Multilingual Grammatical Translator

Lavanya Aprameya

August 2017

This write-up describes a system for translation using multilingual grammars (MGT). The write-up will first outline the motivation for such a system, followed by a description of theories used to produce the system as well and of the system framework, culminating in the results of the system, a reflection on the process and the system itself, and a list of future extensions.

Motivation

In general, publicly available computational systems made for translation lack the robustness to represent the morphosyntactic properties of the phrases involved in translation, both the phrases from which and to which translation occurs. This disregard for underlying linguistic representations is made apparent by cases such as demonstrated by (1), (2), and (3). (1) represents the input into a commonly used translation system, Google Translate, in English. (2) represents the result of translating (1) into Spanish, and (3) represents the result of translating (2) into English once again.

(1) I feel like Google Translate can't understand more sophisticated linguistics.

(2) Siento que Google Translate no puede entender una lingüística más sofisticada.

(3) I feel that Google Translate can not understand more sophisticated linguistics.

Two important features of (3) in comparison to (1) to note are that "like" is translated into "that" and that "can't" is translated into "can not". The first change is not necessarily significant, since the words "like" and "that" can be used interchangeably in this construction, but the second change is more significant since it introduces ambiguity that is not present in (1). "can't" in (1) is interpreted to mean that Google Translate does not have the ability to understand, while "can not" (3) can have an additional interpretation: that Google Translate has the ability not to understand. This distinction, which an underlying syntactic representation can clarify, is not an issue with which translational systems typically concern themselves. However, several flaws in such systems can be resolved by taking morphosyntax into account. The MGT framework hopes to avoid such mistakes by using an underlying representation rooted in morphosyntactic theory in order to translate between languages.

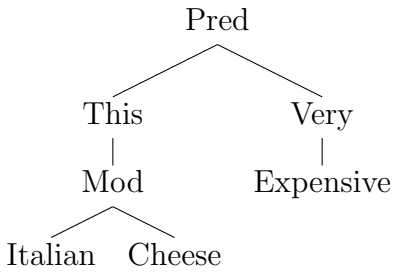
Theories

The main theory in linguistics which drives the MGT framework is the idea that multiple languages can be represented by a similar layout, which is explored in Aarne Ranta's *Grammatical Framework* [1]. The groundwork for the parsing model of the MGT is provided by chart-parsing [2] and by the computational linguistic theories of unification and feature structures [3] [4].

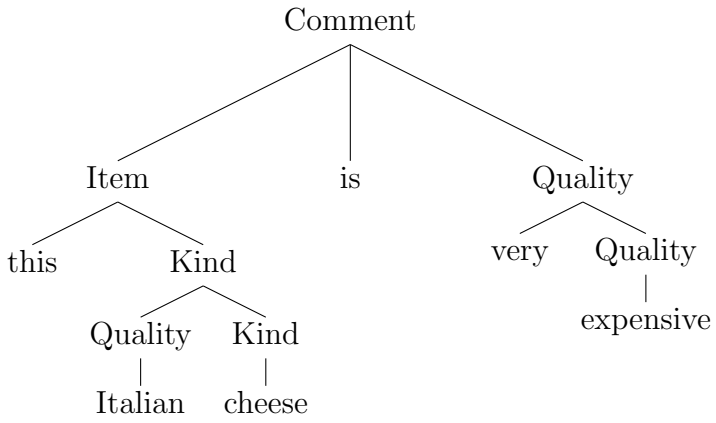
Grammatical Framework

In *Grammatical Framework*, Ranta proposes the idea that languages contain some representation which is inherent in all languages, but the manner in which this representation is realized can differ between languages. For example, the English sentence "This Italian cheese is very expensive" might be represented by a structure such as the following:

(4) (a)



(b)



(4a) represents what Ranta refers to as the *abstract tree* of the sentence, while (4b) represents the *parse tree*. An abstract tree can be turned into a parse tree through a series of linearization rules, and in the same way that abstract trees can be manipulated into parse trees, parse trees can also be manipulated into abstract trees. Since all languages translate the same abstract tree through variations of linearization rules, the parse tree for the target language of a translation can be produced by manipulating the parse tree of the source language in order to match the target language's linearization rules. Using this concept of an underlying abstract representation which allows for such manipulation, the MGT concerns itself with the process of manipulating from parse tree to parse tree.

Chart-Parsing Algorithm

The chart-parsing algorithm, as described by Mellish, uses the approach of memoization with matrices in order to store all possible translations at each point in the process in order to combat left-recursion and to prevent unnecessary computation. For example, if a grammar contains the rules VP $\rightarrow$ TV NP and VP $\rightarrow$ TV NP PP, chart-parsing prevents the need to search for NP once again in order to check the second rule. The algorithm for chart-parsing is as follows:

for j from 1 going up to n do
    set $chart(j\text{-}1, j)$ to $\{A|A \rightarrow word_j\}$
        for i from j going down to 0 do

for k from i+1 going up to j-1 do
    set *chart* (i, j) to
        chart (i, j) $\cup$ (*chart* (i, k) * *chart* (k, j))

If a final category head appears in *chart*(0,n), then the algorithm has succeeded in parsing.

Unification Grammars & Feature Structures

In order to translate between languages, certain features of sentences which are either independent of or not guaranteed by syntactic categories must be taken into account. This core idea which drives unification grammars and feature structures proves to be essential to the MGT framework.

Features are aspects of either a lexical entry or syntactic category which mark something particular. For example, a noun in English might have the feature NUMBER, which is realized as either SINGULAR or PLURAL and represents content of the noun which would not have been apparent through simply its syntactic category. A *feature structure* such as (5) can be used to represent any noun that is masculine and third person singular. In unification grammars, a syntactic tree contains nodes of such feature structures rather than strings.

(5) $\begin{bmatrix} \text{Gender} & \text{M} \\ \text{Number} & \text{Sing} \\ \text{Person} & 3 \end{bmatrix}$

As the name indicates, unification grammars are grammars in which feature structures *unify* in order to produce the correct final feature structure. For example, the subject NP of the sentence and the verb must agree in person and number. In unification grammars, the features of both of these nodes are compared, and if they do not clash, the sentence takes on the unified feature structure. If unification results in a clash, the sentence is ungrammatical. Feature structures might be underspecified for one or more features in some nodes, in which case unification with another node which specifies for those features results in a feature structure with the most specific

feature. The grammar itself specifies for which features unify under what circumstances. (6) outlines a simple example of unification with the syntactic rule NP → Det N.

$$(6)\quad \begin{bmatrix} \text{Entry} & \text{the} \\ \text{Category} & \text{Det} \\ \text{Person} & 3 \end{bmatrix} + \begin{bmatrix} \text{Entry} & \text{dog} \\ \text{Category} & \text{N} \\ \text{Number} & \text{Sing} \\ \text{Person} & 3 \end{bmatrix} = \begin{bmatrix} \text{Entry} & \text{the dog} \\ \text{Category} & \text{NP} \\ \text{Number} & \text{Sing} \\ \text{Person} & 3 \end{bmatrix}$$

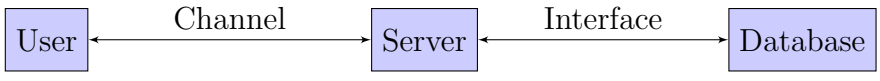
The vocabulary in unification grammars consists of words which map to feature structures, with the most correct and most specific feature structure being the one that takes on the mapping in parsing and translation. Unification grammars are necessary for multilingual grammars since not all languages specify for the same features. For example, English does not specify for case, but German does, and in order to translate between the languages, the feature (case) is overspecified in some cases and sufficiently specified in others, but since vocabulary selection chooses the most correct and most specific, overspecification is not an issue for translation.

MGT Framework

This section describes the framework of the MGT, beginning with an overview of the system architecture, followed by a walkthrough of the system design and a description of data structures created in order to support the framework.

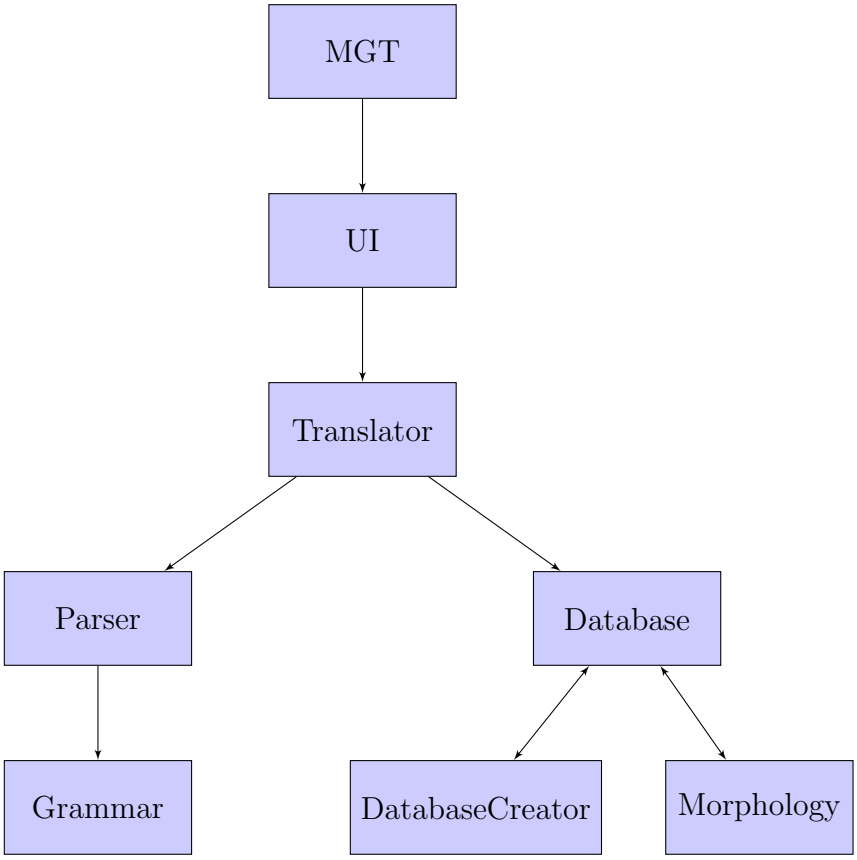
System Architecture

The basic system architecture of the MGT can be described in the following diagram. The MGT is primarily a client-server architecture, in which the client can request information from the server but cannot add information. The MGT also contains a shared data architecture since the server retrieves information from the database and updates it during execution.



System Design

The design of the MGT system centers around the main file `mgt.ml` as follows:



The main file, `mgt.ml`, is the entry point to the MGT via the commands `make` or `make run`.

`mgt.ml` utilizes the module `UI` in order to handle the user-interface portion of the MGT. `UI` is responsible for processing user input and responding appropriately as well as formatting output. `UI` accomplishes this through a series of prompts for the user (e.g., the choice of a language to translate from, the choice of a language to translate to, a request for a string to translate) and extensive exception-handling, both of user-defined exceptions and any other exceptions that might occur throughout the course of the MGT.

`UI` utilizes the module `Translator` in order to perform the main computation portion of the MGT. `Translator` is responsible for translation from an input language to an output language of an input string. `Translator` accomplishes this through the idea of parse tree manipulation, invoking the modules `Parser` and `Database`, to handle parse tree creation.

`Translator` utilizes the module `Parser` in order to parse through an input string and create a list of parse trees for the input string. `Parser` is responsible for chart-parsing and for unification. `Parser` accomplishes this through functions which chart-parse a string, which turn a chart into a tree and impose features on the tree according to the string's grammar, and which reject trees that fail unification.

`Parser` utilizes the module `Grammar` in order to handle features and a rule-based grammar. `Grammar` is responsible for defining the modules `FeatVal` to deal with feature values, `Feature` to deal with features, and `Features` to deal with feature bundles. `Grammar` realizes `Feature` as a module with a type dependent on `FeatVal` and `Features` as a module with a type dependent on `Feature`.

`Translator` utilizes the module `Database` in order to provide grammars for translation. `Database` is responsible for storing the database of grammars and for handling any additional manipulation of the grammars. `Database` accomplishes this by storing the database in a file `database.json`, which is created using `DatabaseCreator` and read from by `Database`. Morphology is added into the database via the module `Morphology`, which applies simple verb and noun morphology to each grammar.

Data Structures

In order to maintain a functional and modular framework, the MGT requires an abundance of data structures, some of which are realized as types and some of which are realized as modules.

Even if not a regular data type, the control flow in the module `UI` is directed by the exceptions `QuittingEarly`, `Restart`, `QuittingEarly`, `WordList`, `Help`, `NoInput`, `History`, `InvalidLang`, and `InvalidYN`. Each exception signifies

that the user has acted out of the ordinary flow of the program, either redirecting the flow (by ending or restarting), by pausing the flow (by requesting a word list, help, or translation history), or by providing input that cannot be used in program flow (by typing nothing or an invalid option).

The module `UI` also contains the module `HistoryEntry`, which is used in a `ref list` in order to allow the user to save translations and to view their history in the same session. A history entry consists of the language translated from, the language translated to, the string translated from, and all the strings translated to. The module contains code for creating a history entry as well as for displaying it.

The module `Grammar` contains the module `FeatVal`, which represents all the root values a feature can take on as described by the concept of feature structures. Currently, a `FeatVal` can be any of 1 or 2 or 3 for person, SING or PL for number, MASC or FEM for gender, NOM or ACC for case, and YES or NO for subordinate.

The module `Grammar` also contains the module `Feature`, which represents a feature in the context of unification. A `Feature` can be any of `Any` (any value added during unification is acceptable), `Only(s)` (only the value `s` is acceptable), `Unifier(s)` (only the value `s` is acceptable, and it is used in further unification), and `UnifyBlock(f)` (the feature is the feature represented by `f`, and it is not used in further unification).

The module `Grammar` also contains the module `Features`, which represents a feature structure. A `Features` is a record containing fields `person`, `number`, `gender`, `case`, and `subordinate`, all of which are of type `Feature`, as well as the fields `entry` and `lex`, which are of type `string`. The `entry` of a feature structure is the string that it represents (i.e., either a category name or a word). The `lex` of a feature structure is the lexical item that it represents, allowing for translation across languages.

The module `Grammar` also contains the type `rule`, which represents a rule in a grammar. A `rule` is a tuple with the first element being of type `Features` and the second being of type `Features list`, where the first element represents the category head of the rule and the second element represents the

categories which the head maps to in order. A `grammar` is defined as a list of such `rule`s.

The module `Parser` contains the types necessary to implement chart-parsing, such as the type `splitpoint`, which represents the splitpoint k in the chart-parsing algorithm, the module `CellEntry`, which represents the information stored in each cell in a chart, and the type `chart`, which represents the chart (i.e., matrix) used for memoization.

The module `Parser` contains the type `tree`, which is realized either as a `Branch` with a tuple argument consisting of a category head (i.e., a feature structure) and the list of trees which stem from it, or as a `Leaf` with a single feature structure as an argument. A `tree` represents a parse tree.

The module `Translator` contains the type `translate_tree`, which functions like a normal tree, except in that a leaf contains a list of possible feature structures to be translated to. A `translate_tree` represents an intermediate tree in the translation algorithm.

Results of the MGT

The MGT successfully parses for sentences which include intransitive, transitive, and subordinate clause-specifying verbs, as well as phrases which requires unification, across the not necessarily binary grammars English, Italian, and German.

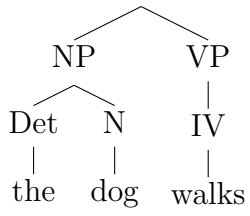
The sentences in (7)-(9) represent one set of intransitive sentences across English, Italian, and German, with trees (10)-(12) as the output of the MGT for these sentences from either of the other two languages. The feature structures have been left out of the output of the trees for readability.

(7) the dog walks

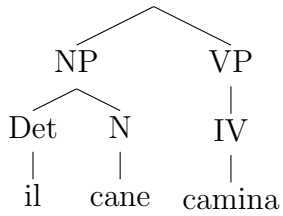
(8) il cane cammina

(9) der hund geht

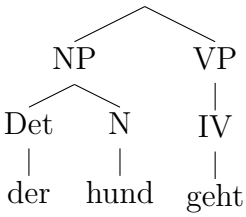
(10)



(11)



(12)



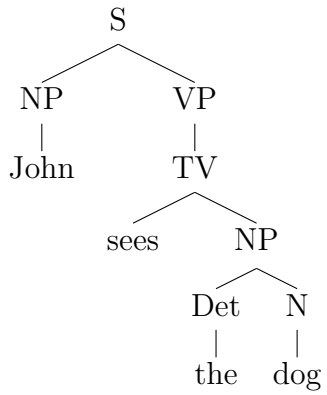
The sentences in (13)-(15) represent one set of transitive sentences across English, Italian, and German, with trees (16)-(18) as the output of the MGT for these sentences from either of the other two languages. The feature structures have been left out of the output of the trees for readability.

(13) John sees the dog

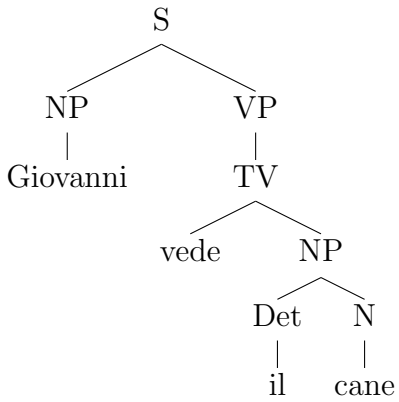
(14) Giovanni vede il cane

(15) Johannes sieht den hund

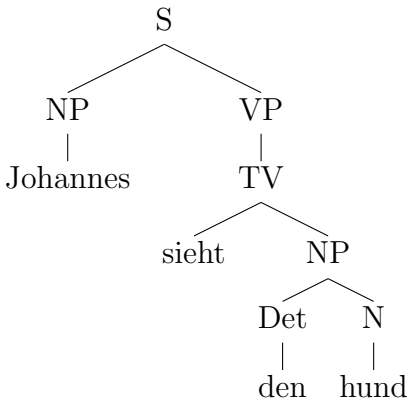
(16)



(17)



(18)



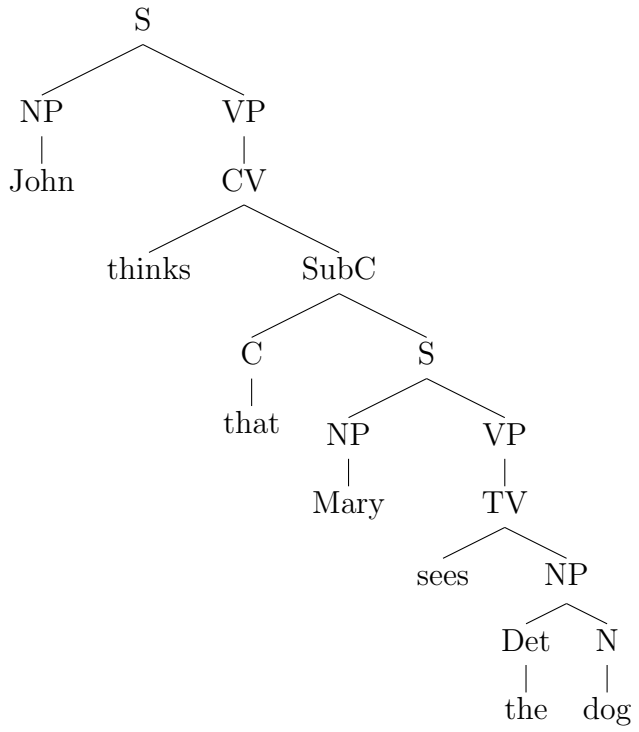
The sentences in (19)-(21) represent one set of transitive sentences across English, Italian, and German, with trees (22)-(24) as the output of the MGT for these sentences from either of the other two languages. The feature structures have been left out of the output of the trees for readability.

(19) John thinks that Mary sees the dog

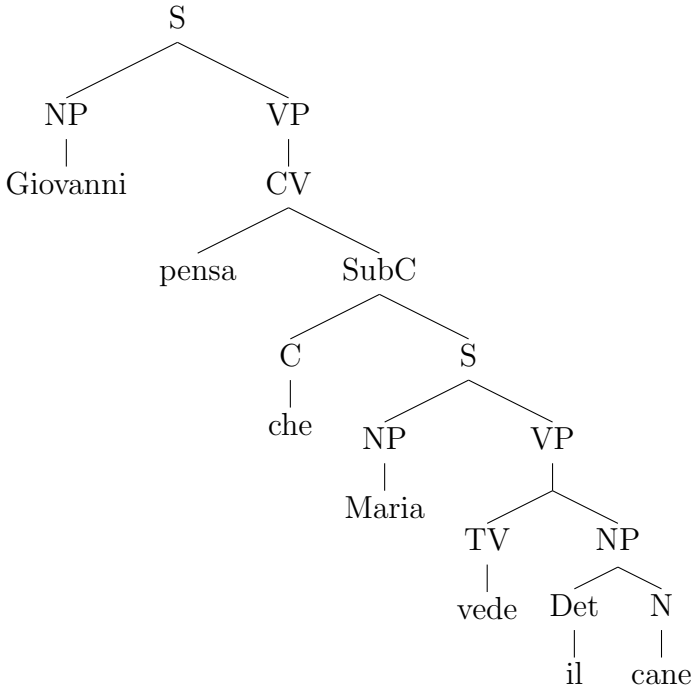
(20) Giovanni pensa che Maria vede il cane

(21) Johannes denkt dass Maria den hund sieht

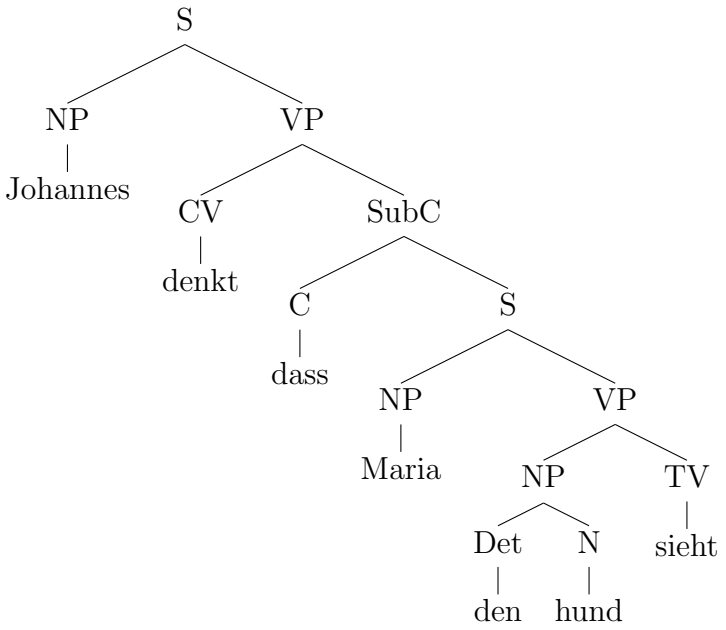
(22)


S
NP
VP
John
CV
thinks
SubC
C
S
that
NP
VP
Mary
TV
sees
NP
Det
N
the
dog

(23)


S
NP
VP
Giovanni
CV
pensa
SubC
C
S
che
NP
VP
Maria
TV
NP
vede
Det
N
il
cane

(24)



Reflections & Future Extensions

The MGT's success at representing an underlying feature structure across languages and utilizing it for translation show that Ranta's theories, as well as those of the proponents of lexical-functional grammars, can be extended to a translation model. Some problems, however, include the tedious rule-crafting required to avoid mishaps in unification, as well as the implementation of unification being parallel to the theory rather than exactly in line (i.e., in other unification grammars, there is no distinction between `Unifier(s)` and `Only(s)`, and there is no need for `UnifyBlock(f)` since unification applies on a rule-by-rule basis). While the MGT's system is efficient, it perhaps does not represent the linguistic theories behind unification in the way that it should.

Some future extensions include incorporating rules which currently violate the unification process of the MGT, as well as extending grammars and adding new ones. The most constructive extension would be to extend the MGT to a language which is rather dissimilar to the current languages which the MGT recognizes in order to test Ranta's hypothesis of multilingual grammars.

Acknowledgments

The author thanks Professor John Hale for the work in his Computational Linguistics course, which provided much of the theory for this project, as well as for his support and advice while this project was in its initial stages.

References

[1] Ranta, Aarne. *Grammatical framework: Programming with multilingual grammars*. CSLI Publications, Center for the Study of Language and Information, 2011.

[2] Mellish, CS and Whitelock, P and Ritchie, GD. *Techniques in Natural Language Processing 1*. 1994.

[3] Farghaly, Ali Ahmed Sabry. *Handbook for language engineers*. CSLI Publications Stanford University, 2003.

[4] Bresnan, Joan and Asudeh, Ash and Toivonen, Ida and Wechsler, Stephen. *Lexical-functional syntax*. John Wiley & Sons, Vol. 16, 2015.