A Linguist-Friendly Machine Translation System for Low-Resource Languages

Ronald M. Lockwood

A thesis
submitted in partial fulfillment of the
requirements for the degree of

Master of Science

University of Washington

2015

Committee:

Fei Xia

William D. Lewis

Program Authorized to Offer Degree:

Computational Linguistics

©Copyright 2015

Ronald M. Lockwood

University of Washington

Abstract

A Linguist-Friendly Machine Translation System for Low-Resource Languages

Ronald M. Lockwood

Chair of the Supervisory Committee:
Professor Fei Xia
Computational Linguistics

Low-resource languages have largely been left out of the machine translation revolution. Speakers would benefit from machine translation for many different tasks if it were available. Because of insufficient text data, the results of using statistical machine translation are subpar. The best choice for these languages is probably a transfer-based approach where rules define how to translate from one language to another. Unfortunately, the transfer-based systems available today are not easy to use for anyone outside the computational linguistics field. This thesis presents a transfer-based system that is easy to use for ordinary linguists. It is linguist-friendly because a central component is the intuitive application Fieldworks Language Explorer. This application serves as the repository for lexicons, the place where entries are linked and the tool for the analysis piece of the analysis-transfer-synthesis-style system. Apertium, the well-established open-source machine translation platform, is used for the transfer piece of the system and STAMP for the synthesis piece. All of these programs are well-documented. The linguist's role is to link lexicon entries and write transfer rules to do either word or syntactic-level translation. Although this machine translation system is a proof-of-concept system, I show that it translates texts successfully in a test case using Persian and Gilaki. Such a system can be used by ordinary linguists all over the world for almost any language pair where machine translation is needed.

Contents

1 Introduction 1
2 Literature Survey 3
2.1 Existing Systems 3
2.2 Limitations of these Systems 4
3 Methodology 6
3.1 Resources & Requirements 6
3.1.1 Lexicons 6
3.1.2 Rule Management 7
3.1.3 Ambiguity 7
3.2 Solution Approach 8
3.2.1 Lexicons 8
3.2.2 Rule Management 9
3.2.3 Ambiguity 10
3.2.4 Transfer Process 10
3.2.5 Synthesis Process 11
3.2.6 Tool Interaction 11
3.2.7 Incremental Power and Complexity 11
4 Tools 13
4.1 **FLEx** 13
4.2 **Apertium** 15
4.3 **STAMP** 16
4.4 Initial Setup 16
4.4.1 **FLEx** Lexicons 16
4.4.2 **FLEx** Grammar Settings 18
4.4.3 Linking Senses 18

4.4.4 Transfer Rules . . . 18
5 Implementation . . . 20
5.1 System Overview . . . 20
5.2 Analysis . . . 21
5.2.1 Manual Parsing . . . 21
5.2.2 Automated Parsing . . . 23
5.3 Transfer . . . 24
5.4 Synthesis . . . 26
5.5 Full Example . . . 27
5.6 Design Features . . . 28
5.6.1 Bilingual Lexicon . . . 28
5.6.2 Complex Entries . . . 28
5.6.3 Grammatical Category Mapping . . . 29
5.6.4 Bilingual Lexicon Customization . . . 29
5.6.5 Unknown Words . . . 30
5.6.6 Inflection Information . . . 30
5.6.7 Variant Entries . . . 30
5.7 Limitations . . . 31
5.8 Transfer Rules File Overview . . . 32
5.8.1 Declaration . . . 32
5.8.2 Rules . . . 33
5.9 Modules . . . 35
5.9.1 Extraction Modules . . . 35
5.9.1.1 Extract Source Text Module . . . 36
5.9.1.2 Extract Bilingual Lexicon Module . . . 39
5.9.2 Transfer Module . . . 42
5.9.3 Synthesis Modules . . . 42
5.9.3.1 Catalog Target Prefixes Module . . . 42
5.9.3.2 Convert Text to **STAMP** Format Module . . . 42
5.9.3.3 Extract Target Lexicon Module . . . 44
5.9.3.4 Synthesize Text Module . . . 46

5.9.3.5 Insert Target Text Module . . . 46
6 Experiments . . . 47
6.1 A Test Case with Persian and Gilaki . . . 47
6.1.1 Overview of the Language Differences . . . 47
6.1.2 Test Case Initial Setup . . . 48
6.1.3 Scope of the Test Case . . . 48
6.2 Initial Implementation . . . 48
6.3 Experiment Setup . . . 50
6.4 Results . . . 50
6.4.1 Previous Training . . . 51
6.4.2 Stage 1 - Lexical Coverage Improvements . . . 51
6.4.3 Stage 2 - Fixes to Synthesis Problems . . . 51
6.4.4 Stage 3 - Persian Possessive Pronominal Enclitics . . . 52
6.4.5 Stage 4 - Persian Adverb ham . . . 54
6.4.6 Stage 5 - Prepositions to Postpositions . . . 55
6.4.7 Stage 6 - Indefinite Nouns . . . 56
6.4.8 Stage 7 - Custom Bilingual Lexicon Entries . . . 57
6.4.9 Stage 8 - Accusative Pronouns . . . 57
6.4.10 Stage 9 - Possessive Enclitics and Prepositions . . . 59
6.4.11 Unhandled Phenomena . . . 59
6.5 Final Rules . . . 59
6.6 Discussion . . . 60
7 Conclusion and Future Work . . . 63
A Installing and Running the System . . . 67
A.1 Installation . . . 67
A.1.1 Install **FLExTools** . . . 67
A.1.2 Download MT Modules . . . 67
A.2 Running . . . 68
A.2.1 Edit the Configuration File . . . 68
A.2.2 Start **FLExTools** . . . 68
A.2.3 Run Step 1 Collection . . . 69

A.2.4 Run the Transfer Module (Step 2) 69

A.2.4.1 Apertium Commands Reference 70

A.2.5 Run Step 3 Collection 70

B Configuration File 71

C Troubleshooting 73

References 75

1

Introduction

Even though the machine translation (MT) field is advancing more and more, enabling the translation of texts between languages with growing quality, for the low-resource languages of the world, MT is still largely out of reach. The main reason for this is that the tremendous advances in MT are due mainly to the use of statistical methods. Statistical methods rely heavily on parallel texts to train an MT system. In low-resource languages there are often very few written texts and of those that exist, they do not have a parallel text in another language. Without the parallel texts, statistical MT will give subpar results. If we do not take a statistical approach, the other main options for MT are either an interlingua approach or a transfer-based approach. Interlingua approaches have not been successful except in specific domains which leaves us with a transfer-based approach. Of the transfer-based systems that exist, I believe they are too complex or too specialized for an ordinary linguist to use. This thesis shows that it is possible to create a transfer-based MT system that is linguist-friendly. The system I propose is linguist-friendly because it leverages a common application that linguists use as the core of the system, namely **Fieldworks Language Explorer (FLEx)** (FLEx, 2014) as well as other tools that are free and well-documented. I believe that an ordinary linguist can, with some training, effectively employ MT for low-resource languages using this system. I demonstrate a successful test case of the system using two Western Iranian Languages. The system is able to translate texts from Persian into Gilaki. In the below documented experiment using a 55-sentence narrative text, I attained a BLEU score of nearly 35 with 17 rules. It can in principle be used to translate between almost any two languages.

The system I propose is in some ways not a new MT system. It uses parts of three existing systems: **FLEx**, **Apertium** (Forcada et al. 2010), and **STAMP** (Weber et al. 1990). What is new is how these applications are put together and how easy it is to use. **FLEx** serves as the repository for lexicons, the place where entries are linked and the application for the analysis piece of the analysis-transfer-synthesis-style[1] system. **Apertium**, the well-established open-source machine translation platform, is used for the transfer piece of the system and **STAMP** for the synthesis piece. All of these programs are well-documented. I use **Python** scripts to enable the applications to interface with each other and

[1] Some use the term "generation" instead of "synthesis".

to add certain features. The linguist's role is linking lexicon entries and writing transfer rules to do either word or syntactic-level translation. The result is a synergic MT system that is simple to use.

One initial question that comes up is whether there is a need for MT in low-resource languages? I believe the answer is a definite yes. Community development is a big need in many parts of the world and many of these communities speak low-resource languages. Take primary education, for instance. Studies have shown that children learn best if they begin the educational process in their own mother tongue and then transition to a national or regional language after the first couple years (King, 2003). Illiteracy is a big issue in many communities and again it is best to start literacy training in the mother tongue. For these two domains translating texts into the low-resource vernacular languages is a big need. Many groups desire to communicate with people via their vernacular language. This includes those who want to communicate about health issues, such as hygiene or viruses. Others want to translate authoritative texts like the Bible. Many wish to communicate with these communities about human or political rights.

One might argue that most of the people in such communities are already bilingual in another language. Therefore there is no need for texts in the vernacular. Yes, there is always certain percentage of people that are bilingual with a high degree of fluency in both languages, but there is always a certain percentage that do not have a high fluency level in a second language, be it the national or a regional language. There always seems to be some socio-economic sub-group that is only adequately served in their mother tongue.

While an MT system like this one is very promising in helping translate texts for these communities, we also have to realize that even the best system is not going to give perfect translations. I assume that the output of this MT system will need to be refined by hand before being acceptable.

In chapter 2, I review some literature on existing transfer-based MT systems. In chapter 3, I look at the requirements of a linguist-friendly transfer-based MT system and show how this system meets these requirements. Chapter 4 gives a closer look at the system's three core tools and an explanation of how you set up the system. Chapter 5 presents an overview of the system's design and features, a start to finish example and detailed descriptions of the working modules with examples. In chapter 6, I explain the test case used for this system, report the results of an experiment of putting a new text through the system and discuss the results and wider implications. Lastly, in chapter 7 I conclude with closing remarks and ideas for future development.

2

Literature Survey

In this chapter I highlight some of the existing transfer-based MT systems in use today. All of them use at their core the analysis-transfer-synthesis paradigm. I show why each of these is unsuitable for our goals.

2.1 Existing Systems

Apertium is an open-source software MT platform (Apertium, 2015). It originated from a Spanish government-funded program called, Open-Source Machine Translation for the Languages of Spain (OpenTrad) (Forcada et al. 2010). It began as a system to translate between closely related languages and was later extended to work with unrelated languages. Because of its Spanish origins, languages of Spain are well-developed in the **Apertium** platform. Spanish and Catalan, for example, can be paired with many other languages for translation. A total of 40 language pairs are deemed stable for translation at the time of writing. **Apertium** is actively being used today and the global community is developing more and more language pairs. Anyone can contribute to the rules and lexicons that make up a language pair. All the files that make up the lexicons and rules are in a well-documented XML format.

An **Apertium** MT system for a language pair can include many different modules in a pipeline process. The basic inputs are the following: monolingual lexicons (source and target languages), probability models and/or rules (for both the disambiguation and lexical selection), a bilingual lexicon (for lexical transfer), transfer rules and of course the source text to be translated. The output is the text in the target language.

Another transfer-based MT system is CARLA which stands for computer assisted related language adaptation. CARLA is free software and originated in Latin America in the 1980s as a system to translate between language dialects.[1] The seminal article related to the research and development of CARLA is Weber & Mann, 1981.

Text files in CARLA are formatted with "standard format markers" where a marker precedes each piece of data. Inputs to the system are the following: source and target lexicons, settings files (for various definitions), transfer rules,

[1] Originally the system was called CADA — computer assisted dialect adaptation. (Black & Weber, 1987).

a lexical substitution table and the source language text. Disambiguation is a separate process with CARLA which can be done after analysis and/or after synthesis. Analysis is done with a module called AMPLE (Weber et al. 1988). Lexical transfer and morpheme manipulation is done with a module called STAMP (Weber et al. 1990). Originally these two modules were the core of the system. Later SENTRANS was added to allow the user to manipulate words at the sentence level (Buseman, 1991).

Another transfer-based MT system is OpenLogos (OpenLogos, 2005). OpenLogos descended from a commercial system called LOGOS (Scott 2003). The original system dates back more than 40 years ago. OpenLogos supports German and English as source languages and French, Italian, Portuguese and Spanish as target languages. It is open-source software under the Gnu Public License and is extensible. The system in its original design is able to work with any language. The basic inputs to the system are lexicons in source and target languages, tables for morphology, sets of rules for transfer, semantic tables, a translation memory database and the input text. The output is the text in the target language. The natural language input is converted to symbolic strings which are "semantico-syntactic". Symbolic string patterns are matched by rules at multiple points in a pipeline and these rules fire and convert the strings. "Notations pertinent to target equivalences are recorded as analysis of each source constituent is completed, in contrastive linguistic (tree-to-tree) fashion." (OpenLogos, 2005) At the end synthesis to the target output is performed.

SYSTRAN is a well-known MT system that is also a transfer-based system (SYSTRAN, 2015). SYSTRAN is also the name of the company that sells products and services around their translation technology. The company has provided products for language translation for more than 40 years. SYSTRAN has capability in over 130 language pairs. The basic inputs to SYSTRAN according to their 2001 architecture are the following: source and target syntactic lexicons, a transfer lexicon, transfer rules, a translation memory database and the source text (Senellart et al. 2001). The output is the target text. Today SYSTRAN includes statistical MT components where probability models enhance the system (Koehn & Senellart, 2010). The system has been and continues to be proprietary.

2.2 Limitations of these Systems

Each of these systems have issues that make them unsuitable for the goal of a linguist-friendly MT system for low-resource languages. **Apertium** is probably the best system of the ones mentioned above for the goal. It is open-source, well-documented and easy to extend to any new language pair. It is because of this flexibility that I have chosen the transfer component of **Apertium** as part of this MT system. The main issue with **Apertium** is the "unfriendliness" of the lexicons. The two monolingual lexicons and the bilingual lexicon are text files. These are tedious to maintain, whereas using **FLEx** gives the user a graphical user interface for working with lexicons. Also, when using the traditional **Apertium** setup, the entries in the monolingual **Apertium** lexicons have to be manually kept in sync with the

entries in the bilingual lexicon. Any manual maintenance like this is error-prone. This MT system, on the other hand, is less error-prone since you only have to maintain a link between source and target lexicon entries. Lastly, **Apertium** does not have the built-in capability to work with senses of words. **FLEx**, on the other hand, has this built-in capability.

CARLA is a system that is also free and you are able to configure it for any language pair. It has a windowed interface that hides the underlying file maintenance (CARLAStudio, 2008). This makes it more user-friendly. Its main weakness is the number of different settings that you have to maintain. It is not easy for an ordinary linguist to keep track of all the settings for the two languages. Each module in the pipeline has its own settings. CARLA requires someone that understands computers very well to get everything right and to figure out what to change when results are incorrect. Also, CARLA's standard transfer component, SENTRANS, is less powerful than the **Apertium** transfer engine that I am using. With **Apertium** you can use logical structures and operators; SENTRANS cannot. **Apertium** can reference target entries; SENTRANS cannot.

OpenLogos seems promising on the surface, especially with its long history of MT. It is open-source and you can in theory extend it to any language pair. Going deeper, one finds that "Development of new source languages is *hard*. Target languages are merely *challenging*" (OpenLogos, 2005). It does not seem likely that the ordinary linguist will be able to configure OpenLogos for any language pair he or she chooses.

SYSTRAN is powerful and has a proven track-record. The problem is that it is not free and open. For a fee one can use it for language pairs that they support, but for a low-resource language not found in the list, there are no options available.

3

Methodology

In this chapter I look at the resources needed and the requirements for a linguist-friendly transfer-based MT system. Then I describe my solution to these issues and discuss other considerations. The basic methodology used in this MT system is an analysis-transfer-synthesis approach. The approach goes as follows: The source text is broken down into morphemes, the morphemes are modified and rearranged as necessary and at the end the morphemes are put back together to form the target text.

3.1 Resources & Requirements

The basic needs of the system start with a tool or tools to do each step of the process — the analysis process, the transfer process and the synthesis process. Inherent in the transfer process is the need for a way to define and apply rules to transform a text from the source language to the target language. In addition to that, an electronic lexicon for the source language and the target language are needed. A final basic need is a bilingual lexicon where source words are linked to target words. Let us look at some of the resources needed for a transfer-based MT system and also examine the necessary requirements.

3.1.1 Lexicons

First of all, there is a need for lexicons. For the source language a lexicon is needed which can be referenced for every word in the text that is to be translated. A lexicon gives us a lot of useful information about a word such as its grammatical category or part of speech. For the target language a lexicon is also needed since words from the source text need to end up as valid target words. In other words, the target language lexicon will help determine if words are well-formed.

A resource like a lexicon may not be easy to obtain in a low-resource language. The linguist may need to create it if there has never been an electronic lexicon before. How can a person create a lexicon in a language that has never

had one? It is a big job and requires extensive help from the language community. What tool can be used to hold the lexical data? There are many tools available, but it would be ideal if it is a tool that is commonly used by linguists for lexicography. When thinking about it from an MT perspective it should be a kind of database. A database that is highly normalized — where consistency of the data is maintained.

For the MT task, only a minimal amount of information is really necessary in a lexicon. Just having the form of the word, a lemma and a grammatical category might be sufficient, but a linguist and the language community might be interested in much more than this. They might want to have a gloss or definition of the word in another language at a minimum and there are many more useful things such as example sentences, synonyms, antonyms and notes. It would be a huge advantage to have a lexicon tool that can handle all this information and at the same time serve as a repository for words in an MT system.

A bilingual lexicon where source words are linked to target words is also a needed resource. A bilingual lexicon is very important because it tells the transfer process how to convert a source word to a target word. A bilingual lexicon needs to be able to deal with homographs (words that have the same written form but different meanings) and be able to distinguish between them. When we consider linking words together we must consider at what level we are linking them. A bilingual lexicon can link just the forms, it can link form plus grammatical category or it can link at the level of senses of words. Having more granularity in how linking is done between source and target languages is preferable. The higher the degree of granularity there is, the higher chance there is to resolve ambiguity.

3.1.2 Rule Management

Central to a transfer-based MT system is the need for a rules repository. A good tool to manage the rules is needed so that rules can be rearranged easily. The rules should be able to be turned off or on. Having a degree of modularity in the rule system would also be a big advantage, so that if rules share common components, those can be abstracted into separate modules.

There is also a need for some kind of rule-language. We need a way to refer to morphemes in words, a way to refer to whole words and a way to refer to grammatical categories, to name a few. The rule-language should also be able to allow rules to be conditional. For example, if a certain condition exists, do a certain thing.

3.1.3 Ambiguity

A requirement of any good MT system is the ability to deal with ambiguity. Ambiguity can occur at multiple points in the analysis-transfer-synthesis process. Looking at just the analysis phase, there can be significant ambiguity after the analysis process has been done. This could be ambiguity in form, meaning, grammatical category, affixation or in

other ways. The analysis process will typically examine a word and output all the possible parses for the word. How do we determine which is the right parse for a given sentence? A statistical solution is to use a tagger to disambiguate using the grammatical categories of the possible parses. This is achieved by training the tagger on a corpus of tagged texts. For a low-resource language this is not possible since text corpora are limited. Other solutions have to be found. One approach would be to disambiguate the text after the analysis process is done and then a clean list of words could be sent to the transfer process. Another approach would be to transfer all the ambiguous words and disambiguate at a later stage. If the source and target languages share a lot of the same ambiguity, disambiguating might be reduced due to ambiguous parses collapsing into the same form. The first approach is preferable in my opinion since having ambiguous words in the transfer process would complicate rule application.

Ambiguity is present at other levels too. For one source word, there may be more than one target word to which it translates. There may also be syntactical ambiguity. A word could ambiguously pattern with the preceding phrase or the following phrase. If the parsing into phrases is done incorrectly, the translation will be wrong.

The last requirement is that the MT system should be able to handle any language in any script.

3.2 Solution Approach

My approach to creating a linguist-friendly MT system for low-resource languages is to take tools that already exist and put them together to make a usable beginning-to-end solution. For the lexicon and the analyzer tool I use **FLEx**. For the transfer process I use the transfer engine that is part of the **Apertium** open-source MT platform. For the synthesis process I use a tool called **STAMP**. **Python** scripts act as the glue between processes. The text in the source language originates in the source language **FLEx** project and the resulting translated text is inserted into the target language **FLEx** project.

How does this solution address the resources and requirements discussed above? Let us look at the issues one by one.

3.2.1 Lexicons

FLEx is an excellent choice for a lexicon tool. It is a common lexicography tool used by linguists around the world. It makes sense to leverage a tool linguists are already using as the place to draw lexical data from for an MT system. **FLEx** is a good repository for a lexicon because it uses a highly normalized database where consistency is maintained. Most data in **FLEx** is stored just one time in its database and all other objects link to the data. For example the grammatical category “noun” is stored once in the category list. Many lexical entries have “noun” as their

grammatical category, but "noun" is never copied into these entries; instead they hold a link to "noun" in the category list. If "noun" were to be changed to some other label, say "substantive" in the category list, that change would be reflected in all the lexical entries that were linked to "noun".

FLEx is also a good application for developing a lexicon from scratch. It has tools such as the ***Collect Words*** view to support rapid word collection. **FLEx** has over 80 built-in fields for storing lexical information and supports user-defined custom fields. It is capable of being a repository for all the information a language community would want to store about their language. Lastly **FLEx** is fully Unicode compatible and supports all languages and scripts. It also includes support for smart fonts for complex lesser-known languages where the native script is not supported.[1]

The source language lexical data can be stored in one **FLEx** project and the target language data in another. In this scenario, it is possible to create links between source lexical entries and target lexical entries. See Section 4.4.3. This then becomes the bilingual lexicon that we need for the MT system.

FLEx has the concept of ***headword*** which helps to distinguish homographs from each other. This is useful for the MT system since it can help us uniquely refer to each lexical entry in the lexicon. For example, for two different words that have the same spelling *bank*, one ***headword*** would be named $bank_1$ and the other $bank_2$. I make use of the unique ***headword*** when passing the names of lexical entries back and forth between **FLEx**, **Apertium**, and **STAMP**.

In regards to the granularity of the lexical data that can be worked with in **FLEx**, we can go down to the sense level when establishing links between source and target languages. **FLEx**'s database is hierarchical. At the top level are entries, within an entry are senses and each sense can have a set of subsenses. See example (1). It is at the sense-level that grammatical category is assigned. Using links at the sense-level helps us avoid ambiguities that normally arise because of sense differences. As an example of how this plays out, if a word in the source language has two possible translations in the target language, two senses can be created for the source word and each sense can be linked to the appropriate word (and sense) in the target language.

3.2.2 Rule Management

For managing transfer rules, the **Apertium** system provides a good solution. **Apertium** uses an XML format to store the rules in plain text. As such, it is human-readable and not dependent on an application to read it. This is critical when considering long-term archiving. Also an XML format clearly defines what data is being stored. A document type definition (DTD) is available for the rule file which can be used to test the rule file's integrity. The **Apertium** transfer engine uses a pattern matching system which is a good way to organize and execute the rules. For a given pattern a rule is specified. Rules can be turned off by commenting out the top-level rule element. The **Apertium**

[1] The Graphite system (Graphite, 2014) is used to create and render such fonts.

system also has a description language in XML terms for doing many things, such as if ... then ... else logic, string manipulation and list checking. It also helps modularize common components of rules using a macro system. More details are in Section 4.4.4. One of the powerful things this language allows is being able to refer to the source items or the target items at the word or morpheme level.

3.2.3 Ambiguity

In terms of managing the ambiguities produced in the analysis of the source language, **FLEx** is a good solution. In addition to being a lexicography tool, **FLEx** is also a text analysis tool. It has a text repository and a facility to analyze a text. When I use the term analyze, I use it in the narrow sense that **FLEx** uses which means breaking a word into its constituent morphemes. The analyzing of words into morphemes can be either done manually or automatically using built-in parsers. Section 5.2 goes into more details. Each analyzed morpheme has a link to a lexical entry's sense in the lexicon which makes it convenient to determine which entry and sense is being used for the words in the text. Naturally each word in the text can have ambiguous parses, but the user can decide which parse is correct in the given context. With **FLEx** we can let the user disambiguate the words to have the correct sense and category. This way we can ensure that we have a completely unambiguous text coming into the transfer process.

One may argue that a manual approach to disambiguation at this level is a huge amount of work. Fortunately, **FLEx** has features in place to minimize the work involved. First, if automatic parsing is used, many spurious parses will be eliminated. I admit that setting up automatic parsing is some work initially, but it pays dividends in the long run especially if you will be doing a lot of parsing, which will be the case if you plan to translate many texts. The setup of automatic parsing can be done incrementally. Decent results can be achieved with a moderate amount of setup and as more constraints are added, the parsing will improve. Secondly, there is the manual component where a user can override incorrect parses suggested by the automatic parser. With or without automatic parsing, the **FLEx** application memorizes each parse that the user approves and the next time the same surface form is encountered, it will suggest the memorized parse. The user only has to change the analysis if he or she decides a different parse is appropriate. The more texts that are disambiguated, the larger the memorized parse database grows and the faster the work goes.

3.2.4 Transfer Process

The transfer process is handled by the **Apertium** transfer engine. The engine needs three things: 1) a bilingual lexicon, which we can create from the linked senses between the two **FLEx** projects, 2) a rule file as discussed in Section 3.2.2 and 3) an analyzed input text where the words are analyzed into morphemes. This can be extracted from **FLEx**'s text repository. We have all the pieces the engine needs.

3.2.5 Synthesis Process

The synthesis process is handled by a tool called **STAMP**. **STAMP** is part of a suite of older SIL International tools that do MT. I am just using the synthesis component of **STAMP** which takes an input text that is analyzed into morphemes and a lexicon containing all the morphemes. The text comes from the transfer process and the lexicon is extracted from the target **FLEx** project. **STAMP** is a good solution for synthesis since it understands the same syntax for environment constraints as is used in **FLEx**. An environment constraint is a description of the phoneme context in which a morpheme occurs. It also has the same constructs for natural classes of phonemes that **FLEx**'s XAMPLE parser uses.

3.2.6 Tool Interaction

Using these different third-party tools requires some programs that can serve as the glue to bring data from one repository or process to another. **FLEx** has built into it an application programming interface that allows an external program to read and write the contents of a **FLEx** database. Using programs written in the **Python** language I am able to extract data from **FLEx** and also insert data. I have written six **Python** scripts that are detailed in Section 5.9 for the MT system. For the execution of the **Python** scripts I use the **FlexTools** environment. It is a system that works closely with **FLEx** and provides a **Python** library for **FLEx** access as well as a way to manage and execute **Python** scripts. The installation instructions for **FlexTools** is outlined in Appendix Section A.1.1.

A final comment on the basic concept of this solution is that it is flexible for future developments. By having a **FLEx**-centric system where **FLEx** serves as the repository for lexical and text data, the other tools for MT, i.e. for the analysis, transfer or synthesis processes, can be replaced without affecting the core data.

3.2.7 Incremental Power and Complexity

An aspect of this system that makes it linguist-friendly and very flexible is how the system's complexity and thus its power can be progressively increased. A linguist is less likely to use a system that is very complex and that requires extensive setup just to get started. But a system that will do MT at a basic level and then be able to progressively do more would be happily accepted by linguists. Having the bar low to getting basic MT functioning encourages linguists to try the system and evaluate its utility. Experimentation can be done at a relatively low cost and if the linguist sees good potential for doing MT with a language pair, the next steps in complexity and power can be taken.

At the basic level, a person can use this system to do lexical substitution of words. To do this, all that is necessary are two lexicons, the links established between source and target word-senses and a manually parsed text. If the language

pair is closely related where affixes are similar, even affixation will be correctly transferred into the target language.[2] By default, source word-senses that are not linked are carried over to the target language. With all this, a linguist would have a basic functioning system of MT.

A next step the linguist can take is improving the analysis system in **FLEx** so that fewer manual parses have to be done on the source text. Incrementally constraints can be added to the source **FLEx** project producing better parsing results. The linguist can get to the point where all the words are correctly parsed by **FLEx**'s automated parser and the linguist just has to manually adjust ambiguous parses.[3]

If the linguist wants to go a step further, he or she can start writing transfer rules. The whole paradigm of transfer rules is incremental. The linguist starts with a simple rule to handle a common phenomenon when translating between the two languages. Gradually more rules are introduced to handle more phenomena. With each addition the linguist sees better results from adding rules. The transfer system becomes increasingly complex with each rule that is added. One rule may cancel out another rule, so the whole collection of rules should be refined together as rules are added.

At the same time that the analysis system is improved and the rule system is expanded to handle more phenomenon, the synthesis system can also be improved. The target lexicon at a basic level, does not have to have any environment constraints on allomorphs; and the result is that the first allomorph of a word is always chosen in the synthesis process. The linguist can incrementally add environment constraints so that the synthesis process chooses the correct allomorph for a given phonetic environment.

The incremental nature of this MT system makes it easy for linguists to get their feet wet in MT. As needed more can be added to the MT model for a given language pair to make it more powerful and produce better results.

[2] This assumes that the affix glosses are the same in both languages.

[3] As stated above, previously memorized parses help the linguist choose between ambiguous parses based on what was previously done. The disambiguation process becomes more efficient over time.

4

Tools

This chapter introduces the core third-party tools used in this MT system: **FLEx**, **Apertium** and **STAMP**. Following this is an explanation of the initial setup needed to use the system.

4.1 FLEx

FLEx is a tool that facilitates the recording and analysis of linguistic and anthropological data.[1] It provides a well-ordered set of fields to record data. It also provides a means of recording a text corpus with the built-in capability to interlinearize text. Running an optional automated morphological parser can make interlinearizing text an efficient task. Grammar tools are also available which allow the user to capture grammatical information. This information is used by the morphological parser, providing a way to check grammatical rules recorded against real language data. The grammar information can also be compiled in an automatically generated grammar sketch.

Here are some things you can use **FLEx** for:

- Inputting lexical information for the development of a lexicon
- Recording vernacular texts and viewing a concordance of words within recorded texts
- Creating an interlinear analysis of texts
- Analyzing morphology
- Creating a grammar sketch
- Recording anthropological field notes
- Rapidly entering words as part of a rapid word collection workshop[2]

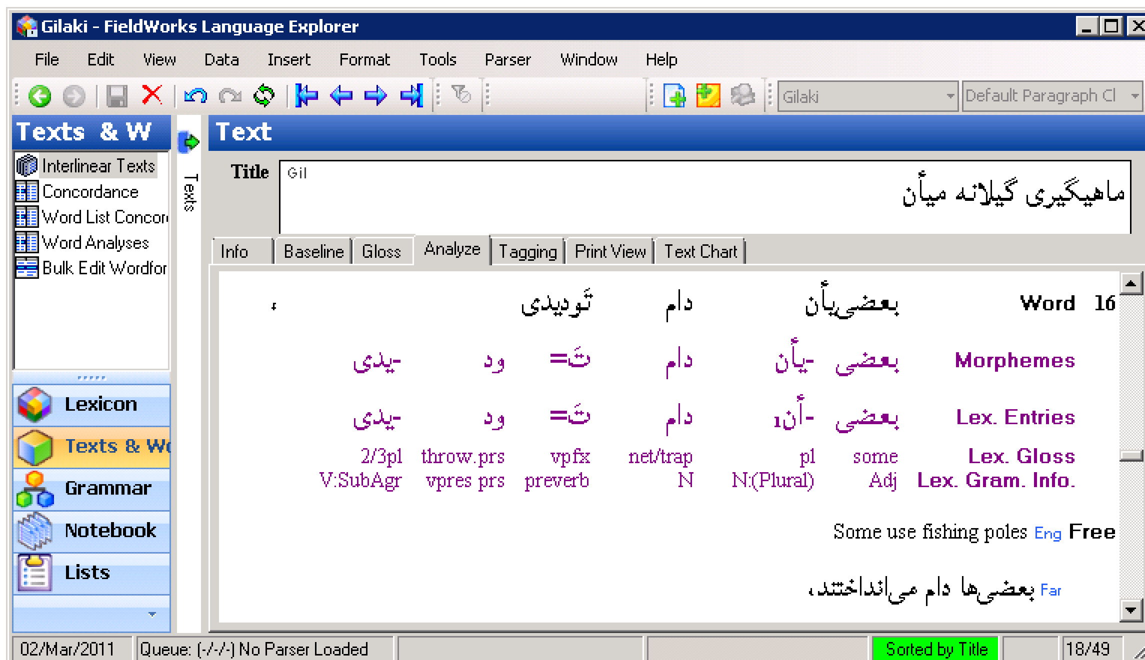
[1] In version 6.0 and earlier the Fieldworks software suite included a tool called Data Notebook for recording and analyzing anthropological data. In version 7.0 and beyond, this tool has been integrated into **FLEx**.

[2] See http://rapidwords.net for more information.

Two screen shots of the **FLEx** are shown in (1) and (2).

(1) Lexicon edit view showing a sample lexicon entry

(2) Interlinear text analysis view showing an interlinearized sentence



FLEx works in both Windows and Linux.

4.2 Apertium

Apertium was developed as one of the open-source MT systems which originated within the project “Open-Source Machine Translation for the Languages of Spain” (“Traducción automática de código abierto para las lenguas del estado español”) (Forcada et al. 2010). The first paper that describes **Apertium** as a platform was published in 2006, although the origins of the system date back earlier than that (Ramírez-Sánchez, 2006). **Apertium** is an MT system designed originally for related languages, but later it added tools to facilitate unrelated language MT.

Even though the MT system described in this thesis only uses the shallow-transfer engine of **Apertium**, the platform is a full-fledged system for doing MT. It is at its core an analysis-transfer-synthesis system. The shallow-transfer engine is described in Section 5.9.2, but the other parts of the full system include finite-state transducers for lexical processing which are used for both analysis and synthesis, hidden Markov models for part-of-speech tagging and finite-state-based chunking for structural transfer. The system is fast, boasting a throughput of tens of thousands of words per second on an ordinary desktop computer (Forcada et al. 2010).

Apertium also has a web-based front end (Apertium, 2015). From this website, you can translate texts or documents in 40 language pairs. Since **Apertium** is an open-source platform, new language pairs can be added by anyone at anytime. There are stages a language pair goes through until it is deemed mature enough to be called stable and

therefore available on the website. Besides the 40 stable pairs, there are many pairs in various stages of development. There is quite a lot of documentation on the website and in Forcada et al. 2010, but I found the most useful help in the live IRC chat page which is manned by experienced users. It is here that practical problems can be solved.

4.3 STAMP

STAMP as seen in the title of Weber et al. 1988, is a tool that was designed originally for dialect adaptation. It is essentially the transfer-synthesis part of the analysis-transfer-synthesis paradigm and was designed to work in conjunction with **AMPLE** as described in Weber et al. 1990. Together **AMPLE** and **STAMP** make the complete MT system which has come to be known as CARLA. **STAMP** has three modules: transfer, synthesis and "textout". The MT system described in this thesis just uses the synthesis module which is described in Section 5.9.3. **STAMP**'s transfer module does the jobs of lexical substitution between source and target lemmas as well as deletion, insertion and reorderings of morphemes. The "textout" module simply formats the results of the synthesis module as plain text.

4.4 Initial Setup

There are several things that have to be in place before you can run the MT system: lexicons in both languages, grammar settings, linked senses between the source and target lexicons and transfer rules. You must also install and configure the MT system. See Appendices A and B. Let us look at these things one by one.

4.4.1 FLEx Lexicons

The basic requirement for the source lexicon is that all the stems and affixes in use be recorded in the lexicon.[3] Each entry should include, at a minimum, the lexeme form and one sense.[4] Each sense must have a grammatical category assigned and a gloss.[5] For the affix and clitic entries, it is required that their glosses be unique. Clitics are treated as affixes in this MT system. Those are the minimum requirements, but more information can be entered into each lexical entry. Inflection features or inflection classes are especially useful. These apply to the entire root entry. The MT system will automatically export the inflection features and classes as tags on a word just like it does for affixes. The inflection feature and class abbreviations need to be unique among themselves and the affix glosses. In this way, a transfer rule can uniquely refer to any tag which represents a feature or affix for a word.

[3] A stem does not have to be in the lexicon if you choose to treat it as an unknown word. See Section 5.6.5.
[4] In the case of variant entries, the sense can be omitted as long as the variant entry is linked to a main entry that has a sense.
[5] Technically a gloss is not required for stem entries, but it would be difficult to analyze the text in **FLEx** without this.

On the target side, the lexicon must at a minimum have an entry and sense for every source sense that will be linked to it. As in the source lexicon, every entry needs to have a lexeme form and a sense with a grammatical category and a gloss. Similarly, affix glosses must be unique. Probably the most important consideration for the target lexicon is that allomorphs are defined appropriately. This only applies when there exists more than one allomorph for an entry. In such a case the additional allomorph or allomorphs are added to a section of the entry called Allomorphs. See example (3). It is important to remember that in **FLEx**'s parsing system and in this MT system the order of the allomorphs is significant. For **FLEx**, the first allomorph evaluated is the first one in the Allomorphs section. The next is the one below it and this continues until there are no more allomorphs in the Allomorphs section. The last allomorph to be evaluated is the lexeme form. In other words, the lexeme form is the "elsewhere" allomorph. In (3) an entry is shown that contains three allomorphs, the first two of which have environment constraints.

(3)

Lexeme Form	Gil	ه
Morph Type		suffix
Sense 1		
Gloss	Eng	3sg
Grammatical Info.		v:SubAgr
Allomorphs		
Affix Allomorph	Gil	يه
Environments		/ [VMinusYe] _
Affix Allomorph	Gil	يه
Environments		/ [Ye] _

When evaluating allomorphs in the order described above, **FLEx** assumes a negation of all the previous allomorphs' environment constraints as part of the current allomorph's environment constraint (Black, 2014). The lexeme form will have the cumulative negated environment constraints of all the other allomorphs in addition to it having possibly its own environment constraint or constraints. In practical terms this means that you should put the allomorph with the most restrictive constraints first; subsequent allomorphs should be less restrictive. It is good practice to test the parsing of a word or affix in its different allomorph forms.[6] If a form does not parse in **FLEx**, it will also not be able to be synthesized by **STAMP** in the MT system. See Section 5.9.3.3 for more information.

[6]This can be done by selecting *Parser* → *Try a Word* from the menu in **FLEx**, entering the word form and clicking the ***Try it*** button.

4.4.2 FLEx Grammar Settings

Strictly speaking, the only setup needed in terms of grammar settings in FLEx are: 1) having all affixes and clitics in the lexicon and 2) having grammatical categories defined for all entries in the lexicon. With these things in place you can run the MT system. This would be a minimum setup, however and would mean for the analysis phase, you would need to manually parse words into morphemes. To take advantage of automated parsing, you should employ the strategies discussed in Section 5.2.2.

4.4.3 Linking Senses

The bilingual lexicon that the MT system uses is based on links set up between source and target word-senses. **FLEx** has a built-in feature for linking two entries. A prerequisite to doing this linking operation is creating two custom fields in the source **FLEx** project, one to hold the link to the target entry and one to hold the sense number being linked to in that target entry. The custom fields should be sense-level fields, type: "Single-line Text", writing system: "First Analysis Writing System." The names of these custom fields are up to the user and should be recorded in the configuration file, see Appendix B. The procedure for linking is as follows: Navigate to the target entry to be linked to, select *Copy Location as Hyperlink* from the *Edit* menu, navigate to the source entry's pertinent sense, click in the link custom field and select *Paste Hyperlink* from the *Edit* menu. Set the sense number in the sense number custom field.[7] (4) shows a source entry that is linked to a target entry. "Gilaki Sense Number" and "Gilaki Equivalent" are the custom fields.

(4)

⊟ Sense 3	
Gloss	Eng **call.pst**
Definition	Eng
Grammatical Info.	Verb pst
Gilaki Sense Number	2
Gilaki Equivalent	silfw://localhost/link?app%3dflex%26database%3dGilaki%26server%3d%26tool%3dlexiconEdit%26guid%3d74af40a2-6f5b-4d0b-96e8-b0759921a6fa%26tag%3d

4.4.4 Transfer Rules

In order to rearrange words and morphemes using the **Apertium** transfer engine you must supply a transfer rules file. The contents of this file is well-documented in Forcada et al. 2010 with additional information found at http://wiki.apertium.org. The file is in an XML format. A sample rule is shown in (5). I describe the rule file in Section 5.8.

[7] If this field is left blank, the first sense of the target entry will be used by default.

(5)

```
<section-rules>
  <rule comment="Handle noun or adjective plus the postposition rā.">
    <pattern>
      <pattern-item n="nom-adj"/>
      <pattern-item n="post"/>
    </pattern>
    <action>
      <out>
        <lu>
          <clip pos="1" side="tl" part="whole"/>
          <lit-tag v="acc/dat"/>
        </lu>
      </out>
    </action>
  </rule>
```

5

Implementation

This chapter details how this MT system is implemented. First, I give an overview of the system, followed by a description of design features. Next, I describe some limitations of the system and give an overview of the transfer rules file. Finally, I describe all the modules that do the work. See Appendix A for installation and running instructions.

5.1 System Overview

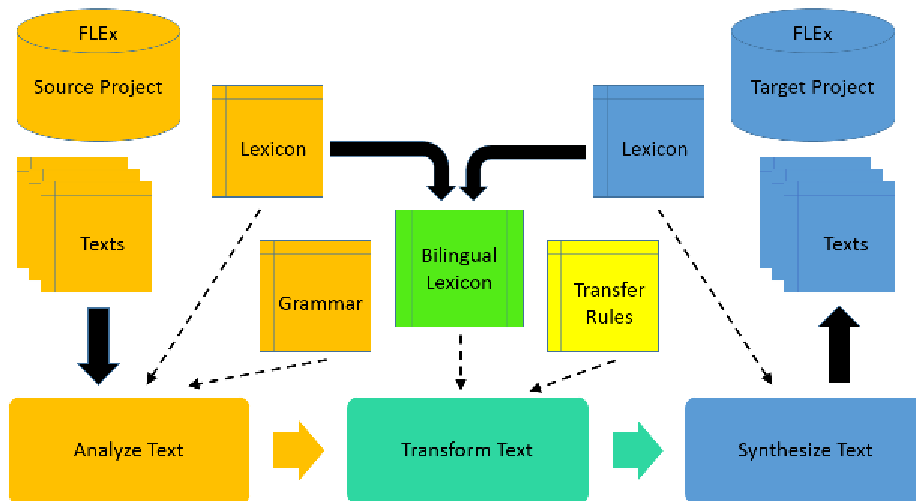


Figure 1

The MT system is composed basically of two language databases, a list of transfer rules and a series of programs that do the work. The language databases, one for the source language and one for the target language are in the form of **FLEx** projects. A **FLEx** project stores among other things a lexicon, grammar settings and a database of texts. The transfer rules are in the form of a text file in XML format. The core programs are the analysis engine, the transfer engine and the synthesis engine. A text in the source language first goes through the analysis process where words

are broken down into morphemes. Then the "analyzed" text is transformed into target morphemes and rearranged as necessary in the transfer process. Lastly, the target morphemes are put together into target words in the synthesis process. The end result is a text in the target language.

The MT system uses the analysis-transfer-synthesis paradigm that is used in many MT systems. At its core, the MT system is made up of an engine for each part of the paradigm as well as **Python** scripts that massage the data and help the three engines work together. Let us examine the inputs and outputs for each part of the system.

The analysis engine, which is part of **FLEx**, takes as inputs a text, a lexicon and grammar settings. The output is an analyzed text, i.e. a text that has been broken down into morphemes where each morpheme is linked to an entry in the lexicon, specifically a sense of that entry. The **Apertium** transfer engine takes as inputs the analyzed text from the analysis process, a bilingual lexicon and a list of transfer rules. The output is a text made up of morphemes in the target language. The synthesis engine which is performed by **STAMP**, takes as inputs the text from the transfer process and a lexicon. The output is a text of surface forms. Let us now look at each step in more detail.

5.2 Analysis

The analysis step is done completely in the **FLEx** application which is a powerful program for analyzing texts. Note that analysis is done at the word-level only. As discussed in Section 3.2.3, **FLEx** has a computer-assisted manual method for parsing words as well as a morphological parser for doing automated parsing.

One nice thing about **FLEx**'s system for analyzing texts is that a fully defined grammar is not necessary to get started doing parsing. In fact, without any grammatical information at all, you can do manual parsing. As your knowledge of a language's grammar increases, grammar settings can be entered which will help the system do better automated parsing. In this way, you can gradually cover more of the grammar and get better parsing results. Let us look at manual parsing initially.

5.2.1 Manual Parsing

To begin with, an unanalyzed word looks like (6).

(6)

Word	goftim
Morphemes	***
Lex. Entries	***
Lex. Gloss	***
Lex. Gram. Info.	***

To divide this word into morphemes you click on the morphemes line under the word and type a hyphen where a suffix or prefix starts or ends. **FLEx** will then divide the word into two parts. It will also look for that morpheme in

the lexicon and if it is found it will fill out the other lines of information. (7) shows the word after the first hyphen was typed; (8) shows the word after all the hyphens were entered and the parse was approved.

(7)

Word	goftim	
Morphemes	gofti	-m
Lex. Entries	***	-əm
Lex. Gloss	***	1sg
Lex. Gram. Info.	***	V:SubAgr

(8)

Word	goftim		
Morphemes	goft	-i	-m
Lex. Entries	goft	-i$_2$	-əm
Lex. Gloss	say.pst	ipfv	1sg
Lex. Gram. Info.	vpst pst	vpst:Impfv	V:SubAgr

As mentioned in Section 3.2.3, **FLEx** has a very helpful memorization feature. Once a word has been parsed and approved by the user, it is memorized and stored in a database. The next time the word is encountered, **FLEx** suggests that previous parse to the user. This can be accepted or modified. In the case where multiple parses for a word have been memorized, **FLEx** suggests the most commonly approved parse in the current text. Suggested parses drawn from the database of parses are shown in blue as in (9).

(9)

Word	goftim		
Morphemes	goft	-i	-m
Lex. Entries	goft	-i$_2$	-əm
Lex. Gloss	say.pst	ipfv	1sg
Lex. Gram. Info.	vpst pst	vpst:Impfv	V:SubAgr

Each morpheme is tied to a lexical entry and sense. In fact it is impossible to "finish" an analysis of a word without adding missing morphemes to **FLEx**'s lexicon. (10) shows an unfinished analysis. Note that one of the morphemes has asterisks for the lines under it. This indicates **FLEx** could find not a lexical entry for it so the entry for this suffix will have to be added.

(10)

Word	goftim		
Morphemes	goft	-i	-m
Lex. Entries	goft	***	-əm
Lex. Gloss	say.pst	***	1sg
Lex. Gram. Info.	vpst pst	***	V:SubAgr

This is the manual method of parsing. You can parse each word in the text this way. When each word has been parsed, the text is ready to be used as input to the transfer process of the MT system. Let us now look at the automated method of parsing.

5.2.2 Automated Parsing

Doing automated parsing can greatly speed up the job. **FLEx** has two morphological parses that you can use (Black, 2014). The default parser is called XAMPLE and uses an item and arrangement approach. This means that the **FLEx** parser looks at the surface form of the word and parses it as many ways as it can using forms from the lexicon. There is another parser called HermitCrab.NET. It uses an item and process approach where you can specify underlying forms of morphemes along with phonological rules that produce surface forms. With either parser, you enter information into the grammar settings of **FLEx** to constrain the parser to select the morphemes that are appropriate for the given word. The more constraints you add, the better your results will be.

The parser will be somewhat helpful even if there are no grammar settings. It will find possible roots and affixes that would produce the surface form you are working on and produce a list of possible parses. In **FLEx** a word parsed by the parser might look like (11).

(11)

Word	goftim		
Morphemes	goft	=i	=m
Lex. Entries	goft	=i	=m
Lex. Gloss	say.pst	ind	be.1sg
Lex. Gram. Info.	vpst pst	prt	cop

The orange background indicates that the parser suggested this parse. The yellow highlighting indicates that there were multiple parses found. You can look at the list of parses for this word and it may look something like (12). In the text itself you can click on the arrow button to choose which parse is appropriate. See (13).

(12)

⊟ Analysis Candidate 1

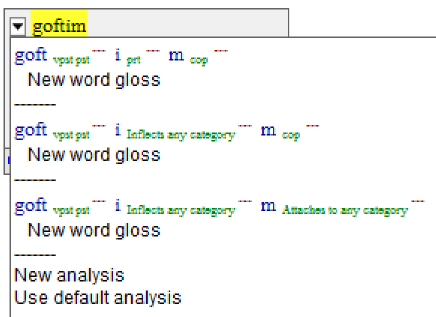
Analysis	Morphemes	goft	=i	=m
	Lex. Entries	goft	=i	=m
	Lex. Gloss	say.pst	ind	be.1sg
	Lex. Gram. Info.	vpst pst	prt	cop
Parse result	Successful			

⊟ Analysis Candidate 2

Analysis	Morphemes	goft	-i	=m
	Lex. Entries	goft	-i	=m
	Lex. Gloss	say.pst	ipfv	be.1sg
	Lex. Gram. Info.	vpst pst	Inflects any category	cop
Parse result	Successful			

⊟ Analysis Candidate 3

Analysis	Morphemes	goft	-i	-m
	Lex. Entries	goft	-i	-m
	Lex. Gloss	say.pst	ipfv	1sg
	Lex. Gram. Info.	vpst pst	Inflects any category	Attaches tc
Parse result	Successful			

(13) 

Some of the constraints you can enter into the grammar settings are as follows: 1) Affix templates. These allow you to specify which affixes can attach to a grammatical category and what their order is. 2) Inflectional features. These allow you to tag a lexical entry with one or more features. The parser prevents co-occurrence of morphemes where the features are not compatible. 3) Inflectional classes. Similar to features, you can tag entries with a certain class. The parser prevents co-occurrence of morphemes where the classes are not compatible. 4) Allomorph environment constraints. These allow you to control which allomorphs can occur based on the surrounding phoneme context. 5) Clitic attachment constraints. Here you can allow clitics to attach only to certain grammatical categories. 6) Ad hoc constraints. These allow you to handle idiosyncratic situations where you want to prevent two morphemes or two allomorphs from co-occurring. See Parser, 2011, Black, 2014 and Lockwood, 2011 for more information about constraints.

Using these constraints, you can greatly reduce the number of ambiguous parses. Valid ambiguous parses may still occur. In this situation, **FLEx** helps you by suggesting a parse from its list of memorized parses for a given word. You can always override **FLEx**'s suggestions when necessary. With a good set of parsing constraints in place, a text can be parsed very efficiently. The end result required for the next stage of the MT system is one approved parse for each word in the text. See Fieldworks Movies, 2009 for a video about the using the FLEx parser.[1]

5.3 Transfer

The transfer step is performed by **Apertium**'s transfer engine which is a component of the **Apertium** MT platform. See Section 4.2 for an overview of **Apertium**. There are two systems for doing transfer with **Apertium** one is the shallow-transfer system and the other is the advanced-transfer system. The following excerpt explains it.

[1] You can leave a word unparsed if you want to treat it as an unknown word. See Section 5.6.5.

> The two transfer systems differ in the number of passes over the input text. The shallow-transfer system makes structural transformations with a single pass of the rules, which detect sequences or *patterns* of lexical forms and perform on them the required verifications and changes. On the other hand, the advanced transfer system works with a new architecture that allows to detect [sic] *patterns of patterns* of lexical forms with three passes, done by its three modules (Forcada et al. 2010).

As mentioned above, the transfer engine takes as inputs the analyzed text from the analysis process, a bilingual lexicon and a list of transfer rules. The job of the transfer engine is twofold: First, it does a lexical substitution, converting each source word in the input text to the appropriate target word as defined by the bilingual lexicon. Second, it uses the transfer rules to detect patterns in the text and for a given pattern operates on the source words to output new words. Many things are possible in the transfer process. Morphemes within a word can be rearranged. Words can be rearranged. Morpheme or words can be inserted or deleted. The system is quite powerful. See Section 5.8 for more details.

Let us discuss the inputs in more detail. The transfer engine operates on the analyzed text that comes from the source **FLEx** project. I use a **Python** script to pull the analyzed text out of the **FLEx** project and put it into the textual data stream format that **Apertium** expects. See Section 5.9.1.1 for how this works. The data stream consists of a headword identifier for the stem morpheme (also called the lemma) plus one or more tags following the headword. The first tag is always the grammatical category of the word taken from the source lexicon. Additional tags can include inflection features, inflection classes, then any prefixes/proclitics and any suffixes/enclitics. (14) shows two words in this data stream format. The whole collection of lemma and tags is called a lexical unit and is enclosed in `^` and `$` characters.

A second input, the bilingual lexicon, is a text file that contains mappings between source and target word-senses. I extract the bilingual lexicon with a **Python** script as described in Section 5.9.1.2. Word-senses get linked to each other by creating links in the source lexicon. See Section 4.4.3 for a description of how to link word-senses.

The last input to the transfer engine is a list of transfer rules. The language used to express the rules is described in Section 5.8. A sample rule is shown in (16). The basic idea is that a pattern is defined as well as the action that should take place if that pattern is found. The patterns usually correspond to grammatical categories.

The output of the transfer engine is a text file similar to the input text file, i.e. a data stream. The difference is that the transfer engine has replaced the source lemmas with target lemmas and the morphemes have been modified as prescribed in the transfer rules. Here is an example.

Suppose we have an input data stream containing two words as shown in (14). Given the rule in (16) and assuming a bilingual lexicon that maps the lemma `sib1.1` to `səb1.1`, the transfer engine would produce the data stream shown in (15).

(14) ˆsib1.1 <n>$ ˆrā1.1 <post>$

(15) ˆsəb1.1 <n><acc/dat>$

(16)
```
<section-rules>
  <rule comment="Handle noun or adjective plus the postposition rā.">
    <pattern>
      <pattern-item n="nom-adj"/>
      <pattern-item n="post"/>
    </pattern>
    <action>
      <out>
        <lu>
          <clip pos="1" side="tl" part="whole"/>
          <lit-tag v="acc/dat"/>
        </lu>
      </out>
    </action>
  </rule>
```

Briefly explained, since we have a noun followed by a postposition (<n>, <post>), the rule pattern[2] is matched and the action is executed. A lexical unit (<lu>) is outputted that consists of two things: 1) The entire lexical unit (part="whole") of the target language (side="tl") of the first word in the pattern (pos="1") and 2) a tag with the string acc/dat.

That is basically how the transfer process works. An input data stream enters and using the mappings in the bilingual lexicon and the list of rules, the transfer process produces an output data stream that is ready to be synthesized.

One note about ambiguous translations for a word-sense; you may have a situation where one source word-sense has two possible translations. You can write transfer rules specifically to choose a particular target word-sense for a given source word-sense in a particular context. Such a rule would overwrite the default mapping of source word-sense to target word-sense in the bilingual lexicon.

5.4 Synthesis

The synthesis step is performed by **STAMP**, a program designed for transfer and synthesis. See Section 4.3 for an overview of the tool. **STAMP** takes as inputs, the text file from the transfer engine and the target lexicon. **STAMP** cannot work directly with **FLEx** and it cannot handle the text file format coming out of the transfer engine. I wrote two **Python** scripts. One converts the target **FLEx** lexicon to **STAMP** format and the other converts the **Apertium**-style data stream to **STAMP**'s format for an input text. See Section 5.9.3.3 and Section 5.9.3.2 respectively for more details.

STAMP's job is basically to take the target language morphemes and put them together to make surface word forms. It may sound trivial to take a root and join it with prefixes and suffixes, but there is more to it. The text data stream

[2] At the top of the rule file the category nom-adj is defined as a list of noun and adjective categories of which n is one.

that has at this point been converted to a **STAMP** file, has records containing root information and affix information. The root information has two components: a grammatical category and a headword. These are essentially a key for looking up an entry in the extracted target root lexicon. **STAMP** looks up the root in the root lexicon and examines what allomorphs exist for the entry. If there is just one allomorph it is used as the output string.[3] If however there is more than one allomorph, **STAMP** chooses the right one based on the environment constraints that are present. The first allomorph to successfully pass the environment constraints is used. For roots it is more common to have only one allomorph.

STAMP treats the affix information similarly except in this case **STAMP** uses the name of the affix, which is a unique gloss, to look up the affix in the appropriate lexicon file. **STAMP** again looks for allomorphs and often there is more than one. Just like in the case of roots, it checks to see which allomorph is the first to satisfy its environment constraints. That is the affix string that is used for the surface form.

It is actually more complex than that. **STAMP** has to simultaneously check environment constraints on the root and the affixes and see which combination of root and affix allomorphs meet all the constraints.[4] The end result of the synthesis process is a text file of surface forms in the target language. The synthesis process is deterministic. The same input will always produce the same output.

5.5 Full Example

Let us take an example text and go through all three core engines. Let us start with the very simple text in Persian: sib rā xordam meaning ‘I ate the apple.’ Our goal is to translate this to Gilaki which is seba buxurdəm. In **FLEx** we would start out with a text that looks like (17). In this case I already had sib and ra in my lexicon so they appear as suggested parses with a blue background. The verb however, was not recognized since I had not parsed it before.

(17)

Word	sib	rā	xordam
Morphemes	sib	=rā	***
Lex. Entries	sib	rā	***
Lex. Gloss	apple	om	***
Lex. Gram. Info.	n	post	***

I run the parser and **FLEx** suggests a parse for the verb as shown in (18).

(18)

Word	sib	rā	xordam	
Morphemes	sib	rā	xord	-am
Lex. Entries	sib	rā	xord	$-am_1$
Lex. Gloss	apple	om	eat.pst	1sg
Lex. Gram. Info.	n	post	v pst	v:SuAgr+

[3] Unless, of course, there is an environment constraint on the one allomorph in which case the constraint would need to be satisfied.

[4] **STAMP** has a built-in test to check if allomorphs pass environment constraints (in **STAMP** terminology “string environment constraints”). If desired you can write your own test to handle complex situations. See Weber et al. 1990.

I am happy with all the suggested parses and I “approve” all of the them. This gives me a text that looks like (19).

(19)

Word	sib	rā	xordam	
Morphemes	sib	rā	xord	-am
Lex. Entries	sib	rā	xord	-am$_1$
Lex. Gloss	apple	om	eat.pst	1sg
Lex. Gram. Info.	n	post	v pst	v:SuAgr+

When the MT system extracts this text into **Apertium**’s data stream format it looks like this:

```
^sib1.1<n>$ ^rā1.1<post>$ ^xordan1.1<v><pst><1sg>$
```

I now run the transfer part of the system. Two rules are applied against this text, the rule shown in (16) and another rule for the verb. At the same time, Gilaki equivalents are looked up in the bilingual lexicon. The result is this:

```
^seb1.1<n><acc/dat>$ ^xurdən1.1<vpst><pfv><1sg>$
```

At this point the MT system converts the text to **STAMP** format and extracts the target lexicon. As a last step, the MT system runs **STAMP** to do synthesis and the result is: seba buxurdəm which is what we wanted.

5.6 Design Features

Beyond just getting the applications talking to each other, my efforts to make this an effective MT system include several design features.

5.6.1 Bilingual Lexicon

One key design feature of this MT system is how words are linked together to make a bilingual lexicon. If MT is being done between related languages, much of the vocabulary between the languages is shared and even spelled the same. It is very helpful if in the bilingual lexicon, by default, source word-senses map to the exact same target word-senses. This is exactly how I designed the system. You only need to make links between source and target word-senses if there is some kind of a difference between the two. The difference could be in spelling, in grammatical category or in sense. Examples of bilingual lexicon entries for linked and unlinked word-senses are shown in Section 5.9.1.2.

5.6.2 Complex Entries

Another design feature of this MT system is the way complex entries are handled. A complex entry is an entry composed of two or more morphemes (Moe, 2009). In **FLEx**, the way complex entries work is that an entry can have links to component entries that make up the complex entry. In MT you have cases where a complex entry needs to be translated as a whole to something in the target language. When analyzing a text with **FLEx** you can analyze the parts of a complex entry without reference to the fact that the parts are elements of a complex entry that has a composite

meaning. This MT system is designed so that you can map a complex entry (and sense) to a word-sense in the target language and still get the benefit of the morphological analysis of the complex entry's components in the text. The system will look for adjacent words that are the components of a complex entry and convert the multiple word analyses to a single lexical unit for processing by the transfer engine. In the transfer process, the bilingual lexicon is applied to map the whole lexical unit to the appropriate target word or words.

On the target side, it is also possible to have complex entries and you can map a source word-sense to a target complex entry (and sense). In this case, the process is reversed. A multiple word lexical unit that comes out of the transfer process is examined and if it matches a target complex entry it is separated into its component elements for processing by the synthesis engine.

Not every kind of complex entry requires this decomposition and recomposition process. Many complex entries have no morphology and can be transferred straight across. To differentiate between the two kinds of complex entries, I rely on the linguist assigning different complex form types in the **FLEx** lexicon. In the configuration of this MT system, you can specify which complex form types are applicable to the decomposition and recomposition process. See Sections 5.9.1.1 and 5.9.3.3 for more details.

5.6.3 Grammatical Category Mapping

Another design feature is the ability to do a wholesale mapping of one grammatical category to another. It may be the case in some language pairs that grammatical categories have different names for the same categories. Or it may be the case that the source lexicon has the language categorized in such a way that two grammatical categories are collapsed to just one in the target language. This MT system allows you to specify a source category and a target category that should be mapped to each other. The outworking of this is that the bilingual lexicon is changed. Bilingual lexicon source entries that already have a map to a target entry are not touched. Just those entries that have the default mapping where source and target are the same in spelling and meaning will be modified to have a different grammatical category for the target. See Section 5.9.1.2 for more details.

5.6.4 Bilingual Lexicon Customization

The ability to make custom changes to the bilingual lexicon is another feature of this MT system. The bilingual lexicon is basically the links that exist between source and target lexicons. This is turned into a file for input into the transfer process. Sometimes you want to make a custom mapping of a source word to a target word. This MT system has a step in the process where you can apply custom changes to the bilingual lexicon before the file is sent to the transfer process. There are two customization options. One is the adding of new lines in the bilingual lexicon file and

the other is replacing an existing line with a customized version. Both of these options are made possible through a special file that contains a section for additions and a section for replacements. See Section 5.9.1.2 for more details.

5.6.5 Unknown Words

Another design feature is the handling of unknown words. Normally each word in the source text is analyzed into its constituent morphemes in the analysis step. If, however, a word is not analyzed, it will be treated as an unknown word. The word will get sent to the transfer process in the form `word<unk>` (instead of the grammatical category, `unk` is outputted.) This word will not be found in the bilingual lexicon and will come out of the transfer process as `@word<unk>`. You choose how you want the target output to look. Unknown words can either be outputted as `@word` in the target text, or the `@` can be stripped out and just `word` would appear in the target text. You control this in the configuration file. This feature can be very useful if you do not want to add certain words like proper nouns to the source lexicon and you are happy for them to have the same exact form in the target language. See Sections 5.9.1.1 and 5.9.3.4 for more details.

5.6.6 Inflection Information

Another design feature is the inclusion of inflection information in the transfer process. **FLEx** has a rich set of inflection information that can be included in each lexical entry. Inflection classes and inflection features are two examples. This information is important to have in the transfer process. Transfer rules can make use of it in determining how to transform the source morphemes into target morphemes. The way this is implemented is that source inflection information is included in the source text extracted from **FLEx** and target inflection information is included in the bilingual lexicon that is also extracted from **FLEx**. See Section 5.9.1.1 and Section 5.9.1.2 for more details.

5.6.7 Variant Entries

Another design feature is how this MT system handles variant entries. I leverage **FLEx**'s system for variant entries. In **FLEx**, a variant entry typically does not have sense information with it, but instead is connected to a main entry that has all of the sense information. In any text it is very possible to encounter variant forms, whether they be spelling variants, dialect variants, inflectional variants or other kinds of variants. Since in **FLEx** a variant form points back to a main entry, it is not necessary to link both a variant and its main entry on the source side to a target entry. The MT system is designed so that just a link from the main source entry is necessary. When the system encounters a variant form in the source text, it follows the connection from variant entry to main entry in the lexicon and the main entry

information is outputted to the extracted text. The system will handle an unlimited number of variants of variants. See Section 5.9.1.1 for more details.

5.7 Limitations

There are several limitations of this MT system. As currently implemented, this system assumes the use of **Apertium**'s shallow transfer engine. This means that phrase-level rearrangements or changes from source to target language might not be possible. Given this limitation, closely related language translation would be more appropriate for this system. That said, it would not be very hard to extend this MT system to use **Apertium**'s advanced transfer engine. The main change would be in the rule files. Instead of using one rule file, you would use three rule files and the rule files would make use of **Apertium**'s "chunking" methodology. The transfer engine would be executed one time for each of the three rule files. The rest of the input items would be the same. You would use the same bilingual lexicon file and the same source text file. The outputs would remain the same.

Another limitation is that you cannot let any ambiguity remain after the analysis phase. You may wish to keep two or more ambiguous analyses after the analysis phase and let another process decide which analysis is appropriate, but this is not possible at this time. Things that can help in this regard are using special transfer rules that decide what translation is needed given a particular context. Custom bilingual entries as explained in Section 5.9.1.1 may also help.

Recursive syntax parsing is not possible in this MT system. Some parsing systems allow recursive application of phrase rules. **Apertium** is limited in this aspect. What a person typically does is use a "flat" system where all different kinds of phrase situations are handled at one level. It is possible to run the transfer process any number of times and in this way you could successively join phrases together, but the number of iterations is finite.

This MT system currently does not support infixes or other complex affixation. Only prefixes and suffixes are handled. Also subsenses of top level senses are not handled. All linking of word-senses have to be done at the top sense level. Text words that are analyzed to a subsense will not be found in the bilingual lexicon and will be therefore treated as unknown words. In the future the system could be extended to handle these situations.

Currently this MT system does not identify punctuation in the source text. Punctuation is left where it is and basically ignored. The system could easily be changed to recognize punctuation and it could prove useful in transfer rules to be able to identify where punctuation is located.

This MT system currently only runs in a Microsoft Windows environment.

5.8 Transfer Rules File Overview

This is an overview of the transfer rules file. I only describe the aspects of the rule file that apply to **Apertium**'s shallow-transfer system. See **Apertium**'s documentation in Forcada et al. 2010 for details about the syntax of rules for the advanced-transfer system which is for phrase-level changes and beyond the scope of this thesis.

The rule file is divided into two main parts: a declaration section and a rules section. The rules section uses information from the declaration section. In fact, every kind of reference in the rules section has to be declared in the declaration section.

5.8.1 Declaration

The declaration section has subsections for defining lexical categories, attributes, global variables, lists of strings and macro instructions. The first two of these are the most important. Lexical categories are used to group words together. The grouping can be very broad such as verbs or very specific such as pronouns with feminine suffixes. The categories are used in the pattern matching system discussed below. (20) is an example of a lexical category for indefinite nominals.

(20)
```
<def-cat n="nom_ind">
  <cat-item tags="n.ind"/>
  <cat-item tags="n.ind.*"/>
  <cat-item tags="n-irreg.ind" />
  <cat-item tags="n-irreg.ind.*" />
</def-cat>
```

The period indicates where a new tag begins. The asterisk is a wildcard indicator. It means that anything can fill that position. An asterisk at the end of the item matches one or more final tags. For example, if I have the lexical unit `^book<n><f><sg>$`, it could be precisely represented by the item `n.f.sg`. More generally we could match book with the item `n.f.*`. This would refer to all words that have the tags `<n>` followed by `<f>` followed by anything else, i.e. feminine nouns. (20) defines the set of all indefinite nouns — words that have the tag `<n>` or `<n-irreg>` followed by `<ind>` and optionally something else afterward.

Attributes are defined in the rule file in order to identify possible values for word characteristics. An example is shown in (21).

(21)
```
<def-attr n="nbr">
  <attr-item tags="sg"/>
  <attr-item tags="pl"/>
</def-attr>
```

Certain words may have affixes that indicate grammatical number. (21) defines the possible values for the number attribute as being `sg` (singular) or `pl` (plural). Like lexical categories, these refer to tags in the data stream which

would have the form <sg> or <pl> in this case. Attributes are used within rules to get or set values. See (23) for an example. I have described the declarations part of the rules file. Let us look at the rules section now.

5.8.2 Rules

As you can see in the first line of (5), repeated here as (22), there is an XML element that marks the beginning of a section of rules. Under this element one or more rules can be defined. Within a rule there are two main parts: the pattern definition and the actions to be carried out. The pattern section is what determines if the rule will be executed or not. If the word or words match the pattern the rule will be executed. The thing that goes in the pattern section is a list of one or more categories that are to be matched. In (22), the pattern will match a word that is a noun or adjective followed by a word that is a postposition. Exactly what is meant by the categories `nom-adj` and `post` will be found in the declaration section of the rule file where they are defined.

(22)

```
<section-rules>
  <rule comment="Handle noun or adjective plus the postposition rā.">
    <pattern>
      <pattern-item n="nom-adj"/>
      <pattern-item n="post"/>
    </pattern>
    <action>
      <out>
        <lu>
          <clip pos="1" side="tl" part="whole"/>
          <lit-tag v="acc/dat"/>
        </lu>
      </out>
    </action>
  </rule>
```

The action section contains the steps that are to be performed in the rule, generally ending with a section where lexical units are outputted into the data stream. In (22) the only thing being done is the outputting of data into the data stream. Looking at this example more carefully we see it is using the `clip` element to identify a piece of data. The `pos` attribute of `clip` identifies a particular lexical unit of the possible lexical units specified in the pattern above it. In (22) there are two possible lexical units. The `side` attribute can either be `sl` for source language or `tl` for target language. This means that the lexical unit or portion of the lexical unit being referred to can either be from the source word or the target word. The transfer engine knows what the target word is because it retrieves it from the bilingual lexicon. Lastly, the `part` attribute of the `clip` element identifies a particular part of the lexical unit. In (22), `whole` is used which refers to the lemma and all of the tags that follow it. Another possible value for `part` is `lem`, which would identify just the lemma part of the lexical unit (without tags). An attribute as defined in the declaration section of the rules file can also serve as a value for `part`. If in (22) we wanted to output just the number affix, we would set

`part` equal to `nbr`, and either `<sg>`, `<pl>` or nothing would be outputted depending on what the lexical unit looked like.

With an understanding of these basic mechanics of the rule file I will now outline the descriptive power of the transfer rules language. Conditional logic is a powerful construct that can be used in the rules. For example, you could have a rule where you say, “If the target word category is a postposition, remove the preposition and append the appropriate postposition.” The logical operators “and” and “or” can also be used in rules. For example, “If the target word category is a postposition and the noun is plural or the noun is dual, output ...” Negation is an available construct where you might say, “If the person attribute is not first singular, then output ...” Nesting of conditionals is also allowed. You can do string matching as well. For example, checking if a string ends with certain characters or starts with certain characters. Assignment of values can be done either to a portion of a lexical unit or to a variable. If a variable is assigned, it is a global variable. (23) shows the use of a conditional construct, an “or” logical operator and assignment of a value to a variable. In addition to these things, there is a mechanism for working with lists of string values. The string values are defined in the declaration section and are used in the rule section. For example, you can check to see if a certain value matches a string that is in the list.

A very helpful construct in the rules file is the macro construct. This gives you a way to modularize your rule instructions. If there is a set of instructions done multiple times across several rules, those instructions can be put into a macro to be reused. The macro is defined in the declaration part of the rule file. An example definition is shown in (23).

(23)
```
<def-macro n="2_3_pl_macro" npar="1">
  <choose>
    <when>
      <test>
        <or>
          <equal>
            <clip pos="1" side="tl" part="per-nbr"/>
            <lit-tag v="3pl"/>
          </equal>
          <equal>
            <clip pos="1" side="tl" part="per-nbr"/>
            <lit-tag v="2pl"/>
          </equal>
        </or>
      </test>
      <let>
        <var n="per_number"/>
        <lit-tag v="2/3pl"/>
      </let>
    </when>
    <otherwise>
      <let>
        <var n="per_number"/>
        <clip pos="1" side="tl" part="per-nbr"/>
      </let>
    </otherwise>
  </choose>
</def-macro>
```

To “call” the macro in a rule, you put special XML elements inside the action part of the rule. See (24). Note that by using the `with-param` element, you indicate which lexical units from the pattern are passed to the macro as parameters.

(24)
```
<call-macro n="2_3_pl_macro">
    <with-param pos="1"/>
</call-macro>
```

The benefit of macros is that the instructions can be maintained in just one place in the rule file which is much better than trying to maintain duplicate instructions elsewhere. The concept is analogous to using a function in a computer programming language.

5.9 Modules

There are three groupings of modules in this MT system: extraction modules, the transfer module and synthesis modules. They are described below.

5.9.1 Extraction Modules

The extraction modules have to do with extracting data from the **FLEx** projects for input to the transfer module.

5.9.1.1 Extract Source Text Module

The Extract Source Text module will find the source **FLEx** text as specified in the configuration file and extract each parsed word into a text file in **Apertium** format. **Apertium** is designed to work with a segmented data stream and uses two types of data streams as it goes through its processes. One is called the *ambiguous segmented stream* which it uses in its analyzer and tagger modules; the other is called the *unambiguous segmented stream* which it uses in its tagger, transfer and generator modules (Forcada et al. 2010). The difference, as indicated by the name, is that the *ambiguous segmented stream* can have multiple lemmas for each lexical unit. The *unambiguous segmented stream* has only one – no ambiguity. It is the *unambiguous segmented stream* format that I write to since I am only using **Apertium**'s transfer module and all ambiguity has been removed in the analysis process.

The data stream at a basic level is made up of lexical units that are delimited by `^` and `$`. (25) shows a stream of three words.

(25) `^He$ ^goes$ ^home$`

Each lexical unit is actually more complex and consists of a lemma and one or more grammatical tags or labels that describe the morphological, inflectional and grammatical attributes of the surface form. (26) shows a possible data stream for three English words.

(26) `^he<n><m><sg>$ ^go<v><3sg>$ ^home<n><sg>$`

In this module, for each word I extract the lexical form (headword in **FLEx** terminology), the grammatical category, zero or more inflectional features or inflectional classes assigned to the lexical entry, prefix glosses and suffix glosses.[5] This is depicted in (27).

(27) `^headword<gramm_cat><feat1>...<featN><pfx1>...<pfxN><sfx1>...<sfxN>$`

A snippet of a data stream in Persian is shown in (28) and (29). (29) shows the stream in Arabic script.[6] The `1.1` after the word indicates the first homonym and the first sense.

(28) `^ruz1.1<n><ind>$ ^budan1.1<cop><pst><3sg_pst>$`

(29) `^روز1.1<n><ind>$ ^بودن1.1<cop><pst><3sg_pst>$`

[5] In this MT system, attached clitics are treated as affixes.

[6] Because all the tools in this MT system use the Unicode UTF-8 standard, Arabic script works seamlessly throughout the system.

In (28) and (29) `n` and `cop` represent the grammatical categories `noun` and `copular`. The tag `pst` is an inflectional feature indicating past tense; `ind` and `3sg_pst`[7] are the glosses of suffixes.

This module implements the design feature mentioned in Section 5.6.7 (Variant Entries). Variant forms in the source text are always converted to the main forms to which they are connected. The way this is done is to recursively follow the connections from variant form to connected entry until a main entry is found that has sense information. When this entry is found, the information such as the grammatical category and inflectional information is extracted. (30) shows an entry that is a variant entry. (31) shows the main entry to which the variant is connected.

(30) **colour color$_2$**

Lexeme Form	Eng **colour**
Morph Type	stem
Citation Form	Eng
Variant Type	Spelling Variant
Variant of	color$_2$

(31) **color$_2$** (colour) *n* Farbe

Lexeme Form	Eng **color**
Morph Type	stem
Citation Form	Eng
⊟Sense 1	
Gloss	Ger **Farbe**
Definition	Ger
Grammatical Info.	Noun
⊟Variants	
Variant Form	Eng **colour**
Variant Type	Spelling Variant

If this module found the word "colours" in the source text which would have an analysis such as (32), it would output the lexical unit shown in (33). Observe that the main entry lemma is present, not the variant entry lemma.

(32)

Word	colours	
Morphemes	colour	-s
Lex. Entries	color$_2$+sp. var.	-s$_1$
Lex. Gloss	Farbe	pl
Lex. Gram. Info.	n	n:(number)

(33)

```
^color2.1<n><pl>$
```

This module also implements the design feature mentioned in Section 5.6.2 (Complex Entries). This module composes one or more words that make up certain complex entries into a lexical unit based on the complex entry from **FLEx**. In a **FLEx** text, you have the option of making one or more words in a text into a phrase and the phrase can

[7] Periods are converted to underscores in affix and clitic glosses. This is because the period is used in Apertium when referencing multiple tags. For example, `n.ind` to indicate `<n>` followed by `<ind>`.

be analyzed as a whole. This is great when you are dealing with idioms or expressions where there are not normally multiple morphemes per word. When you have the situation of multiple morphemes on a word in the phrase, though, this does not work very well in **FLEx**. You have to analyze each word separately. Persian phrasal verbs are a good illustration of this. Take for example, the verb kār kardan, 'to work'. The lexical entry for the verb is shown in (34).

(34)

Lexeme Form	Far-rom **kār kardan**
Morph Type	phrase
Citation Form	Far-rom
Complex Form Type	Phrasal Verb
Components	kār$_1$ kardan
⊟Sense 1	
Gloss	Eng **work.pst**
Grammatical Info.	Verb Phrasal
Gilaki Sense Number	1
Gilaki Equivalent	silfw://localhost/link?app%3dflex%26database%3dGilaki%26server%3d%26tool%3dlexiconEdit%26guid%3d14e61a5d-9bc3-457e-8f4b-4d5ce7ac528f%26tag%3d

Notice the components kār$_1$ and kardan that make up this complex form. Each of these components have their own entries in the lexicon. See examples (35) and (36).

(35)

Lexeme Form	Far-rom **kār**
Morph Type	stem
Citation Form	Far-rom
Complex Forms	kār kardan
⊟Sense 1	
Gloss	Eng **work**
Grammatical Info.	Noun
Gilaki Sense Number	
Gilaki Equivalent	

(36)

Lexeme Form	Far-rom **kard**
Morph Type	stem
Citation Form	Far-rom kardan
Complex Forms	kār kardan
⊟Sense 1	
Gloss	Eng **do.pst**
Grammatical Info.	V Compound pst
Gilaki Sense Number	
Gilaki Equivalent	

Also note that in (34) we have a link to the target entry in the custom field named ***Gilaki Equivalent***. If we have the phrasal verb kār mikardam, 'I was working' in a text, we have to analyze it as two words as shown in (37).

(37)

Word	kār	mikardam		
Morphemes	kār	mi-	kard	-am
Lex. Entries	kār₁	mi-	kard	-am₁
Lex. Gloss	work	ipfv	do.pst	1sg
Lex. Gram. Info.	n	v:(Asp)	vcomp pst	v:SuAgr+

As it extracts the text data from the **FLEx** project, this module detects that kār and kardan are constituents of the complex entry shown in (34). It also determines that the complex entry type which in this case is ***Phrasal Verb*** is in the list of source complex types to process. The list is defined in the configuration file (cf. Appendix B). One lexical unit for the phrasal verb is outputted by drawing on the information from the complex entry shown in (34). The output would look like (38).

(38) `^kār kardan1.1<vphrase><ipfv><1sg>$`

This module also implements part of the design feature mentioned in Section 5.6.5 (Unknown Words). If a word in the source text has no analysis, it is treated as an unknown word. Such a word is assigned the grammatical category `unk`. The lexical unit in the input data stream for the transfer process will be in the form `word<unk>`. Coming out of the transfer process an unknown form will have the form `@word<unk>`. The @ sign indicates that the word was not found in the bilingual dictionary. As explained in Section 5.9.3.4, the user can clean up the target text and have words with the @ sign removed. This is helpful when you want proper names or other similar words to remain as is in the target text.

5.9.1.2 Extract Bilingual Lexicon Module

The Extract Bilingual Lexicon module extracts information from both the source and target lexicons and creates a bilingual lexicon. This lexicon is a mapping of each sense of each lexical entry from the source lexicon to a corresponding sense in the target lexicon. The mechanism that ties source and target senses together are the linking fields described in Section 4.4.3.

The linking of senses applies only to root entries. In other words, linking does not apply to affixes or clitics.[8] As mentioned in Section 5.6.1, the MT system assumes that if there is no link present from a source word-sense to a target word-sense, the lexeme and sense in the source lexicon is the same as the target. This is very convenient when there is a high degree of shared vocabulary between the source and target languages. Essentially the only linking that needs to be done is for those senses in the source language that have different equivalents in the target language. Of course there are situations where the equivalent sense in the target language is unknown. These senses can remain without a link and by default the MT system will generate the same form in the target language as in the source language.

[8]The way this module determines which kind of entries to check for linking is by examining the “Morph Type” field. There is a configuration property called “SourceMorphNamesCountedAsRoots” that enumerates the morph types. See Appendix B for details.

The format of the output is **Apertium**'s XML format for lexicons. For the source word-sense, the headword plus a sense number and the grammatical category is outputted. For the target word-sense, the same thing is outputted with the addition of inflectional information. A sample line from the Persian-Gilaki bilingual lexicon is shown in (39).

(39)
```
<e><p><l>neveštan1.1<s n="v"/></l><r>nivištēn1.1<s n="vpst"/><s n="pst"/></r></p></e>
```

The XML elements have the following meanings shown in (40) (cf. Forcada et al. 2010).

(40)

e	entry
p	string pair
l	left element
r	right element
s	symbol
i	identity (used in (41) below)

As in (28) the `1.1` in (39) means the first homograph and the first sense. The category of the Persian word is `v`, the category of the Gilaki word is `vpst` and an inflection feature of the Gilaki word is `pst`.

A line where the source and target senses are the same is shown in (41). This is what gets outputted when there is no link made.

(41)
```
<e><i>ba'd1.1<s n="adv"/></i></e>
```

The beginning of the bilingual lexicon file has a section for symbol definitions. Symbol is another word for tag in **Apertium** terminology. All the symbols in the lexicon such as `adv` in (41) need to be defined. This module extracts a list of all grammatical categories from the source and target **FLEx** projects, merges them into one symbol list and inserts them into the file. A sample symbol definition line is shown in (42).

(42)
```
<sdef n="adv" c="Adverb"/>
```

This module implements the design feature mentioned in Section 5.6.3 (Grammatical Category Mapping). It may be the case in some language pairs that grammatical categories have different names for the same categories. In such a case you might want to do a wholesale mapping of one grammatical category to another. This feature is implemented via the configuration file. There is a configuration property named `CategoryAbbrevSubstitutionList` that you set to pairs of grammatical categories (cf. Appendix B). For example, if the value of this property is `n-irreg,n`, all source word-senses in the bilingual lexicon that have the category `n-irreg` and which have no current mapping

to a target word-sense will get mapped to a lemma that is the same as the source, except with the category `n`. The way it would look with actual lines in the lexicon is shown below. This module would change (43) to (44).

(43) `<e><i>jazire1.1<s n="n-irreg"/></i></e>`

(44) `<e><p><l>jazire1.1<s n="n-irreg"/></l><r>jazire1.1<s n="n"/></r></p></e>`

Just as in (41), (43) has no explicit mapping of the source word-sense to a target word-sense. With no mapping, the source word-sense `jazire1.1` would remain `jazire1.1` after the transfer process and the grammatical category would remain `n-irreg`. With the MT system set to map `n-irreg` to `n`, this module creates the line shown in (44) where `jazire1.1` is mapped to the identical `jazire1.1` except that the grammatical category is changed to `n` as shown in the element `<s n="n"/>`.

This module also implements the design feature mentioned in Section 5.6.4 (Bilingual Lexicon Customization). It allows you to make custom changes to the bilingual lexicon. As an example of why you might want to do this, let us look at (45) which shows a custom change used in the Persian-Gilaki test case.

(45) `<e><p><l>haman1.1<s n="dem"/></l><r>u1.1<s n="dem"/><s n="same"/><s n="emph"/></r></p></e>`

Here, a word-sense in the source language is mapped to a word-sense in the target language plus two affixes. Normally a bilingual lexicon entry does not have affixes in it, but this customization feature allows such an entry to be created. The same thing could be accomplished by using a transfer rule, but since the change is specific to just one word, it makes more sense to use a custom bilingual lexicon entry.

The customization lines are put into a text file whose path and name are set in the configuration file (cf. Appendix B). The file has three sections as shown in (46).

(46)
a. `<!-- Add definition lines here -->`
b. `<!-- Add replacement lines here -->`
c. `<!-- Add lines to be appended here. (Do not delete this line.) -->`

Lines in the bilingual lexicon that you want to replace go under the text shown in (46b). The example above shown in (45) is an example of a replacement line. This module will add the replacement line to the bilingual lexicon and will comment out the original line. There is no ambiguity about which line this module will comment out because each line begins with a unique source lemma.

Lines in the bilingual lexicon that you want to add go under the text shown in (46c). This module will append any lines listed in this section. An example of a new bilingual lexicon entry that would go in this section is shown in (47).

(47)
```
<e><p><l>hame1.1<s n="quant"/><s n="pc_3sg"/></l><r>haməš1.1<s n="n"/></r></p></e>
```

This line maps a source word-sense with an attached clitic to a target word-sense.

The first section of the file will have definition lines that may be needed for undefined symbols. For the line shown in (47), you would need to add a definition line as shown in (48).

(48)
```
<sdef n="pc_3sg" c="Pron. Clitic 3sg"/>
```

5.9.2 Transfer Module

The transfer module is the **Apertium** transfer engine. Its function is explained in Section 5.3.

5.9.3 Synthesis Modules

The synthesis modules have to do with taking the transferred data from the transfer module and synthesizing it into surface forms.

5.9.3.1 Catalog Target Prefixes Module

The Catalog Target Prefixes module creates a catalog of all the prefixes in the target lexicon. This is necessary because in the **Apertium** data stream, it is unknown which symbols are prefixes, suffixes or inflection information. Other modules consult the catalog to see if a given affix is a prefix. The result of this module is a prefix catalog file named according to the applicable configuration file setting (cf. Appendix B).

5.9.3.2 Convert Text to STAMP Format Module

The Convert Text to **STAMP** module converts the output file from the transfer module to a format that **STAMP** expects. The critical thing that the module does is convert each lexical unit to a different format. Additionally this module handles paragraph breaks and punctuation in the data stream. The format is a text database where each record represents a word and has one or more fields giving information about the word. See Weber et al. 1988. The standard lists many possible fields, but for this MT system I only need the fields `\a` - the analysis breakdown of the word, `\f` - preceding format marks and `\n` - trailing nonalphabetic characters. The data in the `\a` field is simply one or more prefixes followed by the grammatical category, the headword and one or more suffixes. The category and headword are delimited with angle brackets. A sample record is shown in (49a)-(c).

(49) a. \a pfv < vpst daštən1.1 > 1sg
b. \f \n
c. \n ,

This module implements the design feature mentioned in Section 5.6.2 (Complex Entries). Just as the extract source text module (Section 5.9.1.1) combines words into complex entries, this module decomposes complex phrases into their constituent parts.

Let us take an example from Gilaki. After the transfer process, let us suppose that we get the lexical unit shown in (50).

(50) ^ranj kəšen1.1<vcmplx><1sg>$

This module detects that ranj kəšen matches a complex form in the target lexicon. This is shown in (51). It also determines that its complex form type ***Phrasal Verb*** is in the list of types it is to decompose. It proceeds to decompose the phrase.

(51) **ranj kəšen** (ph. v. of **ranj**, **kəšen**) *vcmplx* suffer.pst

Lexeme Form	Gil-lat **ranj kəšen**
Morph Type	phrase
Complex Form Type	Phrasal Verb
Components	ranj kəšen

⊟Sense 1

Gloss	Eng **suffer.pst**
Grammatical Info.	Complex Verb
Semantic Domains	

The module checks the components of the complex entry which are ranj and kəšen. Their entries in the target lexicon are shown in (52) and (53).

(52) **ranj** *n* suffering ph. v. **ranj kəšen**

Lexeme Form	Gil-lat **ranj**
Morph Type	stem
Complex Forms	ranj kəšen

⊟Sense 1

Gloss	Eng **suffering**
Grammatical Info.	Noun

(53) **kəšen** *vpst* pull.pst ph. v. **ranj kəšen**

Lexeme Form	Gil-lat **kəše**
Morph Type	stem
Citation Form	Gil-lat kəšen
Complex Forms	ranj kəšen
⊟ **Sense 1**	
Gloss	Eng **pull.pst**
Grammatical Info.	VerbPast pst

This modules extracts grammatical category information from these entries and two corresponding words are outputted in the file. These are shown in (54) and (55).

(54) `\a < n ranj1.1 >`

(55) `\a < vpst kəšen1.1 > 1sg`

5.9.3.3 Extract Target Lexicon Module

The Extract Target Lexicon module extracts the lexicon from the target **FLEx** project. It is extracted into three lexicon files: one for roots, one for prefixes and one for suffixes. Each of these lexicon files is in a text database format that **STAMP** can read. Similar to the text format described in Section 5.9.3.2, each record represents a sense and has fields that pertain to that sense. For this MT system I use the fields shown in (56) for roots and for affixes I use the fields in (57).

(56)

Field	**Meaning**
\m	headword
\c	grammatical category
\g	gloss [9]
\a	allomorph

(57)

Field	**Meaning**
\g	gloss
\a	allomorph

The allomorph field may be repeated multiple times for all the possible allomorphs of this entry. See Section 5.9.3.4 for more information about how allomorphs are processed in the synthesis module. See Section 4.4.1 for information

[9] The gloss field is not necessary for **STAMP** but is helpful when viewing the root lexicon.

about how allomorphs are set up in **FLEx**. A sample root record is shown in (58a)-(d) and an affix record is shown in (59a)-(c).

(58)
a. `\m darbār1.1`
b. `\c n`
c. `\g court`
d. `\a darbār1.1`

(59)
a. `\g 3sg`
b. `\a yə / [VMinusYe] _`
c. `\a yə ~/ [VMinusYe] _ / [Ye] _`

A very important function this module performs is organizing the environment constraints. One difference between **FLEx**'s interpretation of allomorphs and **STAMP**'s is that **FLEx**, as mentioned in Section 4.4.1, assumes a negation of all the previous allomorphs' environment constraints as part of the current allomorph's environment constraint (Black, 2014). **STAMP** does not make this assumption. This means that in order to get the same choice of allomorph as in **FLEx**, **STAMP** has to add the negation of the previous allomorphs' environment constraints as part of the current allomorph's constraint. (60) shows the **FLEx** allomorph listing of the 3sg suffix in Gilaki. (59b) and (59c) show how they look in the **STAMP** suffix lexicon. Note that (59c) has the negated environment of (59b) included in its environment constraint (~ is the indicator of negation).

(60)
⊟ **Allomorphs**

Affix Allomorph	Gil یه
Environments	/ [VMinusYe] _
Affix Allomorph	Gil یه
Environments	/ [Ye] _

Another function of this module is the creating of other input files that are used by **STAMP**. One of these is a catalog of all the grammatical categories used in the root lexicon. This module extracts grammatical categories from the target **FLEx** project and creates the needed **STAMP** input file. Also, this module extracts natural class information from **FLEx** and creates an input file that **STAMP** uses. A natural class is a label given to set of phonemes. Natural classes are used in environment constraints as a shortcut to refer to multiple phonemes at one time. In (59b) `VMinusYe` is the natural class which in this case refers to all vowels except the "ye" vowel. Other input files are also created that **STAMP** needs. See Weber et al. 1990. As a last step, this module runs the **STAMP** program.

5.9.3.4 Synthesize Text Module

The Synthesize Text Module is basically the **STAMP** program. Its function is explained in Section 5.4.

This module implements part of the design feature mentioned in Section 5.6.5 (Unknown Words). When an unknown word goes through the system, it will normally be outputted in the target text with an @ sign in front of it, for example, `@word`. If the MT system is configured to clean up the target text, the @ sign will be removed. This is helpful when you want proper names or other similar words to remain as they appear in the target text. In this configuration, this module also will clean up words that exit the transfer process in the form `word1.1` — the `1.1` is removed.

5.9.3.5 Insert Target Text Module

The Insert Target Text module is very straightforward. It takes the text produced by STAMP and inserts it as a text in the target **FLEx** project. This module uses the title of the text from the source **FLEx** project as the new text title in the target project. If a text exists in the target project with that title, it appends text to the title to make it unique.

6

Experiments

In this chapter I look at a test case of using the MT system to translate some texts from Persian to Gilaki. I give an overview of the language differences and describe the initial setup of the system to get texts translating from Persian to Gilaki. I then report the results from an experiment of taking a new short narrative text and putting it through the system. I document the adjustments made to the system to improve it along with the BLEU scores at each stage. I conclude with a list of phenomena that remained unhandled.

6.1 A Test Case with Persian and Gilaki

The test case languages for this MT system were Persian and Gilaki. Persian served as the source language and Gilaki as the target language.

6.1.1 Overview of the Language Differences

Persian and Gilaki are both Indo-European languages. They both share the classification of belonging to the Western branch of the Iranian, Indo-Iranian family, but Persian is a Southwestern language and Gilaki is Northwestern. Persian is a major world language spoken by 57 million people worldwide (Lewis et al. 2015). Gilaki is spoken by around three million people (Lewis et al. 2015). The two languages share a great deal of vocabulary. Some of the bigger differences are as follows: Persian has a noun phrase order where the adjective follows the noun. Gilaki commonly has the adjective in front of the noun, but also follows the Persian pattern in some cases. Persian uses prepositions; Gilaki uses both prepositions and postpositions. Persian could be said to have nominative and accusative case while Gilaki has additionally the genitive and dative cases. Persian has just one set of pronouns while Gilaki has pronouns for the nominative, accusative/dative and genitive cases. Gilaki has a rich set of verbs with a pre-verbal element; Persian does not have this. Gilaki has significant differences in verb morphology from Persian. For example, it has a form-building prefix that gets added to past verbal stems. Persian lacks this. Gilaki uses a suffix to mark the past continuous

tense while Persian uses a prefix. Persian has separate verb endings for second and third person plural; Gilaki uses the same form for both. See Thackston, 1993 and Rastorgueva et al. 2012. Persian and Gilaki serve as good test case languages for developing an MT system. They are similar enough to share a lot of vocabulary which makes it easier to create a bilingual lexicon, yet they are different enough that the MT system is challenged to deal with significant morphological and syntactic differences.

6.1.2 Test Case Initial Setup

To set up this test case the things mentioned in Section 4.4 had to be in place. First of all I had lexicons for both languages. I made use of two **FLEx** projects that I and colleagues developed over many years; one **FLEx** project for Persian and one for Gilaki. Both of these projects have lexicons in them. The Persian lexicon has about 11,000 entries and the Gilaki lexicon has about 3,000 entries. These lexicons contain affixes and clitics in addition to stems.

Secondly I had in place grammar settings in each **FLEx** project which allow me to efficiently perform morphological parsing of texts in each language. Besides the lexicon and grammar settings, I created the other input items to the system incrementally. I created links between Persian word-senses and Gilaki word-senses, wrote transfer rules, made a list of Persian grammatical categories that needed to be mapped wholesale to Gilaki categories and created custom bilingual lexicon entries. All these things plus a source text comprised the inputs of the Persian-Gilaki test case to this MT system.

6.1.3 Scope of the Test Case

I intentionally limited the scope of this test case. I decided not to handle certain kinds of phenomena when translating from Persian to Gilaki. The main limitation I set was not handling large scale constituent reordering in the syntax. For example, I do not handle P NP to NP P changes where the NP is more than one word. Likewise I do not handle NP Adj to Adj NP changes where the NP is more than one word. The main focus of the test case was word-level transfer with the addition of some word reordering.

6.2 Initial Implementation

A short parallel text for this test case was used to develop the MT system. For my text, I selected five verses of chapter ten of the book of Acts from the Bible. It is comprised of a six-sentence narrative text with some direct speech. The Persian text was entered into the Persian **FLEx** project and analyzed into morphemes. For each sense in the Persian text that needed it, I linked it to the appropriate sense in the Gilaki lexicon following the procedures in Section 4.4.3.

The next step was to write transfer rules to handle the morphology changes and some simple syntactic changes. The description of the rules are shown in Table 1. In addition I created one category mapping rule and one custom entry for the bilingual lexicon.

A category substitution rule was necessary because the Persian lexicon uses two basic noun categories: a standard noun and a noun that has an irregular plural form. In Gilaki, there is only one basic noun category. The category mapping rule that I created substitutes the irregular plural noun in Persian with the standard noun in Gilaki (cf Section 5.9.1.2). The custom entry in the bilingual lexicon was needed because I had the need to convert a fused word in Persian to a Gilaki word with certain suffixes.

I did some additional development and testing of the system using a list of verb paradigms in Persian. This exercised the system in translating verbs in all different tenses, aspects and moods.

After getting two texts translating reasonably well into Gilaki, I had 12 transfer rules (see Table 1) and 108 Persian word-senses linked with Gilaki word-senses. The BLEU score I got when comparing the generated Bible text in Gilaki to the reference text was 25.0.

Num.	**Rule**	**Description**
1	noun-adjective reversal	For a limited set of adjectives that follow a noun, reverse the order to adjective noun.
2	accusative postposition	Change the Persian accusative postposition to the Gilaki accusative enclitic.
3	preposition and pronoun	Change the prepositions be and baraye before a pronoun to the appropriate case enclitics. Also change applicable prepositions to postpositions.
4	imperfective verbs	Change the Persian imperfective prefix to the Gilaki imperfective suffix.[1]
5	infinitive verbs	Remove unneeded features.
6	past copular verbs	Remove unneeded features.
7	past verbs	Add the form-building prefix.
8	present participle verbs	Remove unneeded features.
9	imperative verbs	Change affix names.
10	present verbs	Remove the mi prefix if present.
11	present copular verbs	Change affix names.
12	3sg copula	Change it to an enclitic.

Table 1 Initial Transfer Rules

[1] In all the verb rules, the 2nd and 3rd plural endings in Persian get collapsed to one Gilaki ending.

6.3 Experiment Setup

This section describes an experiment of testing the MT system with a new text. The purpose of the experiment was to process an unseen text and see how well the MT system handled it. Also, I wanted to see what adjustments caused the best improvements in the system. The text I used is a Persian story titled, “The Hungry Mouse”, see Mouse, n.d. It is a short narrative text with 55 sentences. See Table 4. It has direct speech interspersed throughout the narrative. The first step for the experiment was entering the text into the Persian **FLEx** project, adding any missing vocabulary to the lexicon and analyzing the text into morphemes.

At each stage of improving the translation, I calculated a BLEU score using **iBLEU** (Madnani, 2011). To calculate a BLEU score you need one or more reference texts. For this, I had two native Gilaki speakers translate “The Hungry Mouse” story from Persian into Gilaki. I edited the two stories to regularize the spelling of the texts. This was necessary because lacking standardized rules for writing Gilaki, the authors spelled words differently not only from each other, but also from the system used in my Gilaki lexicon. The next section shows the results of running this new text through the MT system. Rule numbers that are referred to are the numbers listed in Table 1 and Table 3 (the new rule numbers are in Table 3 only).

6.4 Results

The BLEU score results for each of the nine stages is shown in Table 2.

Stage Description	**BLEU Score**	**% Improvement**
Previous Training	0.82	
Stage 1 - Lexical coverage improvements	23.65	
Stage 2 - Fixes to synthesis problems	26.24	11.0%
Stage 3 - Persian possessive pronominal enclitics	26.66	1.6%
Stage 4 - Persian adverb ham	29.78	11.7%
Stage 5 - Prepositions to postpositions	31.11	4.5%
Stage 6 - Indefinite nouns	32.91	5.8%
Stage 7 - Custom bilingual lexicon entries	33.31	1.2%
Stage 8 - Accusative pronouns	33.44	0.4%
Stage 9 - Possessive enclitics and prepositions	34.98	4.6%

Table 2 Experiment Results

6.4.1 Previous Training

The previous training score was calculated against the same exact system that had been used in the test case development. In other words, the same lexicons for Persian and Gilaki were used and the rules listed in Table 1 were used. It was expected that the BLEU score would be low. The setup only had links from Persian word-senses to Gilaki word-senses which covered the vocabulary used in the two initial training texts.

6.4.2 Stage 1 - Lexical Coverage Improvements

In stage 1 I added lexical entries for missing vocabulary and created the needed links between word-senses in the two lexicons. As expected the BLEU score goes up dramatically. This score serves as a baseline against which we can measure subsequent improvements. I added 62 lexical entries to the Gilaki lexicon and 90 word-sense links to the Persian lexicon. This figure is higher than the number of Gilaki entries added because links were sometimes added to multiple senses of an entry. Also links were sometimes added to handle a change of grammatical category.

6.4.3 Stage 2 - Fixes to Synthesis Problems

In stage 2 I fixed synthesis problems in the Gilaki lexicon. I needed to adjust the allomorph lists of about a dozen Gilaki entries. Sometimes I deleted an unneeded allomorph, sometimes I added an environment constraint to an unconstrained allomorph, sometimes I adjusted a phoneme environment constraint for an allomorph. (61) is an example of an entry that had an unconstrained allomorph. The allomorph `səv` is an allomorph in Gilaki that is only used with the ordinal suffix `-om`, but because it was listed as the first allomorph and had no constraints, **STAMP** always selected it as the stem to use for the synthesized form. The data stream coming into **STAMP** was `^sə1.1<num><clf>$`[2](the numeral stem `sə` with classifier suffix `clf`).[3] The lexeme form for `clf` is `ta` in the Gilaki lexicon. Given this entry and this data stream, **STAMP** outputted the word `səvta`, but the expected output is `səta`.

(61)
```
\m sə1.1
\c num
\g three
\a səv
\a sə
```

When I changed the entry to what is shown in (62),[4] **STAMP** produced the expected word. The first allomorph is now rejected because there is no `-om` suffix following the stem as specified in the environment constraint. The next

[2] The actual stream in Arabic script was: ^1.1سه<num><clf>$.
[3] Recall that suffixes and other affixes are identified by their gloss.
[4] The example is given in Latin script and simplified for illustrative purposes.

allomorph's environment constraints are checked and it passes the tests. ([A] is a phoneme natural class that refers to all phonemes.) With these environment constraints in place, **STAMP** selects the stem sə and the correct word səta is produced.

(62)
```
\m sə1.1
\c num
\g three
\a səv / _ om
\a sə ~/ _ om / _ [A]
```

When making these fixes, I changed the entry sə1.1 in the Gilaki **FLEx** project. Also, within **FLEx**, I tested word-affix combinations with the *Try a Word* utility.

6.4.4 Stage 3 - Persian Possessive Pronominal Enclitics

In stage 3 I handled Persian possessive pronominal enclitics. Gilaki rarely uses possessive pronominal enclitics. Instead, it uses a genitive pronoun of the appropriate person and number. The genitive pronoun in Gilaki precedes the thing it modifies. To convert Persian possessive pronominal enclitics to genitive pronouns, I wrote transfer rule 0.5.[5] The pattern for the rule is specified at the top of (64) as nom_pc. The string nom_pc is a category defined in another part of the rule file as is shown in (63).

(63)
```
<def-cat n="nom_pc">
  <cat-item tags="*.pc_1sg"/>
  <cat-item tags="*.pc_2sg"/>
  <cat-item tags="*.pc_3sg"/>
  <cat-item tags="*.pc_1pl"/>
  <cat-item tags="*.pc_2pl"/>
  <cat-item tags="*.pc_3pl"/>
</def-cat>
```

Any word that has one of these six enclitics at the end matches the pattern. In line seven of (64) you see a call to a macro which is partially shown in (65).

[5] The initial transfer rules were numbered sequentially (1-12). As new rules were added that needed to be in sequence before an existing rule, I used a decimal number that was less than the existing rule.

(64)

```
<rule comment="0.5 - convert possessive pronominal enclitics to genitive pronouns">
  <pattern>
    <pattern-item n="nom_pc"/>
  </pattern>
  <action>
    <!-- Set the gen_pro variable to the appropriate Gilaki genitive pronoun. -->
    <call-macro n="gen_pro_macro">
        <with-param pos="1"/>
    </call-macro>
    <out>
      <lu>
        <var n="gen_pro"/>
        <lit-tag v="perspro"/>
      </lu>
      <b/>
      <lu>
        <clip pos="1" side="tl" part="lem"/>
        <clip pos="1" side="tl" part="tl_pos"/>
        <clip pos="1" side="tl" part="nbr"/>   <!-- if it exists -->
      </lu>
    </out>
  </action>
</rule>
```

(65)

```
<def-macro n="gen_pro_macro" npar="1">
  <let> <!-- initialize the variable to null -->
    <var n="gen_pro"/>
    <lit v=""/>
  </let>
  <choose>
    <when> <!-- 1sg -->
      <test>
        <equal>
          <clip pos="1" side="sl" part="pron-clitic"/>
          <lit-tag v="pc_1sg"/>
        </equal>
      </test>
      <let>
        <var n="gen_pro"/>
        <lit v="2.1می"/>
      </let>
    </when>
    <when> <!-- 2sg -->
      <test>
```

In (65) you can see that the `pron-clitic` suffix is examined and if it matches the value `pc_1sg`, the `gen_pro` variable is assigned to the genitive pronoun lemma string. The rest of the macro does similar tests to match the other five person-number enclitics. Going back to (64), you see that the `gen_pro` variable is outputted along with its grammatical category (`perspro`), then a blank, symbolized by `</b>`, is outputted and the components of the modified word are outputted.

Aa an illustration of data going through this process, let us take the Persian word saram meaning ‘my head’ which should be translated to mi sər in Gilaki. The data stream coming into the transfer engine is `^sar1.1<n><pc_1sg>$`. The pattern in (63) is matched because the word ends with `<pc_1sg>`. The macro part of the rule in (64) as shown in (65) is executed, `pc_1sg` is found and thus the variable `gen_pro` is set to the Gilaki lemma `2.1` می which in

Latin script is mi2.1. Lastly, the following things are outputted: the value of the gen_pro variable, the category perspro, a blank, the translation of the Persian lemma sar1.1 which yields the Gilaki sər1.1 from the bilingual lexicon, the target language part of speech (also from the bilingual lexicon) and the number tag which is empty in this case. The resulting data stream from the transfer engine is ^mi2.1<perspro>$ ^sər1.1<n>$. The synthesis process turns this into mi sər as desired.

In addition to rule 0.5 shown in (64), I modified rule 2 to handle possible possessive pronominal enclitics that might be present on the noun. With this modification, for example, the Persian češmhāyat rā gets translated to ti češmana.

6.4.5 Stage 4 - Persian Adverb ham

In stage 4 I handled the Persian adverb ham. The adverb ham, meaning 'also', translates to the Gilaki enclitic =am. To do this conversion, I wrote two new transfer rules: 2.5 and 2.6. Rule 2.5 does the conversion when we have a noun, adverb, or pronoun followed by the word ham. Rule 2.6 does the conversion when we have a noun, adverb or pronoun, the postposition ra and the word ham. Let us look at the first rule which is shown in (66).

(66)
```
<rule comment="2.5 - convert noun/adv/pro plus the adverb 'ham' to an enclitic.">
  <pattern>
    <pattern-item n="nom-adv-pro"/>
    <pattern-item n="adv-ham"/>
  </pattern>
  <action>
    <out>
      <lu>
        <clip pos="1" side="tl" part="whole"/>
        <lit-tag v="also"/>
      </lu>
    </out>
  </action>
</rule>
```

The rule is straightforward. The pattern is looking to match a noun, adverb or pronoun (this is what the symbol category nom-adv-pro encapsulates) plus ham. If the pattern is matched, the rule's action is executed and the first word in its entirety, after being translated to Gilaki, is outputted as well as the tag <also> (also is the gloss for the Gilaki enclitic =am). If we had the Persian phrase barghā ham meaning 'the leaves also', the input to the transfer engine would be ^barg1.1<n><pl>$ ^ham2.1<adv>$. After rule application, the output becomes ^barg1.1<n><pl><also>$. The synthesis process then produces barganam where -an is the plural suffix and =am is the enclitic meaning 'also'.

The second rule for handling ham (rule 2.6) is similar. There is just some extra processing for the postposition ra which is covered in Section 6.4.9.

Note that these two rules are limited to nouns, adverbs and pronouns with the possible ra postposition present. There are potentially other grammatical categories that need to be handled as well.

6.4.6 Stage 5 - Prepositions to Postpositions

In stage 5 I converted Persian prepositions to the appropriate Gilaki postpositions. I added rule 3.5 to handle this. Prepositional phrases can be of any length, but minimally they are made up of a preposition and a noun. The new rule only handles this minimal situation. The pattern for this rule is a preposition followed by a noun. The rule has this basic logic: If the Persian preposition coming in maps to a Gilaki postposition in the bilingual lexicon, then the noun followed by the postposition should be outputted. The test part of this logic is shown in (67).

(67)
```
<test>
  <equal>
    <clip pos="1" side="tl" part="tl_pos"/>
    <lit-tag v="post"/>
  </equal>
</test>
```

Not all Persian prepositions map to Gilaki postpositions, but the ones that do will succeed on this test. The ones that do not pass will have the same two words outputted again in their Gilaki forms. If the test is successful, there is a further test. With the use of postpositions in Gilaki, the preceding noun sometimes has a genitive enclitic attached. The enclitic is present if the noun ends in a consonant but is not present if the noun ends in a vowel. The test to see if the noun ends in a vowel is shown in (68).

(68)
```
<test>
  <ends-with-list>
    <clip pos="2" side="tl" part="lem"/>
    <list v="end_vowels"/>
  </ends-with-list>
</test>
```

The list referred to is a list of vowels which is defined at the top of the rule file. If the test passes, the noun and postposition are outputted; if not, the `<gen>` tag is appended to the end of the noun. Take, for example, the Persian phrase dar sahrā meaning ‘in the field’. The Gilaki translation of this is sahrā durun. The input to the transfer engine would be `^dar1.1<prep>$ ^sahrā1.3<n-irreg>$`. The bilingual lexicon entry for the first Persian word-sense is shown in (69).

(69)
```
<e><p><l>1.1در<s n="prep"/></l><r>1.1دورون<s n="post"/></r></p></e>
```

Since the entry maps to a word with the grammatical category tag `<post>`, it passes the test in (67) and proceeds to the test in (68). This word ends in a vowel, so the tag `<gen>` does not get added. The noun gets outputted and then the postposition. The result is `^sahrā1.2<n>$ ^durun1.1<post>$`. After the synthesis process, **STAMP** produces the desired Gilaki phrase sahrā durun.

Another change I made at this stage was to remove rule 1. The noun-adjective pattern of the rule was being matched more often than desired and causing other rules not to be executed.

6.4.7 Stage 6 - Indefinite Nouns

In stage 6 I converted Persian indefinite nouns to a Gilaki construction using an indefinite article before the noun. I added two new rules to handle this, rules 0.7 and 3.3. The first rule is straightforward and is shown in (70).

(70)
```
<rule comment="0.7 - convert the indefinite enclitic to the Gilaki indefinite particle">
  <pattern>
    <pattern-item n="nom_ind"/>
  </pattern>
  <action>
    <out>
      <lu>
        <lit v="ایتأ1.2"/>
        <lit-tag v="indf"/>
      </lu>
      <b/>
      <lu>
        <clip pos="1" side="tl" part="lem"/>
        <clip pos="1" side="tl" part="tl_pos"/>
        <clip pos="1" side="tl" part="nbr"/>   <!-- if it exists -->
        <clip pos="1" side="tl" part="ezafe"/> <!-- possible ezafe -->
      </lu>
    </out>
  </action>
</rule>
```

This rule matches a pattern named `nom_ind` which is basically any noun containing the tag `<ind>`. If this is matched, the indefinite particle `ita1.2`[6] will be outputted by the rule and then components of the noun. The key point is that the indefinite enclitic is omitted from the noun output. Taking the Persian word muši meaning 'a mouse', this would be translated to Gilaki as ita muš. The data stream coming into the transfer engine would be `^muš1.1<n><ind>$`. The rule would add the Gilaki particle ita and remove the `<ind>` tag. The result is `^ita<indf>$ ^muš1.1<n>$`. When synthesis is done we have ita muš.

I added rule 3.3 to handle an indefinite noun following a preposition. This was needed because rule 3.5's pattern meant that all prepositions followed by nouns were funneled to it. Keep in mind that the longest pattern matched gets executed first, so a two word pattern gets executed before a one word pattern. Rule 3.5 did not have the ability to handle indefinite nouns. I elected to create a new rule to do this instead of modifying the existing preposition rule. The reason is that rule 3.5 handles possessive pronominal enclitics as discussed in Section 6.4.4, but the indefinite enclitic and possessive pronominal enclitics are mutually exclusive. Therefore I did not need a rule that would handle both situations. A separate rule for each situation made more sense. Rule 3.3 uses the pattern shown in (71).

---

[6] In Arabic script ایتأ1.2

(71)
```
<pattern>
  <pattern-item n="prep"/>
  <pattern-item n="nom_ind"/>
</pattern>
```

Since this is also a kind of preposition-noun pattern like in rule 3.5, I had to be careful to put the new preposition indefinite noun rule before rule 3.5. In other words, I put the rule with the more restrictive pattern first. If I did not, the transfer engine would execute rule 3.5 for indefinite nouns following prepositions and would never reach the new rule. The new rule is basically the same as 3.5, except that every place where output occurs, an output section like that in (70) is used. As an example, the Persian phrase tuye surāxi, 'into a hole' translates to ita surāxə mian in Gilaki.

6.4.8 Stage 7 - Custom Bilingual Lexicon Entries

In stage 7 I added some custom entries to the bilingual lexicon and replaced some existing entries. These are to handle idiosyncratic translation issues with words. The additions and replacements are shown in (72) and (73) respectively.

(72)
```
<e><p><l>1.1همه<s n="quant"/><s n="pc_3sg"/></l><r>1.1همش<s n="n"/></r></p></e>
<e><p><l>شدا2.1تن<s n="aux"/><s n="3sg_pst"/></l><r>1.1کره<s n="prt"/></r></p></e>
```

(73)
```
<e><p><l>2.1سیر<s n="adj"/></l><r>3.1سر<s n="adj"/><s n="pfv_sfx"/></r></p></e>
<e><p><l>1.1ول<s n="n"/></l><r>1.1ویل<s n="adj"/><s n="pfv_sfx"/></r></p></e>
```

In (72) two inflected Persian forms are translated to uninflected Gilaki forms. In (73) two uninflected forms are translated to inflected forms.

6.4.9 Stage 8 - Accusative Pronouns

In stage 8 I handled accusative pronouns. Persian may mark a personal pronoun used as a direct object with the postposition rā. The equivalent in Gilaki is a one-word accusative/dative pronoun. To get this working I modified rule 2. I modified the pattern to include pronouns along with nouns and adjectives. The pattern is shown in (74). Note, there is only one postposition in Persian which is rā.

(74)
```
<pattern>
  <pattern-item n="nom-adj-pro"/>
  <pattern-item n="post"/>
</pattern>
```

In the body of the rule I added a call to a new macro for assigning accusative Gilaki pronouns to a variable if there is a match with a Persian pronoun. The call is shown in (75) and a portion of the macro is shown in (76). These are very similar to what is shown in (64) and (65) for the genitive pronouns.

(75)
```
<call-macro n="acc_pro_macro">
    <with-param pos="1"/>
</call-macro>
```

(76)
```
<def-macro n="acc_pro_macro" npar="1">
  <let> <!-- initialize the variable to null -->
    <var n="acc_pro"/>
    <lit v=""/>
  </let>
  <choose>
    <when> <!-- 2sg -->
      <test>
        <equal>
          <clip pos="1" side="sl" part="lem"/>
          <lit v="1.1تو"/>
        </equal>
      </test>
      <let>
        <var n="acc_pro"/>
        <lit v="1.1ترَ"/>
      </let>
    </when>
    <when> <!-- 3sg u -->
      <test>
```

One difference between the genitive macro and the accusative one is that in the latter the test is made on the lemma whereas in the former the test is made on the enclitic tag. After the macro has been called, the rule tests to see if there is something in the `acc_pro` variable and if so it is outputted as well as the grammatical tag `<perspro>`. This is shown in (77).

(77)
```
<when> <!-- if acc_pro variable is not null, output the acc form in Gilaki -->
  <test>
    <not>
      <equal>
        <var n="acc_pro"/>
        <lit v=""/>
      </equal>
    </not>
  </test>
  <out>
    <lu>
      <var n="acc_pro"/>
      <lit-tag v="perspro"/>
    </lu>
  </out>
</when>
```

Let us take as an example the Persian phrase mā rā, ‘they DO’[7] which in Gilaki would be ušana. The data coming in would be `^ānhā1.1<pro>$ ^rā1.1<post>$`. After rule application the data stream would be `^ušana1.1<perspro>$`. After synthesis we have simply ušana as desired.

6.4.10 Stage 9 - Possessive Enclitics and Prepositions

In stage 9 I handled nouns with possessive pronominal enclitics that follow prepositions. Rule 3.5 created in stage 5 matched the pattern `preposition-noun`. This meant that the rule 0.5 created to handle possessive pronominal enclitics was bypassed because rule 3.5, having the longer pattern, had priority. In this stage, I added the handling of

[7] DO = direct object marker

possessive pronominal enclitics even when they are attached to something followed by a preposition. The main change to rule 3.5 was calling the macro shown in (65) to assign a variable to the Gilaki genitive pronoun that corresponds to a possible enclitic on the noun. In addition, the genitive pronoun is outputted if necessary by rule 3.5. For example, the Persian bā suzanhayemān, meaning ‘with our needles’ translates to the Gilaki ami suzananə amra. Here the preposition is translated to a postposition and the enclitic becomes a genitive pronoun at the start of the phrase.

6.4.11 Unhandled Phenomena

In total I wrote 17 rules to do translation between Persian and Gilaki. They are shown in Table 3. There are still some phenomena that are not handled with these rules. Below is a description of some of the significant ones.

Destination phrases that accompany verbs like ‘to go’ in Gilaki are typically put after the verb. In Persian they come in front of the verb as a prepositional phrase. This moving of a prepositional phrase after the verb was unhandled in this experiment. To be able to handle this phenomenon, **Apertium**’s advanced-transfer system would need to be used.

The authors of the reference translations consistently translated Persian verbs in the imperfective aspect into verbs with continuous aspect. Gilaki also has an imperfect aspect and the MT system correctly translated the verbs to this form, but no attempt was made in this experiment to try and predict when a continuous verb form should be used.

The benefactive construction was not handled except for pronouns (rule 3). Persian uses a prepositional phrase for a benefactive construction, but Gilaki uses a benefactive case marker.

Gilaki sometimes omits the perfective prefix on verbs and instead uses an enclitic on the preceding word. In this experiment, I did not try to predict when this enclitic would be used.

6.5 Final Rules

Num.	Rule	Description
0.5	possessive pronominal enclitics	Convert Persian possessive pronominal enclitics to Gilaki genitive pronouns.
0.7	indefinite enclitics	Convert Persian indefinite enclitics to the Gilaki indefinite particle.
2	accusative postposition	Change the Persian accusative postposition to the Gilaki accusative enclitic.
2.5	something plus ham	Convert noun/adv/pro plus the adverb ham to an enclitic.
2.6	something plus ra plus ham	Convert noun/adv/pro plus ra plus the adverb ham to an enclitic and omit the accusative.
3	preposition and pronoun	Change the prepositions be and barāye to the appropriate case enclitics. Also change applicable prepositions to postpositions.
3.3	preposition to postposition I	Convert a preposition to a postposition for phrases where the noun is indefinite.

Num.	Rule	Description
3.5	preposition to postposition II	Convert a preposition to a postposition for phrases where the noun is a possessive pronominal enclitic.
4	imperfective	Change the Persian imperfective prefix to the Gilaki imperfective suffix.
5	infinitive verbs	Do the lexical change and remove features.
6	past copular verbs	Do the lexical change and remove features.
7	past verbs	Add the form-building prefix.
8	present participle verbs	Do the lexical change and remove features.
9	imperative verbs	Change affix names.
10	present verbs	Remove the mi prefix if present.
11	present copular verbs	Change affix names.
12	3sg copula	Change it to an enclitic.

Table 3 Final Transfer Rules

6.6 Discussion

The stage 1 BLEU score of 23.65 is dramatically higher than the baseline score of 0.82. This is not surprising because the baseline score represented running the MT system using the same lexicons and rule set used in developing the system. I used a six-sentence text during development. At stage 1, I added many new lexical entries and I added links between source and target word-senses.

At stage 2, I achieved the second highest increase in BLEU score, 11.0%. Stage 2 improvements had to do with fixing synthesis problems. Without these fixes, the MT system produced wrong morpheme combinations. Correcting these meant that many more words were well-formed. The significant improvement shows the importance of well-defined environment constraints in the target lexicon in order to tell the MT system which allomorphs are appropriate in which context.

Stage 3 is the first stage where I actually made rule changes. One rule was added and one rule was modified. The changes had to do with converting Persian possessive pronominal enclitics to separate words in Gilaki. I achieved a modest increase of 1.6% by making these changes.

Stage 4 changes that I made produced the biggest improvement over any other changes. By adding rules to change the Persian adverb ham to an enclitic I achieved a 11.7% improvement. The improvement is probably so large because of the frequency of the adverb in the text (15 times) and also the fact that encliticizing the word made many n-grams much closer to the reference texts.

In stage 5 I added a rule to turn applicable Persian prepositions into Gilaki postpositions. By adding this rule I achieved a good 4.5% increase in the BLEU score. The rule I added was pretty basic in that it only found matches of preposition followed by noun which covered the majority of cases.

In stage 6 I added two rules to handle indefinite nouns. These rules convert a Persian enclitic to a Gilaki word. Adding these rules gave me a nice 5.8% increase in the BLEU score. The Persian indefinite enclitic occurred 17 times in the source text.

In stage 7 I added customization to the bilingual dictionary as described in Section 5.6.4. These changes made a modest 1.2% increase in the BLEU score, but they were a good illustration of how idiomatic words could be handled in the MT system.

By modifying a rule to handle accusative pronouns in stage 8, I only achieved a small 0.4% increase in the BLEU score.

I did my final changes to the rule set in stage 9. This had to do with handling possessive enclitics that followed prepositions. In stage 3 I had handled possessive enclitics in general but not after prepositions. It turned out that most of the possessive enclitics occurred after prepositions, so making a rule change to handle this situation gave a nice 4.6% bump in the BLEU score.

Table 4 shows a comparison between the original Persian text and the generated Gilaki text. The average sentence length is shorter in the Gilaki text because the system converted many separate words in Persian to enclitics in Gilaki. Rules 2, 2.5, 2.6, 3, and 12 did this kind of conversion. Countering this were rules 0.5 and 0.7, where the system converted Persian enclitics to separate words in Gilaki. The former rules, however, outweighed the latter giving shorter average sentence lengths in Gilaki.

	Persian Source Text	**Generated Gilaki Text**
Sentence Count	55	55
Total Word Count	573	542
Average Words per Sentence	10.4	9.9

Table 4 Text Statistics

Looking at all the stages together, I started with a BLEU score of 23.65 after lexical coverage was added, and after stage 9 I had a score of 34.98 — an increase of 47.9%. Just looking at the improvement made through rule changes and additions, there was an increase of 33.3% in the BLEU score. This seems to be a good result for adding seven new rules and modifying several existing rules. As mentioned in Section 6.4.11, phrase transformations could help improve the system. This would mean extending the MT system to use **Apertium**'s advanced transfer engine.

The results of this experiment are encouraging. With just 17 rules I achieved reasonable translation quality for two related languages. For two languages that are more closely related the same quality could probably be achieved with fewer rules. Unrelated languages would likely require more rules to achieve the same quality. Of course 17 rules is just a start; a person would need to add many more rules to get good quality translation that would cover all parts of

the language. The good news, though, is that reasonable machine translation quality with this system is possible with relatively few rules.

7

Conclusion and Future Work

This thesis has shown that it is possible to create a linguist-friendly MT system for use in low-resource languages. By combining three different free applications, **FLEx**, **Apertium** and **STAMP**, I have created a system that successfully employs the classic analysis-transfer-synthesis paradigm. **FLEx** is at the core of the system and is the chief reason that this is an MT system that is easy for ordinary linguists to use. A text to be translated starts in one **FLEx** project and turns into translated text in the other **FLEx** project. The bilingual dictionary is easily created by linking source and target word-senses. Analysis is done in **FLEx**'s intuitive parsing system. **Apertium**'s transfer engine does the heavy lifting for transforming the source text via transfer rules. It provides a rich and powerful language in which to write the rules. The language is well-documented with lots of examples freely available. There is even live chat help for **Apertium**'s tools. Synthesis is done via **STAMP** which operates with environment constraints in almost exactly the same way that **FLEx** does which means constraints in the target lexicon do not need to be modified.

This MT system meets the aforementioned requirements: that the lexicon tool be one that is common to linguists, that it be built on a normalized database, that it be multi-purpose — serving as a tool for creating lexicons from scratch and for holding all manner of lexical information. It meets the requirements that there be a system for holding and maintaining rules which in this case is a well-defined human-readable XML file. As required, rule components can be modular, and the rule-description language is rich. This MT system also meets the requirements in regards to ambiguity. Most ambiguity in this system is dealt with at the analysis stage which means the text coming out of the analysis stage is completely unambiguous. Ambiguity in translation between source and target words is solved to a large extent by being able to map source words to target words at the sense level as well as the option to use transfer rules to handle remaining ambiguity.

Besides being a system that connects three applications to do MT, this system offers features that allow it to handle complex entries, unknown words and variant entries. It is able to do wholesale mapping of grammatical categories, make customized bilingual lexicon changes, make use of inflection information in the lexicons and use a default mapping system from source word-sense to identical target word-sense if no link to a target word-sense is given.

Other existing MT systems are not suitable. Either they are not free to use, or they are too complex for the ordinary linguist or they are not flexible enough to work with any language pair. This system, however, is free, simple to use and flexible. It allows the linguist to gradually increase the power of the system by increasing the complexity of constraints and rules.

I have proved the system successful by implementing a test case which translates texts from Persian to Gilaki. In an experiment with a previously unseen text, I attained a BLEU score of 35.0 with 17 rules and showed how the addition of eight of these rules improved the BLEU score by 33%. I achieved this just by using **Apertium**'s shallow-transfer engine without phrase-level changes. Extending this MT system to use **Apertium**'s advanced-transfer engine which would enable phrase-level changes, would lead to an even higher BLEU score.

Although creating lexicons and writing transfer rules is a time-intensive process, it pays dividends in the fact that MT can be done in languages that are considered low-resource — where the absence of parallel texts means statistical MT will give subpar results. The success of a setup for a language pair depends mainly on how well transfer rules can be written for the pair. In essence almost any two languages can be implemented in this MT system.

While much has been accomplished here, there are many things that could be done in the future to improve this MT system. First and foremost, since this system is a proof-of-concept system which has only been tested with one language pair, it should be tested with many more languages to see what kind of issues come up. Other improvements fall into categories dealing with **FLEx**, user interface, **Python** scripts, and other things.

In the category of **FLEx**-related improvements, changing from the current three-step system using three different applications to integrating the whole system into **FLEx** would be a big improvement. It would be ideal to push one button in **FLEx** to execute everything. **FLEx** is open-source, so anyone could take this on, but having the **FLEx** development team do it would probably be preferable. Another improvement would be to develop an automated way to link entries between lexicons. It would be very useful if the user could select a source text that has been fully analyzed and have **FLEx** prompt the user for every sense that has not yet been linked to a target sense. **FLEx** could suggest a target sense by attempting to make a match on the gloss field. In the future it would also be useful if the resulting text in the target **FLEx** project could end up not just as paragraphs in a target text, but instead as a fully analyzed text. All the pieces are available in the MT system at the point where **STAMP** is being executed, so it is feasible. It would just be a matter of validating the lexical entries, adding parses to the analysis list for each word form and adding all the parses to the interlinear text. A last **FLEx**-related improvement would be the ability to link a source sense directly to a target sense instead of the current method of linking to an entry and specifying a sense number. The current method leads to potential problems if the order of senses is changed.

In the category of user interface improvements, one helpful change would be a better system for interacting with transfer rules. The way transfer rules are created and maintained in a XML text file is perhaps the most unfriendly part of this MT system right now. Developing a windows type user interface for interacting with these rules would be a big improvement. Errors can occur at every stage of the system. Right now the errors must be dealt with by examining text files. Having debugging aids in a windows type user interface would make the system more user-friendly.

Some improvements could be made to the **Python** scripts as well. For one thing, it would be useful in the future to be able to identify punctuation in the source text. By doing so, transfer rules could check punctuation as part of their logic. Also, in the future it is necessary that the system be able to support infixes and subsenses.

There are at least two other improvements that would be beneficial. It will be necessary that this MT system support the advanced transfer engine of **Apertium** so that phrase-level syntactic transfer can be performed. This should not be a difficult change. It is probably just a moderate change to the structure of the system — specifically the interface to the **Apertium** tools. It would also be very useful to have a way to integrate a test suite into the system so that regression testing could be done after changes are made to the translation setup.

I close by reiterating the main achievement of this work. I have created a proof-of-concept MT system that is linguist-friendly using available tools. It has been proved successful in one test case and should be effective in doing MT in other language pairs — especially in low-resource languages where other MT options are limited.

A

Installing and Running the System

This appendix describes how to install and run the MT system.

A.1 Installation

A.1.1 Install FLExTools

In order to run the MT system, the open-source **FLExTools** application needs to be installed. **FLExTools** is an environment where **Python** scripts can be run to access and/or modify **FLEx** databases. This MT system is run within the **FLExTools** environment.

To Install **FLExTools**, go to this page: https://github.com/cdfarrow/FLExTools/wiki, navigate to the **Setup** section and download the zip file of the latest version of **FLExTools**. At the time of writing, the file to download was named: `FLExApps1.2.3.zip`. Extract the contents of this file to your hard drive. There is no Windows installation process; **FLExTools** is run from this folder directly.

A.1.2 Download MT Modules

Download the **Python** MT scripts from this link: https://github.com/rmlockwood/FLExTrans/raw/master/FlexTrans.zip.

Extract the contents of the zip file to the `FlexTools` folder under the `FlexAppsN.N.N` folder that you created above. You may be asked to confirm the overwriting of two or more files. Please overwrite them.

STAMP is installed automatically in the top level folder when you extract the zip file contents. You do not need to do a separate step. You can download it yourself if you want from the following URL: http://carla.sil.org/Corporate_Release/DOSVersions/stamp32-221.zip.

A.2 Running

The basic procedure for running the MT system is to first run a "collection" of two scripts, then do the transfer process, then run a "collection" of four scripts. The following sections go into more detail and explain the setup steps as well.

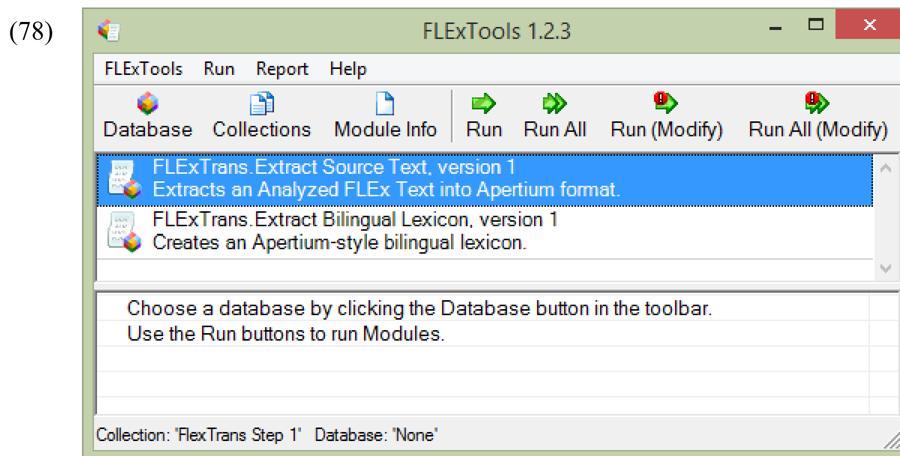
A.2.1 Edit the Configuration File

Before you run the MT system you want to make sure your configuration file is correct. You will find the configuration file in the `FlexTools` folder of the `FlexAppsN.N.N` folder. It is called `FlexTrans.config`. Edit this file in any text editor. The important properties to set are: `SourceTextName`, `TargetProject`, `SourceCustomFieldForEntryLink` and `SourceCustomFieldForSenseNum`. Refer to 4.4.3 for information on how to create the custom fields that these properties use. You may find it convenient to set the path of the following properties to the same folder: `SourceCustomFieldForEntryLink`, `AnalyzedTextOutputFile` and `BilingualDictOutputFile`. If you do not set the paths of these properties, they will default to the `FlexTools` folder. See Appendix B for more details.

*A.2.2 Start **FLExTools***

The next thing to do is to start **FLExTools** which you do by double-clicking on `FlexTools.vbs`. The program should look like (78) when it first starts up:

(78)



A.2.3 Run Step 1 Collection

The two modules for the collection “FLExTrans Step 1” should have been loaded by default. If not, select the collection using the ***Collections*** button.

Before you run the collection, you need to choose your source database as the database on which these modules will be run. Click ***Database*** and choose your source **FLEx** project. Note: close both your source and target FLEx projects before running the collection; otherwise you will receive an error. Now you want to run the modules in the current collection. You can run each module separately or all together. To run them all together, Click on the ***Run All*** button. You will see output in the lower pane of the **FLExTools** window. Now you are ready for step two.

A.2.4 Run the Transfer Module (Step 2)

The next step is to run the transfer module. Go to the following website: http://uakari.ling.washington.edu/flextrans/ and follow the instructions on the page where you are asked to upload three files. Two of the three files to upload will be found in the folders that you specified in the configuration file for the properties: `AnalyzedTextOutputFile` and `BilingualDictOutputFile`. The third file, `transfer_rules.t1x`, you will have edited yourself.

Immediately after these files are uploaded, the file `target_text.aper` is created on the web server. Check the log file to see that everything worked correctly. The `log` file will have output that looks something like this in the normal case:

```
Sat Apr 4 11:48:37 PDT 2015
lt-comp lr bilingual.dix bilingual.bin
main@standard 4579 8766
apertium-preprocess-transfer transfer_rules.t1x transfer_rules.t1x.bin
cat source_text.aper | apertium-transfer transfer_rules.t1x transfer_rules.t1x.bin
bilingual.bin > target_text.aper 2>>err_log
```

Note: you may only see one or two of the above lines if only one or two input files are uploaded.

The `err_log` file will just show a date-time stamp if it worked normally. Otherwise an error or a warning will be displayed at the end of the file.

If there are no errors, save the file `target_text.aper` via the given link on the web page to the folder you specified in the `TargetTranferResultsFile` property of the configuration file. At this point you are ready to do step three.

A.2.4.1 Apertium Commands Reference

For reference, here are the **Apertium** commands that are being executed in Step 2 and their function in this MT system. Usage examples of the commands are shown in the example output above.

lt-comp	Compiles a bilingual dictionary into a binary transducers file.
apertium-preprocess-transfer	Preprocesses transfer rules into a binary file.
apertium-transfer	Applies transfer rules and a bilingual dictionary to a source text resulting in a target text.

A.2.5 Run Step 3 Collection

The last step is to run the final modules. Open another collection by clicking on the **Collections** button and selecting "FLExTrans Step 3". This time you need to use the button **Run All (Modify)** since we will be modifying the target project, namely inserting the translated text.

Again you will see output in the lower pane of the **FLExTools** window. The last module creates a text with the same title as the source text in the target **FLEx** project. Intermediate files are created in the Windows temp folder.[1] You are finished! Open the target **FLEx** project and see the results. Note: you may need to click on the **Refresh** button in **FLEx** before the new text name will show in the list.

[1] If you type %TEMP% into the address bar of file explorer, it will take you to the Windows temp folder.

B

Configuration File

Description of the properties and values in the configuration file.

Property	Description
SourceTextName	The name of the text (in the first analysis writing system) in the source **FLEx** project to be translated.
AnalyzedTextOutputFile	The path and name of the file which holds the extracted source text.
TargetOutputANAFile	The filename of the file holding the intermediary text in **STAMP** format. This file is created in the Windows temp folder (%TEMP%).
TargetOutputSynthesisFile	The filename of the file holding the intermediary synthesized file. This file is created in the Windows temp folder.
TargetTranferResultsFile	The path and name of the file which holds the text contents after going through the transfer process.
SourceComplexTypes	One or more complex types from the source **FLEx** project (separated by commas). These types will be treated as a lexical unit in the MT system and whenever the components that make up this type of complex form are found sequentially in the source text, they will be converted to one lexical unit. See Section 5.9.1.1 for more details.
SourceCustomFieldForEntryLink	The name of the custom field in the source **FLEx** project that holds the link information to entries in the target **FLEx** project. See Section 4.4.3 for more information.
SourceCustomFieldForSenseNum	The name of the custom field in the source **FLEx** project that holds the sense number of the target entry. See Section 4.4.3 for more information.
BilingualDictOutputFile	The path and name of the file which holds the bilingual lexicon.

Property	Description
BilingualDictReplacmentFile	The path and name of the file which holds replacement lines for the bilingual lexicon. See Section 5.9.1.2.
TargetProject	The name of the target **FLEx** project.
TargetPrefixGlossListFile	The ancillary file that hold a list of prefix glosses from the target **FLEx** project.
TargetComplexFormsWithInflectionOn1stElement	One or more complex types from the target **FLEx** project (separated by commas). These types, when occurring in the text file to be synthesized, will be broken down into their constituent entries. Use this property for the types that have inflection on the first element of the complex form. See Section 5.9.3.2.
TargetComplexFormsWithInflectionOn2ndElement	Same as above. Use this property for the types that have inflection on the second element of the complex form.
TargetMorphNamesCountedAsRoots	Morpheme types in the target **FLEx** project that are to be considered as some kind of root. In other words, non-affixes and non-clitics.
SourceMorphNamesCountedAsRoots	Same as above for the source **FLEx** project.
CategoryAbbrevSubstitutionList	One or more pairs of grammatical categories where the first category is the “from” category in the source **FLEx** project and the second category is the “to” category in the target **FLEx** project. Use the abbreviations of the **FLEx** categories. The substitution happens in the bilingual lexicon. See Section 5.9.1.2 for more information.
CleanUpUnknownTargetWords	“y” (without quotes) to do the cleanup; “n” otherwise. If set to “y”, the system will remove preceding @ signs and numbers in the form N.N following words in the target text.

C

Troubleshooting

This appendix has some helpful information about troubleshooting the MT system.

If you see an @ sign in your text, this means that **Apertium** could not find the lemma in the bilingual lexicon.

If you see something like this: `word1.3` i.e. a target word with numbers after it in your target text, it means that **Apertium** found no mapping for the source lemma.

If you see something like this: `%0%word1.3%`, **STAMP** could not synthesize the word.

If you find an affix gloss attached to a target word in your target text, this means that **STAMP** did not find the affix in the target lexicon.

A good method of tracking down what is going wrong in a particular situation is to put just one word into your source text file and try and get that word to translate correctly.

The following table shows the modules in sequence and the files they create. It may be helpful in tracking down problems.

Module	**File Name Property**	**Default Name**	**In Temp Folder**
Extract Source Text	AnalyzedTextOutputFile	source_text.aper	No
Extract Bilingual Lexicon	BilingualDictOutputFile	bilingual.dix	No
Transfer Module	TargetTranferResultsFile	target_text.aper	No
Catalog Target Prefixes	TargetPrefixGlossListFile	target_pfx_glosses.txt	No
Convert Text to STAMP Format	TargetOutputANAFile	myText.ana	Yes
Extract Target Lexicon	TargetOutputSynthesisFile	myText.syn	Yes
Extract Target Lexicon	TargetProject	<TargetProject>_pf.dic	Yes
Extract Target Lexicon	TargetProject	<TargetProject>_sf.dic	Yes

Module	File Name Property	Default Name	In Temp Folder
Extract Target Lexicon	TargetProject	<TargetProject>_rt.dic	Yes
Extract Target Lexicon	TargetProject	<TargetProject>_stamp.dec	Yes

References

Apertium. 2015. Apertium, A free/open-source machine translation platform. http://apertium.org (accessed 5/11/2015)

Black, H. Andrew, David Weber. 1987. What is CADA. *Notes on Computing* 6(4):11-15.

Black, H. Andrew. 2014. A Conceptual Introduction to Morphological Parsing for Stage 1 of the Fieldworks Language Explorer. SIL International. Manuscript.

Buseman, Alan. 1991. Sentence Transfer Program Reference Manual. SIL International. Manuscript.

CARLAStudio. 2008. [Software]. http://carla.sil.org/carlastu.htm (accessed 5/20/2015)

Fieldworks Movies. 2009. Using the Parser (Interlinear). http://downloads.sil.org/FieldWorks/Movies/parser%20menu.html (accessed 3/2/2015)

FLEx. 2014. Fieldworks Language Explorer (Version 8.1.2) [Software]. http://fieldworks.sil.org/download/fw-812/ (accessed 5/20/2015)

Forcada, Mikel L., Bonev, Boyan Ivanov, Rojas, Sergio Ortiz, Ortiz, Juan Antonio Perez, Sanchez, Gema Ramirez, Martinez, Felipe Sanchez, Armentano-Oller, Carme, Montava, Marco A., Tyers, Francis M. 2010. Documentation of the Open-Source Shallow-Transfer Machine Translation Platform Apertium. Departament de Llenguatges i Sistemes Informàtics Universitat d'Alacant. Manuscript.

Graphite2 (Version 1.2.4). 2014. [Software]. http://scripts.sil.org/cms/scripts/page.php?site_id=projects&item_id=graphite_home (accessed 5/20/2015)

King, Linda. 2003. Education in a Multilingual World. Paper presented at the UNESCO. Paris.

Koehn, Philipp, Jean Senellart. 2010. Convergence of Translation Memory and Statistical Machine Translation. Paper presented at the AMTA Workshop on MT Research and the Translation Industry. Denver.

Lewis, M. Paul, Gary F. Simons, and Charles D. Fennig (eds.). 2015. Ethnologue: Languages of the World, Eighteenth edition. http://www.ethnologue.com (accessed 3/3/2015)

Lockwood, Ron. 2011. Machine Parsing of Gilaki Verbs with Fieldworks Language Explorer. SIL International. Manuscript. https://iranianlinguistics.org/publications/nlp/Lockwood2011

Madnani, N. 2011. iBLEU: Interactively Scoring and Debugging Statistical Machine Translation Systems. In *Proceedings of the Fifth IEEE International Conference on Semantic Computing (Demos).* 213-214. Palo Alto, CA.

Moe, Ronald. 2009. Introduction to Lexicography for Fieldworks Language Explorer. SIL International. Manuscript.

Mouse. n.d. موش گرسنه muš-ə gorosne, The Hungry Mouse. http://www.koodakaneh.com/story.aspx?Id=129 (accessed 3/4/2015)

OpenLogos. 2005. OpenLogos. https://sourceforge.net/p/openlogos-mt/wiki/The%20Logos%20Machine%20Translation%20System/ (accessed 5/5/2015)

Parser. 2011. Using the Parser. http://downloads.sil.org/FieldWorks/Movies/44-using%20parser.mp4 (accessed 5/11/2015)

Ramírez-Sánchez, Gema, Felipe Sánchez-Martínez, Sergio Ortiz-Rojas, Juan Antonio Pérez-Ortiz, Mikel L. Forcada. 2006. Opentrad Apertium open-source machine translation system: an opportunity for business and research. In *Proceedings of Translating and the Computer 28 Conference.* pages unknown. London.

Rastorgueva V, Kerimova A, Mamedzade A, Pireiko L, Edel'man D, Lockwood R M. 2012. The Gilaki Language. *Studia Iranica Upsaliensia.* 19. Uppsala: Acta Universitatis Upsaliensis. http://uu.diva-portal.org/smash/get/diva2:560728/FULLTEXT02.pdf

Scott, Bernard. 2003. The Logos Model: An Historical Perspective. *Machine Translation* 18(1):1-72.

Senellart, Peter Dienes, Tamas Varadi. 2001. New Generation Systran Translation System. Paper presented at the MT Summit VIII. Spain.

SYSTRAN. 2015. SYSTRAN. http://www.systransoft.com/ (accessed 5/20/2015)

Thackston, W. M. 1993. *An Introduction to Persian.* Bethesda: Ibex Publishers.

Weber, David J., William C. Mann. 1981. Prospects for Computer-Assisted Dialect Adaptation. *Computational Linguistics* 7(3):165-177.

Weber, David J., H. Andrew Black, and Stephen R. McConnel. 1988. AMPLE: A Tool for Exploring Morphology. *Occasional Publications in Academic Computing,* 12. Dallas, TX: Summer Institute of Linguistics. http://www.sil.org/computing/catalog/stamp.html

Weber, David J., H. Andrew Black, Stephen R. McConnel, and Alan Buseman. 1990. STAMP: A Tool for Dialect Adaptation. *Occasional Publications in Academic Computing,* 15. Dallas, TX: Summer Institute of Linguistics. http://www.sil.org/computing/catalog/ample.html