán a kutatások.

A problémát tovább nehezíti, hogy az IRTG szabályokat nagyon körülményes definiálni. Egy jelenleg is az AUT tanszéken futó kutatás során egy négy interpretációjú nyelvtant fejlesztettünk. Ez szabályonként öt sort jelent. Ha minimálisan tagoljuk is a kulcsot, akkor már hat sort. Ez az általunk írt 2700 szabály esetén már 16200 sort jelent. Ezen kívül

ez már olyan komplexitást jelentett, hogy az ALto 200000 mondaton 30 óráig futott! Mivel a 16200 sor mind egy fájlba került, mert az IRTG-ben nincs importálás. A szabályok is ránézésre nagyon hasonlóak voltak. White space karaktereken és kommentelésen kívül nincs más mód az IRTG kód tagolására. A szabályok nagy része között pár rövid szónyi a különbség, de nem volt mód a szabályok közötti öröklődésre. Végül a szabályokat kénytelenek voltunk generálni, de nem tudtunk képesek sokat optimalizálni a nyelvtanon, mert nem láttuk át eléggé. Új ember sem tudott a projekthez csatlakozni, mert félévbe telt az új tagoknak csak az IRTG megtanulása, nemhogy a nyelvtan átlátása.

Emiatt kezdtem el az IRTG és más alulfejlett nyelvek és nyelvtanokkal foglalkozni. Az IRTG esetében látványos a becsült fejlődés! Ennek szemléltetéséhez viszont szükség van az IRTG mélyebb ismertetésére.

5.1. Az Alto

A kutatás során a kódot az Alto-val[1] (Algebraic Language Toolkit) fordítottuk és futtattuk. Az Alto egy nyílt forrású parszer, többféle algebrát is megvalósít, amelyek IRTG-be ágyazva használhatók, mint például az `s-graph` és a `tag tree` algebrák. Ezen kívül is szabadon bővíthető új algebrákkal. Már korábban is használták gráftranszformációra és szemantikai elemzésre is, lásd [4, 1]. Nagy előnyt jelent, hogy Java-ban lett implementálva, így szinte bármely platformon futtatható. Rendelkezik grafikus és konzolos felhasználói felülettel is.

5.2. Az IRTG

Az IRTG (Interpreted Regular Tree Grammar, interpretált reguláris fa-nyelvtan) [5] 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)
```

5.1. ábra. Egy, az `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 csúcssorrendet tartó, fa kimenetű `tag tree algebra`, vagy az irányított gráf kimenetű `s-graph algebra`. 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

[1] `https://github.com/coli-saar/Alto`

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 átírása során szúródik be az alkalmazott átírási szabálynak az interpretációhoz tartozó lépése. 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 környezetfüggetlen 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 interpretációk 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, stringé 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 az 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 a `wide string algebra`, a `tree with arities algebra` vagy a `set algebra`[2]. Ezek nem részei a dolgozat fókuszának.

5.3. A string algebra

Az SA (`string algebra` 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.* szöveget adja vissza. Ez a konkatenáció kommutatív és asszociatív.

5.4. A tag tree algebra

A TTA (`tag tree algebra`)-nak, mint az SA-nak, csak egy művelete van, a *merge* (egyesítés). A TTA esetében viszont már a bemenetek nem nyers szövegek, hanem csúcs sorrend

[2] `https://bitbucket.org/tclup/alto/wiki/Algebras`

tartó fa gráfok. Tartalmazhatnak slot-okat, amiket TTA esetében *hole*-nak(lyuk) nevezünk. A hole-okat '*'-gal 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 a *John loves Mary* mondat szintaktikai fája:

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

, aminek ez a fa felel meg:

Ugyanez az igei kifejezés, avagy a VP (Verb 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 bal oldali 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))

fát kapjuk. Éppen ezért a interpretációs lépések segítségével kell összerakni szabályról szabályra ezt a fát. lásd F.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 [6] szintaktikai fáiban. Ilyen a *the black cat* főnévi kifejezés is, aminek a szintaktikai fája a következő:

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

A fát merge nélkül praktikusan csak három bemenetű szabállyal lehetséges implementálni, lásd F.2.

Merge segítségével sokkal optimálisabban is megoldható, lásd F.3.

Négy gyermekű főneves kifejezések esetében ez már nem lehetséges. Ilyen például a *this British industrial conglomerate*, amihez a:

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

fa tartozik. Ezt a fenti logikával nem tudjuk helyesen megoldani. lásd F.4.

Itt a második N_BAR elkészítésekor, a merge művelet során a JJ mindkét lyukba beszúródik, így a NP4(DT(this), JJ(British), JJ(British), NN(conglomerate)) fa jönne létre.

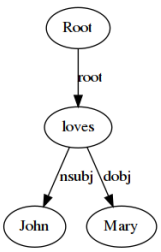
Ezt a TTA korlátai miatt nem tudjuk megkerülni.

5.5. Az s-graph algebra

Az SGA (`s-graph algebra`) a legbonyolultabb algebra, amit használunk. Több műveletet és összetett gráfnyelvet használ. Az egyetlen hiányossága, hogy nem képes a csúcsok sorrendjét kezelni. Ezért szorulunk a TTA használatára csúcssorrendet 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 stringként 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:



5.2. ábra. A *John loves Mary.* mondat UD gráfja

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 F.5.

A példa sok szabálya megfeleltethető a tree interpretáció esetében 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 gráfba 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 F.6.

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

5.6. Konklúzió

Az IRTG egy önmagában is nehezen kezelhető formalizmus, amihez számos formalizmus készült. Ezek a formalizmusok mind sajátos szintaxissal rendelkeznek és egyikükben sem lehet minden lehetséges problámát kellő hatékonysággal kezelni. Ezen kívül a megoldáshoz legközelebb álló formalizmus egy kifejezetten nehezen olvasható szintaxissal rendelkezik.

Ha mindegyik formalizmusnak elkészítjük a sablonjait, akkor mindnek egy egységes szintaxisa lesz. A `Lileto`-ban van importálás és a zárójelek egymásba ágyazhatóak, így az IRTG nyelvtan kódjai jól strukturálhatóak benne. Az egyes szabály típusokhoz elég egy-egy sablont definiálni és onnantól a változó adatot egy átlátható táblázatos formában definiálhatjuk. Az egyes szabályokat is több részből rakjuk össze így, ha jól strukturálunk, könnyebben megfigyelhetjük az egyes szabálysorok közötti eltéréseket és felfedhetjük az optimalizálási lehetőségeket. Ha egy sabályt módosítunk, akkor a lényegi adatokon sokszor nem kell módosítani. Ez eddig súlyosabb esetben több száz szabály módosítását jelentette. `Lileto`-val már csak a sablonon kell változtatni.

A `Lileto`-val képesek leszünk az egyes sablonok között mélyebb logikát is definiálni. Például megtehetjük, hogy a bemenetek egyes szabályokban nehezen megfogalmazható tulajdonságai alapján mindig más szabályokat léptetünk érvénybe.

6. fejezet

Összefoglalás és jövőbeli tervek

Tdk dolgozatom keretében kifejlesztettem a `Lileto` nyelvet, ami más nyelvek kiegészítésére való. Célja, hogy az alárendelt nyelv kódja tömörebb, átláthatóbb és strukturáltabb legyen. Ezt elsősorban sablonműveletekkel valósítja meg, de más alapvető funkciókat is megvalósít. Behozza a nyelvekbe az importálást, típusos változó deklarációt, típusdeklarációt, tároló típusokat, iterációt stb. Ezen kívül számos egyedi megoldást is bevezet, mint például a "referencia" vagy a több nevű változók. Előbbi egy hivatkozás, ami mindig kilistázza a rá illeszkedő változókat, ez valósítja meg a témakiírás egyik fő tételét is.

A kiírás szerint az elsődleges cél az IRTG nyelvtan kiegészítése volt. A szakdolgozatban részletesen írok ennek a nyelvtannak a szakmai hátteréről és működéséről is. Ez alatt ki is elemeztem a nyelvtan és futtatókörnyezete kapcsán felmerülő problémákat. Úgy vélem, hogy ezek többségét maguk a fejlesztők is könnyen orvosolhatják hosszú távon, ezért bővítettem a feladat fókuszát és álltam elő egy univerzális megoldással.

A `Lileto` ugyan akár 80%-os sor-redukciót is képes elérni az IRTG-k általunk kezelt típusa esetén, még így is sok benne a redundancia. A működés sincs kellő hatékonysággal tesztelve, éppen ezért a következő iterációkban három célt fogok szem előtt tartani. Az első a Turing teljesség. Implementálni fogom a számkezelést, szövegkezelést, logikai műveleteket, ciklusokat és feltételes elágazásokat. Ezeket követően kerül majd sor a teljes körű validációra. A cél a 100%-os tesztlefedettség mind kód, mind funkcionalitás tekintetében. Ekkor várhatóan már 85%-os sorredukcióra is képes lesz. Végleges formájában a `Lileto` egy olyan nyelv lesz, ami mindennemű felesleges redundanciát hatékonyan semlegesít. A legfontosabb cél az, hogy mindezt átláthatóan, egyszerűen és hatékonyan tegye.

Köszönetnyilvánítás

Köszönöm mindenkinek, aki munkásságával, jelenlétével a szakdolgozat elkészüléséhez és minőségéhez hozzájárult. Köszönet elsősorban konzulensemnek, Mezei Gergelynek, aki mindig rendelkezésemre állt, ha forrásokra, szakmai segítségre vagy jó tanácsokra volt szükségem. Köszönet Dobai Botondnak és Osman Omarnak, akik segítettek a nyelv véglegesítésében, és felhasználói szempontú kidolgozásában. Végezetül, de nem utolsósorban pedig köszönet szüleimnek és Vágó Nórának, amiért biztosították a lelki és anyagi hátteret a munka folyamán is.

Irodalomjegyzék

[1] Jonas Groschwitz – Alexander Koller – Christoph Teichmann: Graph parsing with s-graph grammars. In *Proceedings of the 53rd ACL and 7th IJCNLP* (konferenciaanyag). Beijing, 2015.

[2] Bach István: *Formális Nyelvek*. 2001, Typotexe. 1. Formális nyelvek.

[3] Bach István: *Formális Nyelvek*. 2001, Typotexe. 6. Az automataelmélet alapjai.

[4] Alexander Koller: Semantic construction with graph grammars. In *Proceedings of the 14th International Conference on Computational Semantics (IWCS)* (konferenciaanyag). London, 2015.

[5] Alexander Koller – Marco Kuhlmann: A generalized view on parsing and translation. In *Proceedings of the 12th International Conference on Parsing Technologies (IWPT)* (konferenciaanyag). Dublin, 2011.

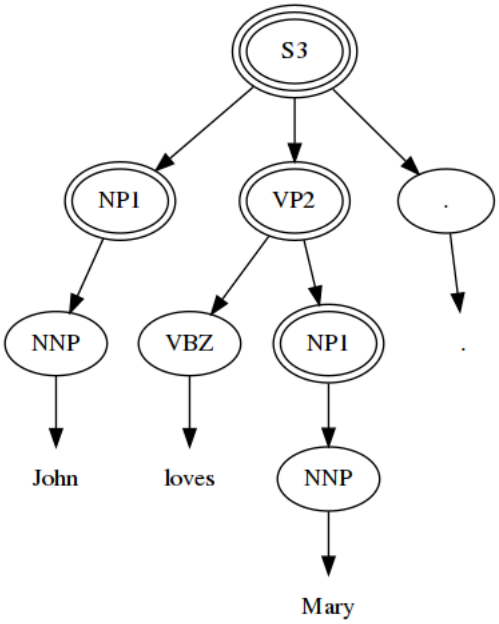
[6] Mitchell Marcus – Beatrice Santorini – Mary Ann Marcinkiewicz: Building a large annotated corpus of english: The penn treebank. 1993.

Függelék

F.1. Példa 1

```
S! -> S_NP_S_BAR(NP, S_BAR)
[tree] @(?2,?1)

S_BAR -> S_NP_VP(VP, PUNCT)
[tree] S3( *, ?1, ?2)

VP -> VP_VB_NP(VB,NP)
[tree] VP2(?1,?2)

NP -> NP_NN(NN)
[tree] NP1(?1)


NN -> John_NNP
[tree] NNP(John)

NN -> Mary_NNP
[tree] NNP(Mary)

VB -> loves_VBZ
[tree] VBZ(loves)

PUNCT -> punct_PUNCT
[tree] .(.)
```

F.1.1. ábra. Egyszerű IRTG a *John Loves Mary.* mondat TT-jének generálására



F.1.2. ábra. F.1-ben leírt IRTG által generált levezetési fa a *John Loves Mary.* mondat TT-jére.

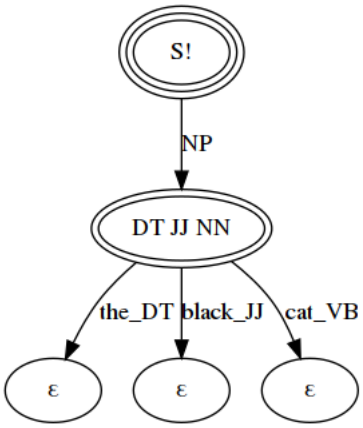
```
@(
S3( *, VP2( VBZ( loves ),  NP1( NNP( Mary ) ) ), .( . ) ),
NP1( NNP( John ) )
)
```

F.1.3. ábra. F.1-ben leírt IRTG tree interpretációjának kimenete *John Loves Mary.* mondat TT-jére.

F.2. Példa 2

```
S! -> NP( DT, JJ, NN )
[tree] NP3(?1,?2,?3)

DT -> the_DT
[tree] DT(the)

JJ -> black_JJ
[tree] NN(black)

VB -> cat_VB
[tree] VB(cat)
```

F.2.1. ábra. Egyszerű három bemenetű IRTG a *the black cat* szószerkezet TT-jének generálására



F.2.2. ábra. F.2-ben leírt IRTG által generált levezetési fa a *the black cat* szószerkezet TT-jére.

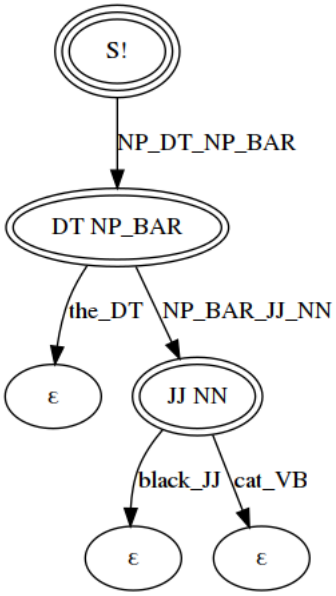
```
NP3( DT( the ), JJ( black ), NN( cat ) )
```

F.2.3. ábra. F.2-ben leírt IRTG tree interpretációjának kimenete *the black cat* szószerkezet TT-jére. A kimenet maga a *the black cat* szószerkezet TT-je.

F.3. Példa 3

```
S! -> NP_DT_NP_BAR( DT, NP_BAR)
[tree] @(?2,?1)

NP_BAR -> NP_BAR_JJ_NN(JJ,NN)
[tree] NP3(*,?1,?2)

DT -> the_DT
[tree] DT(the)

JJ -> black_JJ
[tree] NN(black)

VB -> cat_VB
[tree] VB(cat)
```

F.3.1. ábra. Egyszerű IRTG a *the black cat* szószerkezet TT-jének generálására



F.3.2. ábra. F.3-ban leírt IRTG által generált levezetési fa a *the black cat* szószerkezet TT-jére.

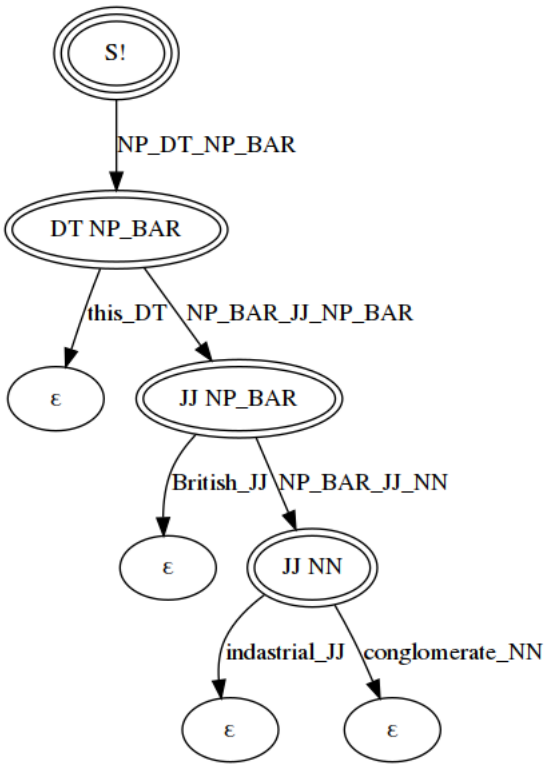
```
@(  NP3( *, JJ( black ), NN( cat ) ), DT( the ) )
```

F.3.3. ábra. F.3-ben leírt IRTG tree interpretációjának kimenete a *the black cat* szószerkezet TT-jére.

F.4. Példa 4

```
S! -> NP_DT_NP_BAR( DT, NP_BAR)
[tree] @(?2,?1)

NP_BAR -> NP_BAR_JJ_NP_BAR(JJ,NP_BAR)
[tree] @(?2,?1)

NP_BAR -> NP_BAR_JJ_NN(JJ,NN)
[tree] NP4(* ,* ,?1 ,?2)

DT -> this_DT
[tree] DT(this)

JJ -> British_JJ
[tree] JJ(British)

JJ -> industrial_JJ
[tree] JJ(industrial)

NN -> conglomerate_NN
[tree] NN(conglomerate)
```

F.4.1. ábra. Egyszerű IRTG a *this British industrial conglomerate* szószerkezet TT-jének generálására

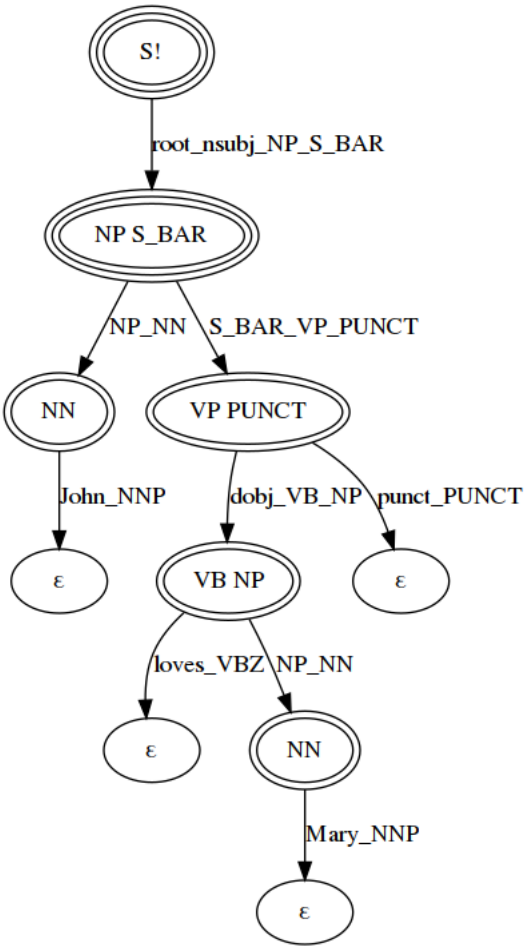


F.4.2. ábra. F.4-ben leírt IRTG által generált levezetési fa a *this British industrial conglomerate* szószerkezet TT-jére.

F.5. Példa 5

```
S! -> root_nsubj_NP_S_BAR(NP, S_BAR)
[ud] merge(
f_dep(merge("(Root/Root :root r<root> :nsubj (d<dep>))", r_dep(?1))),
?2
)

S_BAR -> S_BAR_VP_PUNCT(VP, PUNCT)
[ud] ?1

VP -> dobj_VB_NP(VB,NP)
[ud] merge(f_dep(merge("(r<root> :dobj (d<dep>))", r_dep(?2))),?1)

NP -> NP_NN(NN)
[ud] ?1


NN -> John_NNP
[ud] "(John<root> / John)"

NN -> Mary_NNP
[ud] "(Mary<root> / Mary)"

VB -> loves_VBZ
[ud] "(loves<root> / loves)"

PUNCT -> punct_PUNCT
[ud] "(punct<root> / punct)"
```

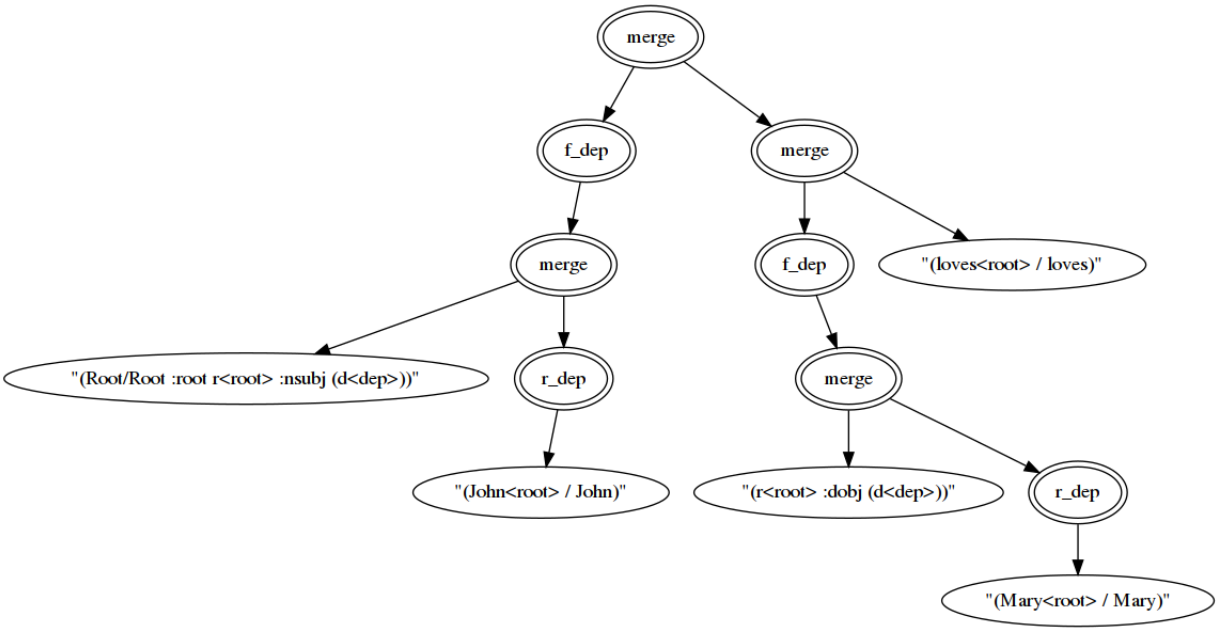
F.5.1. ábra. Egyszerű IRTG a *John Loves Mary.* mondat UD gráfjának generálására



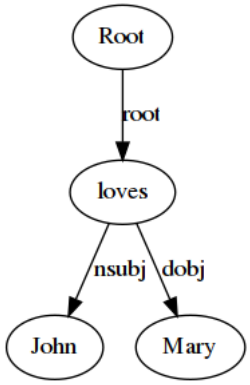
F.5.2. ábra. F.5-ben leírt IRTG által generált levezetési fa a *John loves Mary.* szószerkezet UD gráfját leíró s-graph-ra

```
merge(
f_dep( merge(
"(Root/Root :root r<root> :nsubj (d<dep>))",
r_dep( "(John<root> / John)" )
) ),
merge(
f_dep( merge(
"(r<root> :dobj (d<dep>))",
r_dep( "(Mary<root> / Mary)" )
) ),
"(loves<root> / loves)"
)
)
```

F.5.3. ábra. F.5 UD interpretációjának kimenete



F.5.4. ábra. F.5-ben leírt IRTG UD interpretációja által generált s-graph kifejezés műveleti fája

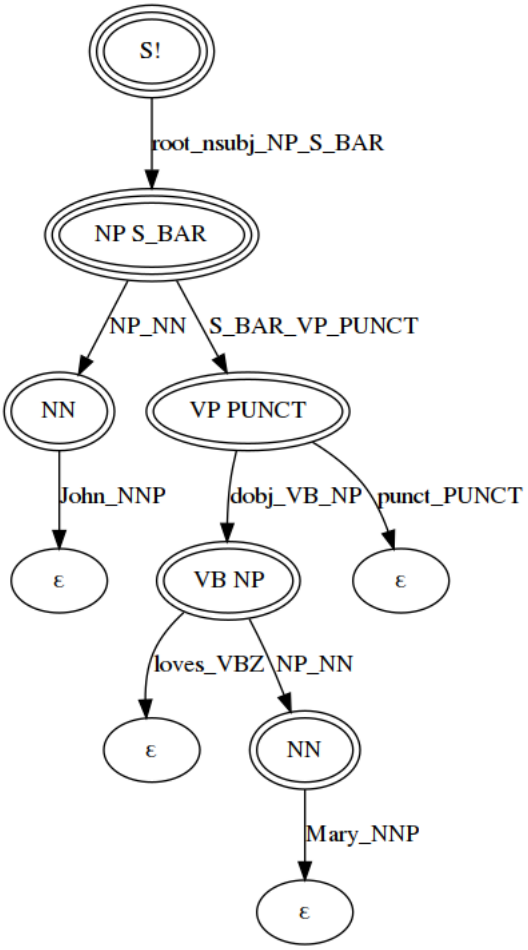


F.5.5. ábra. F.5-ben leírt IRTG UD interpretációja által generált s-graph kifejezés végeredménye. Ez SGA formátumában így néz ki: ”(Root/Root :root loves<root>/loves :nsubj (John/John) :dobj (Mary/Mary))”

F.6. Példa 6

```
S! -> root_nsubj_NP_S_BAR(NP, S_BAR)
[tree] @(?2,?1)
[ud] merge(
f_dep(merge("(Root/Root :root r<root> :nsubj (d<dep>))", r_dep(?1))),
?2
)
[fourlang] merge(
f_dep(merge("(Root/Root :root r<root> :1,0 (d<dep>))", r_dep(?1))),
?2
)

S_BAR -> S_NP_VP(VP, PUNCT)
[tree] S3( *, ?1, ?2)
[ud] ?1
[fourlang] ?1

VP -> dobj_VB_NP(VB,NP)
[tree] VP2(?1,?2)
[ud] merge(
f_dep(merge("(r<root> :dobj (d<dep>))", r_dep(?2))),
?1
)
[fourlang] merge(
f_dep(merge("(r<root> :2 (d<dep>))", r_dep(?2))),
?1
)

NP -> NP_NN(NN)
[tree] NP1(?1)
[ud] ?1
[fourlang] ?1

NN -> John_NNP
[tree] NNP(John)
[ud] "(John<root> / John)"
[fourlang] "(John<root> / John)"

NN -> Mary_NNP
[tree] NNP(Mary)
[ud] "(Mary<root> / Mary)"
[fourlang] "(Mary<root> / Mary)"

VB -> loves_VBZ
[tree] VBZ(loves)
[ud] "(loves<root> / loves)"
[fourlang] "(loves<root> / loves)"

PUNCT -> punct_PUNCT
[tree] .(.)
[ud] "(punct<root> / punct)"
[fourlang] "(punct<root> / punct)"
```

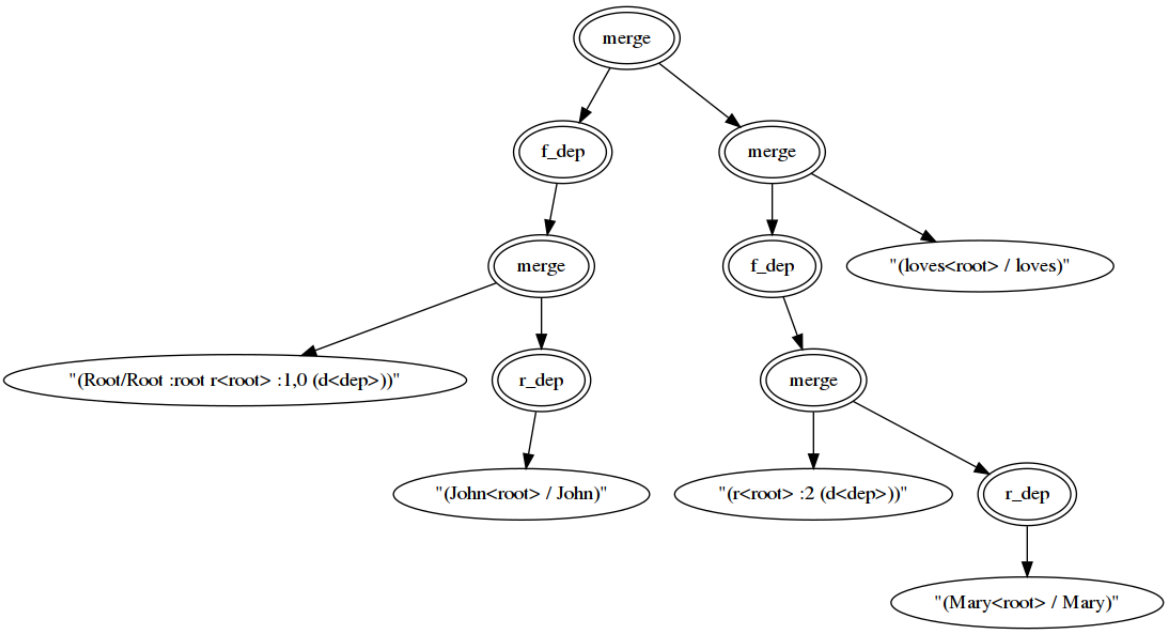
F.6.1. ábra. IRTG három algebrával a *John Loves Mary.* mondat szintaktikai fájának, UD gráfjának és 4lang gráfjának generálására



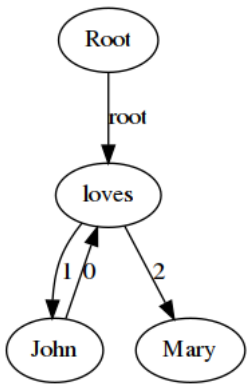
F.6.2. ábra. F.6-ban leírt IRTG által generált levezetési fa a *John loves Mary.* szószerkezet 4lang-ját leíró s-graph-ra

```
merge(
f_dep(
merge(
"(Root/Root :root r<root> :1,0 (d<dep>))",
r_dep(
"(John<root> / John)"
)
)
),
merge(
f_dep(
merge(
"(r<root> :2 (d<dep>))",
r_dep(
"(Mary<root> / Mary)"
)
)
),
"(loves<root> / loves)"
)
)
```

F.6.3. ábra. F.6 fourlang interpretációjának kimenete



F.6.4. ábra. F.6-ben leírt IRTG fourlang interpretációja által generált s-graph kifejezés műveleti fája



F.6.5. ábra. F.6-ben leírt IRTG fourlang interpretációja által generált s-graph kifejezés végeredménye