Development of a Tokenizer and a Rule-Based Part-of-Speech Tagger in Korean

by

Philip D Kim

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Electrical Engineering and Computer Science and Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 22, 1998

Author

Department of Electrical Engineering and Computer Science
May 22, 1998

Certified by

Dr Clifford Weinstein
Group Leader, MIT Lincoln Laboratory
Thesis Supervisor

Certified by .

Dr Young-Suk Lee
Staff Member, MIT Lincoln Laboratory
Thesis Supervisor

Accepted by

Arthur C Smith
Chairman, Departmental Committee on Graduate Students

Development of a Tokenizer and a Rule-Based Part-of-Speech Tagger in Korean

by

Philip D Kim

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degrees of Bachelor of Science in Electrical Engineering and Computer Science and Master of Engineering in Electrical Engineering and Computer Science

Abstract

The Korean-to-English machine translation subsystem of CCLINC (Common Coalition Language System at Lincoln Laboratory) is being developed at the Information Systems Technology Group, M I T Lincoln Laboratory This thesis focuses on the development of a sub-module of the Korean understanding system, namely, the tokenizer and a rule-based part-of-speech tagger The development of the morphological analyzer on the Military Communications Messages Data Set, the Combat Briefer's Course Manual Data Set, and the Naval Operations Message Domain are discussed at length Experimentation on these data sets show a need for multiple tokenization and numeric tokenization The motivations for developing a Rule-Based Part-of-Speech tagger are discussed, as well as the experimentation of this tagger on the Military Communications Messages Data Set Finally, possible future work on the Part-of-Speech Tagger and the Tokenizer are discussed.

Thesis Supervisor Clifford Weinstein
Title Group Leader, MIT Lincoln Laboratory
Thesis Supervisor Young-Suk Lee
Title Staff, MIT Lincoln Laboratory

Acknowledgements

I would like to thank the following individuals for their help on this thesis First and foremost is Dr Young Suk Lee, who patiently offered me guidance through every step of this thesis Without her help, I would not have been able to get through the first line of code in the translation system I would also like to thank Dr Clifford Weinstein, without whom I would not have had the chance to work with the forefront of machine translation technology In addition, I would like to thank Linda Kukolich, who was always there to offer me advice on everything from perl scripts to cooking And finally, I would like to thank Sadiki Mwanyoha, who was a fellow RA student, and offered me valuable insight into my thesis

I would also like to thank Yong H Lee, who taught me much about the C language and the Korean language and selflessly dedicated himself to assisting me in writing this thesis Our many long nights spent together in front of a SPARCstation will not soon be forgotten Next, I would like to thank Caroline J Kim, who meticulously performed the thankless job of doing the final proofreading And to all my friends, thank you for all your support

Finally, I would like to thank my family To my brother, thank you for always supporting me even when I've been unreasonable And to my parents, thank you for everything Without your moral, financial, and educational support, truly none of this would have been possible

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 General Background

For over forty years, the United States military has been working jointly with the Republic of Korea's armed forces While bilingual translators have been used to handle the exchange of information between the two armed forces, the scarcity of translators, as well as the long latency of human translation, has motivated the United States military to develop a machine translation system To this end, the Information Systems Technology Group at Lincoln Laboratory has been developing the Common Coalition Language System at Lincoln Laboratory (CCLINC) under DARPA sponsorship Refer to [Weinstein et al 1997] Using an interlingua approach to machine translation, the English-to-Korean subsystem of CCLINC successfully translates a broad range of military communications messages and briefings In addition to the English to Korean translation system, the Information Systems Technology Group has also been developing a Korean to English machine translation system My research has been to develop specific submodules of this translation system, the morphological analyzer and a part-of-speech tagger The morphological analyzer accepts Korean sentences, and tokenizes the particles from their respective root words The tokenization routine enables the system to have a compact representation of the lexicon and grammar, improving the system efficiency The morphological analyzer has been tested on several different military corpuses, and is fairly robust Part-of-speech tagging applies to the output of the morphological analyzer This routine produces the part-of-speech sequence of the input sentence, enabling the understanding system to utilize the grammar rules defined in terms of part-of-speech rather than words themselves

Since the grammar rules defined in terms of part-of-speech can recognize a broader range of grammatical patterns than the grammar defined in terms of words, it will enable the system to improve the parsing coverage Regarding the application of part-of-speech tagging technique to English-to-Korean subsystem of CCLINC, refer to [Lee et al 1997],[Weinstein et al 1997], and [Park, 1998]

My thesis includes research on both the part-of-speech tagger and the morphological analyzer However, the morphological analyzer was developed for a much greater period of time, and this thesis will reflect that

1.2 Machine Translation

Since the early development of computers, there has been much study in the development of machine translation While there have been no fully effective solutions, the Information Systems Technology Group at Lincoln Laboratory has been able to make substantial advances in developing a machine translation system

Most machine translation systems use one of three different approaches direct, transfer, and interlingua [Weinstein et al 1997], [Hutchins and Somers 1992] A direct translation system produces a word-for-word translation This approach is ineffective, as the grammar in two different languages is almost never the same For instance, a direct translation system would translate a sentence in the following manner

cip-eyse cassta (I slept at home)

home-at slept

In the above example and throughout the entire thesis, the Korean forms are given in Yale Romanized Hangul [Martin 1992], a representation of Korean text in a phonetic form that uses the Roman alphabet However, our tokenizer used a different convention As long as the convention is consistent throughout the entire system, though, the actual representation does not truly matter Thus, when we show examples of output from the tokenizer, we will use the morphological analyzer's own convention

While this may be of some use to a user, it is not difficult to imagine that the translation would oftentimes be inaccurate A transfer approach involves some intermediate form of analysis, then performs a word-for-word translation While a transfer translation system produces an output superior to that of the direct translation system, this approach still lacks proper syntax and meaning in the output sentence The interlingua approach is fundamentally different from the other types of approaches Instead of a bilingual, word-for-word transfer, the input language is translated into a language- independent meaning representation called the interlingua This interlingua is then used to generate the output language While the interlingua approach presents its own set of difficulties, it is a highly modular method, from the meaning representation, the interlingua can be translated into any language that has a generation module The object of my research, the CCLINC system, uses this interlingua approach Because of this, we needed to develop solely the Korean language understanding module, and did not need to duplicate much work that had already been done for the English to Korean translation system

1.3 CCLINC System

Based on the interlingua approach, the machine translation part of the Speech-to Speech Translation System operates by translating the input language into a meaning representation, called Semantic Frame [Tummala, et al., 1995], from which the output language is generated In specific terms, we use TINA [Seneff 1992] and GENESIS [Glass et al 1994], the language understanding and generation systems developed by the MIT Laboratory for Computer Science under DARPA sponsorship The advantages of the ILT method are numerous, with the most important one being its modularity If an understanding module is developed for one language, that language can be translated into any other language that has a generation module Also, if one language has a generation module, any language that has an understanding module can be translated into that language without the need to extensively re-write a complete translation system The translation system is basically independent of the specific pairs of languages being translated

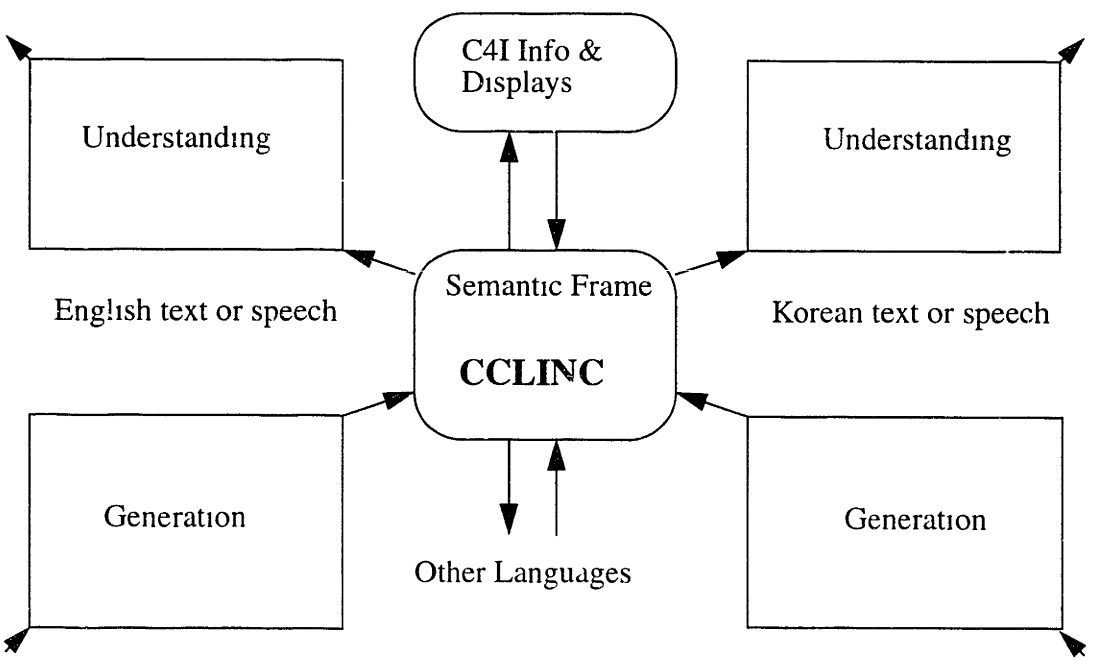

Figure 1-1: Graphical Description of CCLINC

1.4 Thesis Goals and Results

For my thesis, I set out to accomplish two distinct goals First, I was to develop a morphological analyzer for the Korean language understanding module The Korean language differs from the English language in that oftentimes, a particle attached to a root word determines the role the root word plays in a sentence For instance, a word with an attached subject marker will play the role of the subject in a sentence Since the particle can be attached to any noun phrases, and helps to identify the grammatical function [Fromkin and Rodman 1988] of the preceding phrases, tokenization of the particle enables the system to have a much simpler lexicon and grammar The necessity for the morphological analyzer is covered in greater depth later in Chapter 2 of this thesis

In addition, I also set out to develop a part-of-speech tagger for the Korean understanding module The part-of-speech tagger allows the understanding module to deal with a sentence containing unknown words or constructions When an unknown word is encountered during translation, the word is replaced with its corresponding part of speech tag The grammar of the understanding module would then be augmented to handle the part-of-speech tags directly, resulting in an increased parsing coverage, see [Lee et al 1994] for the details

The morphological analyzer was trained on several different domains The first was the Military Communications Messages (ARMYCOM) domain, on which a very basic tokenization algorithm was developed. The Military Communications Messages were provided to us by CECOM via the University of Pennsylvania For other work utilizing the

same set of data for Korean parsing, see [Park 1994], [Egedi et al 1994], and [Dorr. 1997] The ARMYCOM domain was divided into five parts of about 20 sentences each The morphological analyzer would then be run on the first set, and the results would be analyzed Any mis-tokenizations would then be corrected by augmenting the morphological analyzer Once this was complete, the morphological analyzer would be trained on the next set of approximately 20 sentences

Another domain tnat we trained the morphological analyzer on was the COBC domain These were sentences taken from the Combat Briefer's Course manual This domain consisted of nearly 200 sentences and were of a slightly different nature than the ARMYCOM domain Again, the morphological analyzer was trained on this domain using a similar technique as on the ARMYCOM domain Upon coming across the COBC data, we decided that our basic tokenization algorithm required augmentation by implementing a multi-stage tokenization scheme This domain allowed us to develop a multiple morphological analyzer that could more accurately tokenize each word

Finally, we trained the morphological analyzer on the MUC-II domain MUC-II corpus is a collection of naval operational report messages from the Second Message Understanding Conference, which were collected and prepared by the center for Naval Research and Development (NRaD) to support DARPA-sponsored research in message understanding, [Sundheim 1989] Lincoln Laboratory utilized these messages for DARPA-sponsored machine translation research About 300 sentences long, this domain allowed us to determine if our analyzer was prepared to deal with untrained data, and to detect flaws still in the design This domain allowed us to detect the less evident flaws in the translation system, and to make the tokenization algorithm even more robust

The part-of-speech tagger was trained on the ARMYCOM domain. The basis for the part-of-speech tagging was the RULE-BASED Part-of-Speech tagger developed by Eric Brill, [Brill. 1992] and [Brill 1996], both at the University of Pennsylvania and at the Spoken Language Systems Group at MIT Laboratory for Computer Science

This was a system designed to tag any language By training the tagger on a manually tagged corpus, the system would develop configuration files that would either tag a word directly, or in the case of an unknown word, would be able to tag the unknown word from the cues based on prefixes, suffixes, infixes, and adjacent word co-occurrences The biggest challenge in applying the tagger system to the Korean language lied in choosing the set of tags, which will achieve the highest tagging accuracy and reduces the ambiguity of the particular word to be tagged Developing a set of unambiguous tags was a challenging task

In Ch 2, I will discuss some of the basic features of Korean, and the motivation for developing a morphological analyzer Ch 3 will be dedicated to explaining the basic structure of the Korean understanding module, the tokenization algorithm behind the morphological analyzer, as well as the process by which the analyzer was trained and developed Ch 4 will describe the motivations for developing a part- of-speech tagger, the algorithm behind the tagger, and the training of the part of speech tagger Ch 5 will summarize the results of this thesis

Chapter 2

Motivations for the Development of a Morphological Analyzer

2.1 Characteristics of the Korean Language

The morphological rules of a language determine how morphemes combine to form new words [Fromkin, Rodman; 1988] The morphemes of a language are the basic units of meaning, or the most elemental unit of grammatical form. In the English language, certain morphemes change the meaning of a word For instance, the morpheme "ly", added as a suffix to nouns, creates an adjective to modify other nouns, such as in "love" vs "lovely" Suffixes in Korean often play an important role in identifying the grammatical function (e g subject, object, indirect object) of the word to which they are suffixed English determines the grammatical position of a word by either its order, or by special words preceding the word Korean, however, differs in that often times there are particles attached to the end of a word to determine its grammatical position The following example clearly illustrates this point

mari-ka sakwa-lul cohahanta
Mary-Nom Apple-Acc Like
"Mary likes apples"

Thus, in Korean, the *ka*, *lul*, and *ta* all serve to denote the grammatical function of the word to which they are suffixed

The Korean language is distinguished from the English language in many other ways as well Certain subtle meanings do not even have a direct translation into English, an

example being that of the honorific forms of speech As is expected, the basic grammar of the two languages is very different English is characterized by a Subject-Verb-Object order, whereas the word order in Korean is Subject-Object-Verb Other differences between the two languages include

1 The verb comes at the end of a clause.

sunhee-ka youlee-eykey [chak han kwen]-ul cwuessta
SunHi-nom YuRi-dat book one CL-acc gave
"SunHi gave a book to YuRi"

2 Modified verb endings denote negation

sunhee-ka youlee-eykey [chak han kwen]-ul an cwuessta
SunHi-nom YuRi-dat book one CL-acc did not give
"SunHi did not give a book to YuRi"

3 Noun phrases are followed by postpositions

sunhee-ka youlee-eykey [chak han kwen]-ul an cwuessta
SunHi-nom YuRi-dat book one CL-acc gave
"SunHi did not give a book to YuRi"

4 Modifiers precede the word that is modified

EePun sunhee-ka youlee-eykey [chak han kwen]-ul cwuessta
Pretty-adj SunHi-nom Yuri-dat book one CL-acc gave
"Pretty SunHi gave YuRi a book"

[Yang, D , 1996]

The most interesting aspect of these differences lie in the postpositions that follow noun phrases Postpositions are very similar to prepositions in English, with one obvious difference being that they follow the noun phrase to be modified rather than precede it. In

addition, they assist in identifying the role of a word in a given sentence While the condition that the verb comes at the end of a sentence is strictly observed in most cases, there is a great degree of freedom regarding the position of subject and object, as illustrated below.

a sunhee-ka youlee-eykey [chak han kwen]-ul senmwulhayssta
sunhee-nom youlee-dat book one CL-acc gave as a present
'Sunhee gave a book to Youlee as a present'

b sunhee-nom [chayk han kwen]-acc youlee-dat senmwulhayssta
c youlee-dat sunhee-nom [chayk han kwen]-acc senmwulhayssta
d youlee-dat [chayk han kwen]-acc sunhee-nom senmwulhayssta
e. [chayk han kwen]-acc sunhee-nom youlee-dat senmwulhayssta
f [chayk han kwen]-acc youlee-dat sunhee-nom senmwulhayssta

These examples illustrate that for a sentence the verb of which takes three arguments, namely, subject, object and indirect object, all six different permutations of the three arguments are allowed. This has been referred to as "scrambling" Refer to [Lee 1993] for an extensive discussion on the scrambling phenomenon in the Korean language

While by themselves, postpositions have no meaning, when they follow a noun phrase, they provide a relationship between their noun phrase and other words in the sentence. Postpositions come in many different flavors Whereas many of these postpositions have a distant English equivalent, several have no equivalent English word or particle One such postposition is *ka*, which is one of the subject markers For example, the sentence "Mary drove a car" would be written in Korean as

mary-ka cha-lul molass-ta
Mary-nom car-acc drove-declarative
"Mary drove the car"

Clearly, the postposition "ka" define's the noun "Mary"s role as the subject Another type of postposition that is entirely unique to the Korean language are the sentence markers For example, the "ta" at the end of the previous sentence denotes that this is a declarative sentence If the "lass-ta" at the end of the sentence were changed to "lul-ka", the sentence would mean "Would Mary drive a car?". A change of the sentence marker creates a completely different sentence types such as declarative, imperative, interrogative, etc

Thus, by changing the suffix of a Korean word, we can get dramatically different meanings for a the same word This poses a unique hardship on the translation system, from which arises our motivation to develop the morphological analyzer

2.2 Motivation for the Development of a Morphological Analyzer

For any given noun or verb, the translation system should be able to understand the meanings of the word plus any of the many particles that can be attached Thus, a simple personal noun such as "Mary" can end up taking several different meanings depending on the postposition To make matters worse, words can oftentimes take more than one postposition For instance, *cip* means "house" in Korean To express "to the house" in Korean, one would say *cip-ulo* To say "from the house" we would say, *cipulobuthe* If *cip* were used as a subject, we would say, *cip-i* or *cip-un*, depending upon what context the subject was being mentioned If one was to include all of these different forms of "house" into the lexicon (the words of a language, in our case, our system's vocabulary), the size of the lexicon would be staggering Most nouns can be formed with these particles, giving an idea

of the inefficiency of creating a lexicon that includes all the different forms of a noun Also, most verbs can have particles attached to them as well, which makes morphological analysis even more important

To this end, we developed the morphological analyzer The overall morphological analyzer includes a pre-processing stage which takes Korean characters and romanizes them, in addition, this step removes parentheses, quotations, commas, and periods that were deemed to be unnecessary in the translation Next, the analyzer includes a "tokenizer", which takes words, and strips off the particles attached to it, thereby splitting the word up into its basic "tokens" Thus,

cipulo ("to the house")

would become

cip ulo

where *cip* denotes house, and *ulo* is the particle that denotes "to" Thus, instead of having all the combinations of the noun *cip* and all the particles listed in table 2-1 in the lexicon, we need only a single entry specifically for "house" All the particles listed in table 2-1 would have their own entries, and could be analyzed separately from their root words

Even though the development of a good pre-processor is important for the scalability of the system to various types of input, my work entirely deals with the development of the tokenizer, primarily due to time constraints With this in mind, we proceeded to develop the morphological analyzer

Chapter 3

The Development of the Morphological Analyzer

3.1 General Background

As we have shown, a given word in the Korean language can have several particles suffixed to it to denote its grammatical function in a sentence Including in the lexicon all instances of nouns and/or verbs with those particles suffixed to them is highly uneconomical, especially given that the meaning of each particle is regular and predictable Thus, we developed this tokenizer to break up a word into its base noun or verb, and the modifying token (See Appendix 1 for the complete code of the tokenizer) The theoretical portion of this development included researching various Korean textbooks for all the possible particles, and developing conditions for our tokenizer If the tokenizer saw, for example *ulo* at the end of a word, it would tokenize the preceding word and the particle. However, most languages contain exceptions from the general rule, and Korean had a fair number of these exceptions To account for these, we empirically determined what the exceptions are by examining the data No Korean textbook listed every single one, and we had to experiment with the data we had to find them While this may seem simple, exceptions seemed to crop up in just about every single new data set we tokenized An example will clarify this greatly *ttala* means "follow" However, the particle *la* can also be added to the end of a word to establish its grammatical function Our tokenizer algorithm would, then, split up

ttala

into

tta la

considering Ra to be a particle, when in this case, it was not (Again, as we demonstration actual output, we use the tokenizer's romanization convention) While our empirical research gave us a good insight into what the major exceptions were, there were always a few that escaped detections, and became obvious only when a new data set was processed by the tokenizer

The development of the morphological analyzer began with a basic seed program that my thesis supervisor, Dr Young Suk Lee, provided This seed program consisted of an algorithm to read in words, tokenize endings, and then write it back into a file. My task was to take the basic particles that this seed program would recognize, and then extend it to be more robust and general. By using Korean textbooks, we determined what particles existed, and extended the code to recognize these particles, and to strip them off. By reading in about 600 Korean sentences, with lengths varying from 5 to 20 words each, and analyzing the output, we found the inconsistencies, redundancies, and exceptions that needed to be taken into account As of now, we have 80 general rules implemented into our tokenizer, with 26 exceptions

As we developed the tokenizer, we needed to decide exactly what particles needed to be tokenized from their root words Our guidelines consisted of three rules First, if a word with a particle attached could be translated into more than one English word, then we would tokenize the ending For instance, *cip-ulo* would have been one word in any of our data sets, but in English, it would translate to "to the house" Thus, *ulo* was tokenized from the root word Second, if a particle attached to the end of a word had no English equivalent, then we would tokenize it For instance, *ta* follows many verbs and acts as a sentence marker However, there is no word that is an equivalent in the English language Thus, we

decided this particle would be tokenized Finally, we usually tokenized only particles such as postpositions, which are typically closed class items. The reason is that closed class items are bounded in number and their occurring positions are highly predictable Open class items such as nouns are not bounded in number and their occurrences are not regular, and therefore tokenization of open class items will result in a lot more conditions and side effects than tokenization of closed class items These rules helped us determine which words were to be tokenized, and which words were not

3.2 The System Architecture of the Korean Language Understanding Module

The Korean Language understanding module starts with a romanization algorithm In performing our translation, our first stage is pre-processing of the data, so that our translation system can perform the necessary tasks While it would be advantageous to our translation system if we had a programming language that would recognize Korean characters, no such programs currently exist Thus, having an effective romanizer has been important My thesis supervisor provided an excellent romanizer that has performed flawlessly While many different methods of romanization exist, as long as one stays consistent, there seems to be no real problem in choosing any method over another In addition, another module in the pre-processor stripped the text of any punctuations and parenthesis, as they were deemed unnecessary for the translation

The domains that we trained our morphological analyzer on were provided to us in the original Hangul (Korean characters) Our morphological analyzer takes romanized

Hangul, and splits up words so that the basic word and the attached morpheme, or "token", can be separated A specific part of the morphological analyzer, the tokenizer, performs this important task. This output is then sent to the next stage, the grammatical analysis module The grammatical analysis portion of the Korean understanding module takes the pre-processed, tokenized sentences, and outputs an interlingua representation, called the semantic frame Currently the Korean grammar module of the Korean-to-English translation system is capable of handling all of the Military Communications Messages on which my tokenizer was trained, and is defined in terms of words Since the grammar, the rules of which are defined in terms of words, is not easily generalizable to new types of input, we need to find a way of generalizing the grammar rules to cover sentences on which the system has not been trained The Korean part-of-speech tagger I discuss in detail in Chapter 4 will become the basis for the grammar generalization by enabling the system to take part-of-speech sequence, rather than the word sequence, as input to the parser. Regarding the improved parsing coverage by adopting part-of-speech tagging technique in the English analysis module of the English-to-Korean subsystem of CCLINC, refer to [Weinstein et al 1997] and [Lee et al 1997]

To output English, we have an English language generator, that takes interlingua, which we call semantic frame/common coalition language in our system, as the input From there, it generates the equivalent statement in English This generator has already been extensively developed for the English to Korean translation work While the development of an English generator for English to Korean translation might not seem obvious, the English generator was used to verify the accuracy of the translation -- a para-phrased version of the meaning representation was translated back to English for the benefit of the developers who did not understand Korean. This same generator will be used in our trans-

lation system Figure 3-1 shows the process flow of the Korean-to-English system of CCLINC

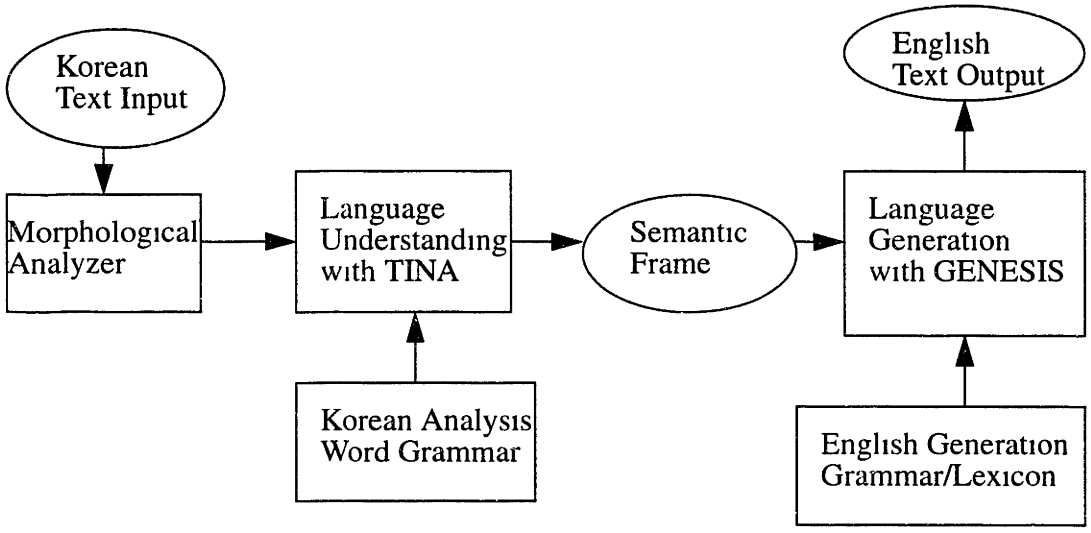

Figure 3-1: Process Flow of English-to-Korean Translation

3.3 The Military Communications Messages Data (ARMYCOM Data)

The data sets we used for our empirical research came from various sources The first was called the "Military Communications Messages" It consisted of approximately 122 sentences, which were broken up into 5 data sets of about 20 sentences each We ran our tokenizer on a data set, analyzed the output, made the necessary corrections to the tokenizer, then proceded to analyze the next data set While we hoped for 100% accuracy in our tokenization, we achieved at least 95% accuracy in all our data sets by the time we completed our modifications to the tokenizer Accuracy was measured by counting the total number of words that required tokenization and then counting the number that were mistokenized. Errors generally fell into two categories The first category consisted of situations

where the tokenizer failed to recognize that a particle should be tokenized, and would not take any action The second category consisted of situations where the tokenizer tokenized letters that were not supposed to be tokenized By adding up the errors, and dividing by the total number of words that required tokenization, we had a measurement for accuracy After the entire first data set was analyzed, we took data from the Combat Briefer's Course Manual, a set that contained approximately 192 sentences. This entire data set was run through our now rather robust tokenizer, and the output analyzed This was followed by the MUC-II data set. The analysis of the first data set proved to be an interesting challenge that was extremely fruitful. The first data set gave us a chance to develop the rules that behind the development of the tokenizer In addition, it gave us a basic framework that we had a chance to build upon Finally, our first motivations for multiple tokenization came from the words in this data set.

This first data set was broken up into 5 subsets of approximately 22 sentences each Each subset was inputted to the tokenizer, and the output was then analyzed Our method of development was excruciatingly simple The theoretical aspect consisted of researching through a Korean textbook, "Korean for International Learners" [Ihm, Hong, Chang, 1988] Noting all the morphemes that could arise, we added them into the conditions for the tokenizer The empirical method consisted of training the tokenizer on piece of new data Statistics were taken for the outputted data, and the tokenizer would be trained on the data so that the accuracy rate would be much higher Then, new data would be inputted into the tokenizer The same procedure was performed on the next subset, and continued until the final subset was done Modifying the tokenizer was a simple issue, a copy of the code (Appendix I) shows that it compares the end of each word to different tokens If a match is found, it splits the word from the token Modifying the tokenizer was basically an

issue of adding if-then statements. The only major modification that was necessary was adding multiple tokenization The need for this become apparent soon after the first subset was analyzed

Our tokenizer took each word and checked it once to see if a token was attached to the end If it found a matching token, it would add a space between the basic word and the token at the end If it did not find a match, it would do nothing to the word In any case, our tokenizer would go onto the next word, regardless of whether it had found a match or not However, Korean words can have several layers of tokens just like English (e g un-precedent-ed), and we decided that positing three levels of tokenization would be enough in designing a tokenizer which is both efficient and accurate Furthermore, in the three levels of tokens, certain particles can come only in certain positions To clarify this, I will present an example "pudaytulloputheey" has these three levels of tokens "puday" is a word meaning "unit", "tul" is the plural marker, "loputhe" is a postpositional particle, and "ey" is the possessive marker The original tokenizer would only recognize the "ey" and would give an output of "pudaytulloputhe ey" whereas we desired an output of "puday tul loputhe ey" Of course, an if-then statement could have been inserted to recognize this particular combination of tokens, but using that method, we would have had to insert if-then statements to account for all such combinations, which would have been too numerous to include Instead, we opted to do multiple levels of tokenization

At first glance, multiple tokenization seemed to be a trivial matter The tokenization conditions were separated into three different groups, each corresponding to the tokenization level upon which it would work The first group would tokenize the outermost token The second would do the middle one, and the third routine would do the innermost How-

ever, when this algorithm was implemented, several memory bugs occurred, and caused a segmentation fault whenever the tokenizer was run Tracking down these elusive bugs took about two weeks In the meantime, the tokenizer was inefficiently implemented by using a script that would call three separate if-then statement routines, more conditions were added or removed as necessary to each stage of tokenization

The actual analysis of the tokenizer output and modifications was a long and, at times, tedious process. I was given a seed program by my supervisor, which I proceeded to modify This seed program was about 150 lines long Running the seed program on the first subset garnered a 52% accuracy By modifying the tokenizer to deal with the improperly tokenized words, we produced an output that was 95% accurate The other 5% lied in the fact that we did not yet possess the ability to do multiple levels of tokenization While we worked on the multiple stages of tokenization, we ignored this 5% error, and corrected everything else that needed to be corrected

Proceeding onto the next data sets, we encountered nothing serious enough to dictate that we modify the entire tokenizer program as we did for multiple tokenization. The task merely including appending conditions, and dealing with other, relatively minor, problems The second data subset gave us a 57% tokenization accuracy when we applied our tokenizer that was optimized for the first subset By appending more conditions to the tokenizer, the accuracy on the second data subset was raised to 95%

The third data subset produced a couple of new problems. The tokenizer that was optimized for both data subsets 1 and 2 produced a 71% accuracy when run on the third data subset However, after extensive modifications, we could only reach 90% accuracy The

main reason for this problem lied in the problem of exceptions and ambiguities, as noted previously. As such, we began to list every single exceptions and tried to find a pattern to each (i.e , were there specific vowels before certain exceptions? Would a consonant before a suspect particle give clues to its status?) In most cases, no such patterns were discovered, and individual exceptions were put into the conditions of the tokenizer

We then moved onto data subset four, where we discovered that the tokenizer optimized for the previous three subsets produced an accuracy of 84% No new problems were discovered in this set, but the need for multiple tokenization and exception/ambiguity resolution was seen even more clearly in this set By modifying the tokenizer once more, we reached a 93% accuracy Finally, the last data subset was perhaps the most heartening. The tokenizer needed almost no modifications, as it gave a 92% accuracy rate on the first pass The fifth data subset merely strengthened the need for the multiple tokenization scheme as well as, again, the problems with ambiguities, since most of the errors arose from these two causes In table 3-1, we summarize our results.

Data Set	Challenges	Untrained/Trained Accuracy
ARMYCOM Subset#1	----------	52%/95%
ARMYCOM Subset #2	----------	57%/95%
ARMYCOM Subset #3	Exceptions, Ambiguities	71%/90%
ARMYCOM Subset #4	Multiple Tokens Issue	84%/95%
ARMYCOM Subset #5	Same as #3 and #4	92%/93%

Table 3-1: Summary of ARMYCOM data

With the fifth data subset complete, we returned to the entire data set and re-ran the tokenizer. We wanted to insure that the added conditions did not have any unwanted side effects on the previous data sets When we performed this experiment, we learned that our tokenizer was overzealous, at times, with tokenizations While it still did not properly perform multiple tokenization, the tokenizer would split up words that had no need to be split up Again, we looked for patterns that could be found, but there really were none, and we merely included exceptions for all the common words that were being mistokenized

With this complete, we proceeded to work full-speed on the multiple tokenization issue As described above, it was a frustrating process that was hampered by memory bugs in the code At first glance, the multiple tokenization system seemed to be simple Instead of running the tokenizer just once through the entire data set, we made three separate subroutines, tokenize1, tokenize2, and tokenize3, that contained morphemes that could only be in certain positions The main program in our tokenizer merely called each tokenize routine sequentially. Some memory mismanagement problems made this stage of development difficult However, once multiple tokenization was perfect, we proceeded onto the next big data set.

3.4. The Combat Briefer's Course Manual Data Set

While the ARMYCOM domain was broken up into 5 separate sets and then analyzed, it was decided that the tokenizer was developed enough to attack the second data set without the need for breaking up the data set This second set, taken from the Combat Briefer's Course Manual, had 192 sentences to be tokenized Our developmental methods were the

same as for the Military Communications Messages. However, we had already done a fairly thorough job of implementing new morphemes through our theoretical training methods. Thus, our focus on the development of the tokenizer for this stage was based mainly on the same empirical method as we used in the ARMYCOM data set run the tokenizer on the data set, take statistics, train the tokenizer (i e , add more conditions), and run it on the same data set once more and garner trained-data statistics

Our first pass on this data set, which was also our first experience with multiple tokenization, was a heartening experience. The statistics were excellent, our tokenization rate was approximately 90% on the untrained data Most of the 10% of the failures were accounted for by repeat failures on three words, *ttala, neyci* and *sikan* None of these words should have been tokenized, but because of their endings, they were. Multiple tokenization was the key factor in improving the tokenization accuracy, without which would have been at least 10% lower Instances of multiple tokenization included, *kuyss I ta, seypyek 5 si ey* Once we trained the tokenizer on this new data, we reached an accuracy level of approximately 95% The remaining 5% resulted from certain parts of the data that were not taken into account during training, and were inadvertently ignored In addition, some of the errors resulted from the increased number of ambiguities developing from the double tokenization scheme Improving this statistic was our primary goal in tokenizer development

3.5 The Numeric Tokenization Subroutine

As we trained the tokenizer on the COBC domain, we realized that we required an algorithm to separate numbers from words For instance, 177*puday* means "177th Unit" This example alone does not indicate a problem, but when we consider that the lexicon will have to have every single instance of <number> + *puday* in order to properly translate a given text, it is immediately obvious that tokenizing numbers from words is a necessity. Thus, we proceeded to develop a "numeric" tokenizer

Our needs for the numeric tokenization algorithm was different from that of the regular tokenization algorithm Whereas the original tokenization algorithm merely checked to see if the last few characters matched a pattern, the numeric tokenizer needed to check every single character in a word to determine if that character was a number. Thus, a whole new algorithm needed to be written

The development of this algorithm was straightforward, we wrote code that cycled through every character in a word, and when it detected a number followed by an alphabetic character, or an alphabetic character followed by a number, a space was inserted between the two characters In addition, we had to determine at which stage of the tokenization algorithm that this new routine should be placed Should it go before, in between, or after tokenize1, tokenize2, and tokenize3? In the end, we decided that the exact placement did not really matter as the tokenize* routines and this numeric routine did different things, and would not interfere with each other. Thus, I placed the numeric routine before the tokenize routines solely to make editing the numeric routine easier

I checked that the numeric routine was performing properly by first, creating my own data set of various random numbers and characters that were strewn together Next, I ran

this routine on the COBC and ARMYCOM domains I encountered no problems with any of the domains that I tested this routine on Satisfied with the results, I decided to continue onto the next domain

3.6 The Naval Operations Message Domain (MUC-II Data)

We decided to attempt to train the tokenizer on a final data set For this task, we chose the MUC-II domain The MUC-II domain is unique in that the sentences are highly telegraphic The sentences in the two previous domains were grammatically complete The MUC-II domain, however, was more concise, and this domain had sentences in which particles that were not completely necessary were dropped For instance, this sentence (from the ARMYCOM data set -- in the tokenizer's romanization convention)

ChoiGeun Eui JiHuiGoan BoGo Ga PilYoHa Da (The most recent report is needed)

would have been written in the MUC-II data set as

ChoiGeun JiHuiGoan Bogo PilYoHam

Thus, all the tokens that are not absolutely necessary are dropped in the MUC-II data set This presented a new style of writing that our tokenizer had not dealt with before, and we encountered a couple of major challenges In addition, this domain was much larger than previous domains It was over 300 sentences long, and provided a plethora of data on which to train our tokenizer

First, the tokenizer was not prepared to deal with the endings that did not have the regular sentence markers, such as *ta*, *ka*, *la*, etc. Instead, the MUC-II data set had sentences that would end in a nominalized form However, there were patterns such as, *toim*, *ham*, and *um* that recurred frequently. Thus, we implemented this change into our tokenizer

In addition, the tokenizer was too general when it dealt with the morpheme *ki* Whereas the previous domains had limited instances of this morpheme, the MUC-II domain had many words that had this morpheme at the end Unfortunately, while *ki* can be a postposition, it can also mean "ship" when attached to the end of a word As discussed earlier, we did not want to tokenize two nouns, only particles Thus, this tokenization of *ki* went against our goals, and was considered an error As the MUC-II domain had many instances of this ambiguity, our tokenizer's initial performance was lackluster at best

We determined that the *ki* that served as a postposition usually after the morpheme *ha*, which comes at the end of most verbs. With this in mind, we made a simple change to our tokenizer, such that *ki* would be tokenized only after this morpheme, this solution resulted in a dramatic improvement in the accuracy of the tokenizer whereas prior to this change we netted a 72% accuracy rate, this modification alone netted us a 83% accuracy rate Otherwise, this domain was relatively straightforward and simple to train the tokenizer on. And it has given us a more robust tokenizer prepared to deal with yet another style of writing that could be encountered in Korean to English translation

3.7. Major Technical Challenges and Integration of the Tokenizer into the Korean-to-English Subsystem of CCLINC

The development led to very few technical challenges. The only major difficulty lied in the fact that our original seed program analyzed the data only once Unfortunately, Korean does not exclusively have single morphemes at the end of a sentence Korean suffixes come in many categories Subject markers, object marker, postpositions are all examples of these suffixes However, for our purposes, we came up with three types of categories, these categories can be determined from there respective position at the end of a word For instance, a suffix in category 1 can only come right after the base word A suffix in category 2 can come right after the word, or after a suffix in category 1 Finally, a category 3 suffix can either come right after the word, right after a category 1 suffix, or right after a category 3 suffix

Our original tokenizer could recognize each suffix, but if a word contained more than one category of suffixes, it would only recognize the outermost one, separate it from the word, and continue onto the next word For example, a word like, *pwudaytulloputhe* needs to be tokenized as: *pwuday tul loputhe* (meaning, "from the units") Our original tokenizer would only recognize the *loputhe*, and split only that off from the basic word, *pwuday* ("unit") However, if the original was *pwudaytul*, it could recognize that *tul* (plural marker) was a token to be split off from the word, and would do it flawlessly. Our first proposal was to attempt to implement all such combinations, it seemed as if only a few suffixes came in combinations However, as work on the ARMYCOM data continued, it was apparent that there were too many different combinations to implement them all into our if-then statements A need for a multiple-stage tokenization procedure was then established

Our if-then statements in our original tokenizer were then broken up into the three categories that we discussed. Each were placed in their own subroutines, with category 1 suffixes being in "tokenize1", and so forth The main program was modified to call each of these tokenize subroutines in order While seemingly a simple matter, our simple seed program had grown to over 700 lines of code. Wading through this amount of code was a new experience for me, and took longer than needed. Table 3-2 lists the categories and the tokens that comprise them Again, this is in the romanization convention of the tokenizer, instead of the Yale convention .

Tokenize Subroutine No	Particles
tokenize1 (outermost tokens)	Myeon, InGa, nGa, Reul, Neun, GeoNa, Myeo, Doim, Eum, Eui, Ga, Eun, RaDo, EuNa, nHan, Eul, I, Goa, Jim, Ham, Im, EuNa, Do, Go
tokenize2 (middle tokens)	EuRoBuTeo, HanTeSeo, RoBuTeo, EGeSeo, NeunDe, DalRə, GgaJi, HanTe, BuTeo, GaJi, Jung, EuRo, MaJu, MaDa, BoDa, ESeo, EGe, Goa, Da, Ge, Ga, E, Ro, Oa, De, Ni
tokenize3 (innermost tokens)	BeonJjai, Nyeon, Neun, DoRog, JuGi, Ueol, Beon, iYa, Geub, Ya, Deul, Seo, Ji, Yeo, Bun, IEox, Eox, Ix, Go, Il, Si, I, Eo

Table 3-2: Stages of Multiple Tokenization

Since our tokenizer was well-developed, we decided to integrate it into the larger translation system The coding was a relatively simple issue once the segmentation faults

were dealt with The small tokenizer built into the translation system was a single subroutine, its coding was identical to each of the subroutines tokenize1, tokenize2, and tokenize3. To integrate the new multi-stage tokenizer, I replaced the this subroutine with one that called each tokenize subroutine in order The tokenizer was easily introduced to the entire translation system

3.8. Remaining Problems

Our tokenizer needs further development in many ways There are several ambiguities that have still not been resolved As of now, we have exceptions in our if-then conditions for specific words such as these However, in many cases, we have three or four exceptions that are in a given if-then statement It is likely that the number of exceptions will grow A good method of finding a general rule to these exceptions needs to be found to ensure consistent tokenization In addition, as was evidenced by the MUC-II domain, new exceptions can show up at any time

In addition, there are many cases in which a decision needs to be made whether tokenizing a given word is necessary For instance, when we dealt with the morpheme *ki*, we had to decide whether there would be a great advantage to tokenize this morpheme in all cases. Indeed, there are several examples where *ki* could mean "ship" when attached to a different noun In the end, we decided this morpheme would be tokenized only when it was used as a postposition This decision was based on the fact that there were not enough different nouns that *ki* could follow to necessitate adding this complexity to the tokenizer However, in other cases, the decision is more difficult, and requires more thought. This is

another problem that needs to be dealt with as we analyze more data sets, in a case-by-case basis

These problems are paramount in our future work on the tokenizer In addition, we must ensure that any new tokens discovered in future analyses will be included into the tokenizer By running the tokenizer on new data sets and training the tokenizer to work properly on these data sets, I believe we can have a working tokenizer that will be highly accurate, and will be an effective part of our translation system

Chapter 4

An Application of the Rule-Based Part-of-Speech Tagger to the Korean Language

4.1 Background

The English to Korean translation system encountered difficulties when it relied on domain specific grammar rules As vocabulary items were a part of the grammar rules, if the translation system encountered a new word that had not been seen before, parsing would fail [Weinstein et al , 1997] In order to handle this problem, the group decided to implement a two stage parsing scheme, where the first stage would continue to use domain-specific grammar rules, if this failed, the system would go to the second stage, and replace the unknown with its part of speech. By using generalized grammar rules, the parsing could be completed

Currently, the Korean analysis grammar is defined in terms of words For instance, the word for "ship" will be recognized by the grammar rule which is rewritten by the word for "ship" itself Unfortunately, if the word for "yacht" is used in the input sentence, and there is no grammar rule which is rewritten as the word for "yacht", the system cannot parse the sentence containing the word for "yacht" However, the part-of-speech tagging technique used for the English-to-Korean translation system [Weinstein et al 1997] and [Lee et al 1997] can be applied to the Korean understanding module Instead of defining the grammar rules in terms of specific words, the grammar rules will be defined in terms of part-of-

speech, making the translation system much more robust, as well as making the development of grammar much more efficient

In this same spirit, we decided to adapt part-of-speech tagging to the Korean to English translation system We used the same rule-based tagger developed by E Brill [Brill, 1992] that was used for the English to Korean translation system This tagger is versatile in that it is not language specific The tagger can be trained on any language in the same manner The tagger operates in two stages First, the words are tagged as if they were standing alone In other words, the most likely tag is assigned to the word Next, the tagger uses contextual clues to improve the accuracy of the tags.

The training of the system operates in two stages as well First, rules are learned to predict the most likely tag for an unknown word For instance, in English, if a word ends in "ed", then it is most likely a past tense verb In some cases, one tag will suffice for a word, no matter what the context, and the word will always be tagged in that manner Next, rules are learned to use contextual clues to improve the tagging accuracy For instance, in English, a word changes from a verb to a noun if the preceding word is a determiner The details of training the tagger are similar to this outline A large, manually tagged corpus, is assumed to exist Given this, the corpus is split up into two sets· the first set is used to learn rules to predict the most likely tag for an unknown word. The second set is used to develop rules based on contextual clues. Training scripts that are available along with the tagger are used to train the tagger

4.2 Training the Rule Based Tagger on the Korean Language

We decided to use the ARMYCOM data set to train the tagger for the Korean language In order to do so, we had to produce a set of tags that were Korean specific For example, the English trained tagger had sets of tags such as nouns (NN), verbs (VBZ), and adjectives (JJ) that would readily transfer to Korean However, there were many parts of speech in the Korean language that had no English equivalent Thus, new tags needed to be developed for these words and particles. This was, perhaps, the most difficult part of developing the Korean Rule Based Part-of-Speech Tagger To come up with these new tags, we consulted "Korean Grammar for International Learners", which has served as our reference text throughout our research Table 4-1 lists the tags we applied to the various particles

Tags	Description
JJ	Adjective
RB	Adverb
NN	Noun
FW	Foreign Word
SYM	Symbol
VB	Verb, base form
VBD	Verb, past tense
VBN	Verb, gerund
VBZ	Verb, past participle
VBF	Verb, present tense
SCM	Subordinate Clause Marker
PM	Plural Marker
CM	Case Marker

Table 4-1: List of Tags and Their Description

Tags	Description
SM	Sentence Marker
PPM	Postpositional Marker
DPN	Demonstrative Pronouns
PRP	Personal Pronouns
TM	Tense Marker

Table 4-1: List of Tags and Their Description

With this, we manually tagged the ARMYCOM corpus We ran the scripts and routines that created the lexicon and contextual rules However, we did not have a chance to test and train these rules on a new corpus Below are samples of our tagged corpus (again, we are using the translation system's romanization, as this is actual text used in our system)

1/CD Geub/JJ JoHang/NN 103/CD JeonUiJiUeonDaiDai/NN E/PPM GoanHan/PPM SangHoang/NN YoCheong/NN

{Speedy}/FW MoDeun/JJ YeoDan/NN Eui/CM (JiHuiGoan/NN BoGo/NN | TongSin/NN EungDab/NN) Eul/CM EodGo/VB Sip/VB Da/SM

BoGo/NN Eul/CM DaSi/RB Hai/VB Dal/VB Ra/SM

JiHuiGoan/NN BoGo/NN Ga/PPM IbSuDoin/JJ YuIlHan/JJ BuDaiNN Eun/CM 149/CD I/VB Da/SM

Future work for the tagger should include testing it on the COBC data set and then training it on the same corpus. I believe the COBC data set is the next logical step as it is similar to the ARMYCOM corpus and will prove to be a decent challenge for the tagger Once this is complete, the MUC-II data should be used to train the tagger. I believe this will be a greater challenge, as its very nature is different from the other two sets

Chapter 5

Future Work and Conclusions

5.1 Future Plans

To this point, we have available a pre-processor that romanizes Korean characters and takes out all the unnecessary punctuations to facilitate the translation analysis. We have also developed a fairly robust tokenizer that has been proven to work on two data sets that total about 300 words Our future work begins with continuing the development of the tokenizer. As has been demonstrated by the MUC-II data set, training on over 300 sentences did not provide enough training to avoid common hazards I believe, however, that an extremely robust tokenizer can be developed by continuing to train the tokenizer on new domains

This tokenizer will then be used as part of the greater Korean language understanding module. We have already modified it and inserted the tokenizer to run with the TINA/ GENESIS translation system Thus future goals would be to continue training the tokenization algorithm The main thrust of the development should be to find patterns in exceptions, such that the tokenizer would be less dependent on individual exceptions In addition, the tokenizer needs to be re-evaluated on the ARMYCOM domain, in order to determine what, if any, side effects are present from its constant evolution. While the tokenizer has been re-evaluated on the COBC domain, no such experiment has been done on the ARMYCOM domain The previous domains, thus, could always serve as a check, to determine if any negative side effects were introduce by new conditions.

In addition, the part of speech tagger needs to be evaluated on a new domain, and then trained. I believe that new tags will be necessary as training continues, and old tags may need to be re-evaluated As such, I believe a correctly functioning part of speech tagger will be an important part of the entire translation system

5.2 Conclusion

The Speech-to-Speech Translation System has demonstrated that it is possible to translate military text from English to Korean The versatility of the SSTS lies in the fact that it uses InterLingua Text (ILT) to perform these translation By developing a robust Korean understanding module, it is possible to translate Korean text into English with a fairly high degree of accuracy. As of now, we have developed a fairly robust morphological analyzer While it is not yet perfect, I believe that continual improvements will eventually result in a tokenizer that can be over 90% accurate on any new domain The part of speech tagger will require far more work Continued training of the tagger is necessary for a robust system

While machine translation is a challenging area, these routines should assist in developing an effective Korean to English translation system Though translation across a general domain is difficult, I believe this translation system will be of great assistance to translations in the military domain

Appendix A

The Tokenization Algorithm

Included below is the entire code for the tokenization algorithm

```
/*
 * (c) Copyright 1996-1997 Massachusetts Institute of Technology Lincoln
 * Laboratory Speech Systems Technology Group All Rights Reserved
 */

/* Prototype Korean Tokenization Program */

/* Seed program written by Young-Suk Lee, 9/22/97 */

/* Modified extensively by Philip D. Kim 9/97-5/98 */

#include <stdlib h>
#include <stdio.h>
#include <string h>

#define LINE_LENGTH 1000

int
string_to_list_of_tokens(char *string, char **array, int entry_point,
 int maxn, char *delimiters),
char        *tokenize(char *sentence);
char        *tokenize2(char *sentence);
char        *tokenize3(char *sentence);
char        *numeric(char *sentence),

void
main(int argc, char *argv[])
{
int         i, nsentences,
char         Line[LINE_LENGTH],
FILE         *ifp, *ofp,
```

```
char        *newLine,
char        *output1;
char        *output2;
char        *numout,

ifp = fopen(argv[1], "r");
if (!ifp) {
fprintf(stderr, "No input file %s\n", argv[1]);
return;
}
for (i = 0, (fgets(Line, LINE_LENGTH, ifp) != NULL); i++);
fclose(ifp),
nsentences = i,

ifp = fopen(argv[1], "r"),
ofp = fopen(argv[2], "w");
if (!ofp) {
fprintf(stderr, "No output file %s\n", argv[2]),
return,
}
while (fgets(Line, LINE_LENGTH, ifp) != NULL)
  {

    numout = numeric(Line),

    output1 = tokenize(numout);

    output2 = tokenize2(output1),

    newLine = tokenize3(output2),
    i++,
    if (i == nsentences)
    break,
    fprintf(ofp, "%s\n", newLine),
  }
return,
}


int
```

```
string_to_list_of_tokens(char *string, char **array, int entry_point,
 int maxn, char *delimiters)
{
int          i,
char         *string_copy,
char         *token,
string_copy = strdup(string),
token = strtok(string_copy, delimiters);
if (token == NULL) {
array[entry_point] = NULL,
free(string_copy),
return (0);
}
array[entry_point] = strdup(token),
for (i = entry_point + 1, ((token = strtok(NULL, delimiters)) != NULL), i++) {
if (i >= maxn) {
fprintf(stderr, "string_to_list_of_tokens  array too small! \n");
break,
}
array[i] = strdup(token),
}
free(string_copy),
return (i - entry_point),
}


char *numeric(char *sentence)
{
  char preceding, detect, current, next;
  int printdetect, pdkdetect,
  char *tokenized,
  int ichar = 0, ochar = 0,
  int sentlen,

  sentlen = strlen(sentence),
  tokenized = calloc(strlen(sentence)+100, sizeof(char)),

  preceding = sentence[ichar++],
  if(ichar < sentlen) current = sentence[ichar++],
  tokenized[ochar++] = preceding;
```

```
 while(ichar < sentlen && (next = sentence[ichar++]) != '\0')
   {
    if((preceding >= '0' && preceding <= '9') &&
((current >= 'A' && current <= 'Z') ||
(current >= 'a' && current <= 'z')))
{
 detect = ' ', printdetect = 1,
}
    else printdetect = 0,
    if(((preceding >= 'a' && preceding <= 'z') ||
 (preceding >= 'A' && preceding <= 'Z')) &&
(current >= '0' && current <= '9'))
{
 detect = ' ', pdkdetect = 1,
}
    else pdkdetect = 0,

    if ((printdetect) || (pdkdetect))
{
 tokenized[ochar++] = detect,
 tokenized[ochar++] = current,
}
    else tokenized[ochar++] = current,

    preceding = current,
    current = next,
   }
 sprintf(&tokenized[ochar], "%c", next),
 /* printf("%s\n", tokenized), */
 return(tokenized),
}


char        *
tokenize(char *sentence)
{
char        *romanHangul, *stringCopy,
char        *words[100],
char        *word, *stem, newSentence[1000],
```

```
int         nwords, i, wordLength, sentLength;

sentLength = strlen(sentence) + 1,
stringCopy = (char *) calloc(sentLength + 3, sizeof(char)),
strcpy(stringCopy, sentence),
/* stringCopy = ks2sshr(sentence), */

newSentence[0] = '\0';
nwords = string_to_list_of_tokens(stringCopy, words, 0, 100, " \n"),
for (i = 0, i < nwords, i++) {
word = words[i],
wordLength = strlen(word);

if ((wordLength > 6) && !strcmp(&word[wordLength - 5], "Myeon")) {
word[wordLength - 5] = '\0',
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "Myeon"),
strcat(newSentence, " "),
} else if ((wordLength > 6) && !strcmp(&word[wordLength - 4], "InGa")) {
word[wordLength - 4] = '\0';
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "In"),
strcat(newSentence, " "),
strcat(newSentence, "Ga"),
strcat(newSentence, " "),
} else if ((wordLength > 5) &&
  !strcmp(&word[wordLength - 3], "nGa")
  && strcmp(word, "NuGunGa")) {
word[wordLength - 2] = '\0',
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "Ga"),
strcat(newSentence, " "),
} else if ((wordLength > 5) &&
  !strcmp(&word[wordLength - 4], "Reul")) {
word[wordLength - 4] = '\0',
strcat(newSentence, word),
strcat(newSentence, " "),
```

```
strcat(newSentence, "Eul"),
strcat(newSentence, " ");
} else if ((wordLength > 5) &&
  !strcmp(&word[wordLength - 4], "Neun")) {
word[wordLength - 4] = '\0';
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "Eun");
strcat(newSentence, " "),
}else if ((wordLength > 5) &&
  !strcmp(&word[wordLength - 5], "GeoNa")) {
word[wordLength - 5] = '\0';
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "GeoNa"),
strcat(newSentence, " "),
}
else if ((wordLength > 5) && !strcmp(&word[wordLength - 4], "Myeo")) {
word[wordLength - 4] = '\0',
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "Myeo"),
strcat(newSentence, " "),
}else if ((wordLength > 4) && !strcmp(&word[wordLength - 4], "Doim")) {
word[wordLength - 4] = '\0',
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "Doim"),
strcat(newSentence, " "),
}
else if ((wordLength > 4) &&
  !strcmp(&word[wordLength - 3], "Eum") &&
  strcmp(word, "DaEum")) {
word[wordLength - 3] = '\0',
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "Eum"),
strcat(newSentence, " "),
} else if ((wordLength > 4) && !strcmp(&word[wordLength - 3], "Eui")) {
word[wordLength - 3] = '\0',
```

```
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "Eui"),
strcat(newSentence, " ");
} else if ((wordLength > 3) &&
  (!strcmp(&word[wordLength - 3], "aGa") ||
   !strcmp(&word[wordLength - 3], "eGa") ||
   !strcmp(&word[wordLength - 3], "iGa") ||
   !strcmp(&word[wordLength - 3], "oGa") ||
   !strcmp(&word[wordLength - 3], "uGa"))) {
word[wordLength - 2] = '\0',
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "Ga"),
strcat(newSentence, " ");
} else if ((wordLength > 4) &&
  (!strcmp(&word[wordLength - 4], "bEun") ||
   !strcmp(&word[wordLength - 4], "dEun") ||
   !strcmp(&word[wordLength - 4], "gEun") ||
   !strcmp(&word[wordLength - 4], "hEun") ||
   !strcmp(&word[wordLength - 4], "mEun") ||
   !strcmp(&word[wordLength - 4], "nEun") ||
   !strcmp(&word[wordLength - 4], "sEun") ||
   !strcmp(&word[wordLength - 4], "lEun"))
  && (strcmp(word, "HogEun"))) {
word[wordLength - 3] = '\0',
strcat(newSentence, word).
strcat(newSentence, " ");
strcat(newSentence, "Eun");
strcat(newSentence, " "),
} else if ((wordLength > 4) && !strcmp(&word[wordLength - 4], "RaDo")) {
word[wordLength - 4] = '\0',
strcat(newSentence, word),
strcat(newSentence, " ");
strcat(newSentence, "RaDo");
strcat(newSentence, " "),
} else if ((wordLength > 4) && !strcmp(&word[wordLength - 4], "EuNa")) {
word[wordLength - 4] = '\0',
strcat(newSentence, word),
strcat(newSentence, " "),
```

```
strcat(newSentence, "EuNa");
strcat(newSentence, " ");
} /* Maybe we don't even need this condition */
    /*
    else if ((wordLength > 4) && !strcmp(&word[wordLength - 4], "Doin")
  && strcmp(word, "JeobSuDoin")) {
word[wordLength - 4] = '\0';
strcat(newSentence, word),
strcat(newSentence, " ");
strcat(newSentence, "Doin"),
strcat(newSentence, " ");
} */
else if ((wordLength > 4) && !strcmp(&word[wordLength - 4], "nHan")
  && strcmp(word, "GoanHan")) {
word[wordLength - 3] = '\0',
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "Han"),
strcat(newSentence, " "),
} else if ((wordLength > 4) &&
  !strcmp(&word[wordLength - 3], "Eul")) {
word[wordLength - 3] = '\0';
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "Eul"),
strcat(newSentence, " "),
} else if ((wordLength > 3) &&
  (!strcmp(&word[wordLength - 2], "sI") ||
   !strcmp(&word[wordLength - 2], "bI") ||
   !strcmp(&word[wordLength - 2], "dI") ||
   !strcmp(&word[wordLength - 2], "gI") ||
   !strcmp(&word[wordLength - 2], "mI") ||
   !strcmp(&word[wordLength - 2], "nI") ||
   !strcmp(&word[wordLength - 2], "lI"))) {
word[wordLength - 1] = '\0',
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "Ga"),
strcat(newSentence, " ");
} else if ((wordLength > 3) &&
```

```
  !strcmp(&word[wordLength - 3], "Eun")
  && strcmp(word, "HogEun")) {
word[wordLength - 3] = '\0';
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "Eun"),
strcat(newSentence, " "),
} else if ((wordLength > 3) && !strcmp(&word[wordLength - 3], "Goa") &&
strcmp(word, "GyeolGoa")) {
word[wordLength - 3] = '\0',
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "Goa"),
strcat(newSentence, " "),
}else if ((wordLength > 3) && !strcmp(&word[wordLength - 3], "Jim")) {
word[wordLength - 3] = '\0',
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "Jim"),
strcat(newSentence, " "),
}else if ((wordLength > 3) && !strcmp(&word[wordLength - 3], "Ham")) {
word[wordLength - 3] = '\0',
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "Ham"),
strcat(newSentence, " "),
}else if ((wordLength > 2) && !strcmp(&word[wordLength - 2], "Im")) {
word[wordLength - 2] = '\0',
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "Im"),
strcat(newSentence, " ");
} else if ((wordLength > 2) &&
  strcmp(word, "GeuReoNa") &&
  !strcmp(&word[wordLength - 2], "Na")
  && strcmp(&word[wordLength - 4], "EuNa")) {
word[wordLength - 2] = '\0',
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "Na"),
```

```
strcat(newSentence, " "),
} else if ((wordLength > 2) &&
  strcmp(word, "GangDo") &&
  strcmp(word, "JeongDo") &&
  !strcmp(&word[wordLength - 2], "Do")) {
word[wordLength - 2] = '\0',
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "Do"),
strcat(newSentence, " "),
} else if ((wordLength > 2) && !strcmp(&word[wordLength - 2], "Go")
  && strcmp(word, "BoGo")
  && strcmp(word, "GeuRiGo")
  && strcmp(word, "ChamGo")
  && strcmp(word, "ChangGo")
  && ((wordLength > 8)
    && strcmp(&word[wordLength - 8], "GyeongGo"))) {
word[wordLength - 2] = '\0',
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "Go"),
strcat(newSentence, " "),
} else {
strcat(newSentence, word),
strcat(newSentence, " "),
}
}
free(stringCopy),
return (strdup(newSentence)),
}

char        *
tokenize2(char *sentence)
{
char        *romanHangul, *stringCopy,
char        *words[100],
char        *word, *stem, newSentence[1000],
int         nwords, i, wordLength, sentLength,

sentLength = strlen(sentence) + 1,
```

```
stringCopy = (char *) calloc(sentLength + 3, sizeof(char));
strcpy(stringCopy, sentence),
/* stringCopy = ks2sshr(sentence), */

newSentence[0] = '\0',
nwords = string_to_list_of_tokens(stringCopy, words, 0, 100, " \n"),
for (i = 0, i < nwords, i++) {
word = words[i],
wordLength = strlen(word);

if ((wordLength > 9) && !strcmp(&word[wordLength - 9], "EuRoBuTeo")) {
word[wordLength - 9] = '\0',
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "EuRoBuTeo");
strcat(newSentence, " "),
} else if ((wordLength > 8) && !strcmp(&word[wordLength - 8], "HanTeSeo")) {
word[wordLength - 8] = '\0',
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "HanTe");
strcat(newSentence, " "),
strcat(newSentence, "Seo"),
strcat(newSentence, " "),
} else if ((wordLength > 7) && !strcmp(&word[wordLength - 7], "RoBuTeo")) {
word[wordLength - 7] = '\0'.
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "RoBuTeo"),
strcat(newSentence, " "),
} else if ((wordLength > 6) && !strcmp(&word[wordLength - 6], "EGeSeo")) {
word[wordLength - 6] = '\0',
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "EGe"),
strcat(newSentence, " "),
strcat(newSentence, "Seo"),
strcat(newSentence, " "),
} else if ((wordLength > 6) && !strcmp(&word[wordLength - 6], "NeunDe")) {
word[wordLength - 6] = '\0',
```

```
strcat(newSentence, word),
strcat(newSentence, " ");
strcat(newSentence, "NeunDe");
strcat(newSentence, " ");
} else if ((wordLength > 5) && !strcmp(&word[wordLength - 5], "DalRa")) {
word[wordLength - 5] = '\0',
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "Dal"),
strcat(newSentence, " "),
strcat(newSentence, "Ra"),
strcat(newSentence, " "),
} else if ((wordLength > 5) && !strcmp(&word[wordLength - 5], "EoSeo")) {
word[wordLength - 5] = '\0',
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "EoSeo");
strcat(newSentence, " ");
} else if ((wordLength > 5) && !strcmp(&word[wordLength - 5], "GgaJi")) {
word[wordLength - 5] = '\0',
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "GgaJi"),
strcat(newSentence, " "),
} else if ((wordLength > 5) && !strcmp(&word[wordLength - 5], "HanTe")) {
word[wordLength - 5] = '\0',
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "HanTe");
strcat(newSentence, " "),
} else if ((wordLength > 5) && !strcmp(&word[wordLength - 5], "BuTeo")) {
word[wordLength - 5] = '\0',
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "BuTeo"),
strcat(newSentence, " "),
} else if ((wordLength > 4) && !strcmp(&word[wordLength - 4], "GaJi")) {
word[wordLength - 4] = '\0',
strcat(newSentence, word),
strcat(newSentence, " "),
```

```
strcat(newSentence, "GaJı");
strcat(newSentence, " "),
}else ıf ((wordLength > 4) && !strcmp(&word[wordLength - 4], "Jung") &&
strcmp(word, "GongJung")) {
word[wordLength - 4] = '\0',
strcat(newSentence, word);
strcat(newSentence, " "),
strcat(newSentence, "Jung");
strcat(newSentence, " "),
} else ıf ((wordLength > 4) &&
  !strcmp(&word[wordLength - 4], "EuRo")) {
word[wordLength - 4] = '\0';
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "EuRo"),
strcat(newSentence, " "),
} else ıf ((wordLength > 4) &&
  !strcmp(&word[wordLength - 4], "MaJu")) {
word[wordLength - 4] = '\0';
strcat(newSentence, word),
strcat(newSentence. " "),
strcat(newSentence, "MaJu"),
strcat(newSentence, " "),
} else ıf ((wordLength > 4) &&
  !strcmp(&word[wordLength - 4], "MaDa")) {
word[wordLength - 4] = '\0';
strcat(newSentence, word);
strcat(newSentence, " "),
sticat(newSentence, "MaDa"),
strcat(newSentence, " "),
} else ıf ((wordLength > 4) &&
  !strcmp(&word[wordLength - 4], "BoDa")) {
word[wordLength - 4] = '\0',
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "BoDa"),
strcat(newSentence, " "),
} else ıf ((wordLength > 4) && !strcmp(&word[wordLength - 4], "ESeo")) {
word[wordLength - 4] = '\0',
strcat(newSentence, word),
```

```
strcat(newSentence, " "),
strcat(newSentence, "ESeo");
strcat(newSentence, " "),
} else if ((wordLength > 3) && !strcmp(&word[wordLength - 2], "Ra") && strcmp(word,
"GeRilRa")) {
word[wordLength - 2] = '\0',
strcat(newSentence, word);
strcat(newSentence, " "),
strcat(newSentence, "Ra"),
strcat(newSentence, " "),
} else if ((wordLength > 3) && !strcmp(&word[wordLength - 3], "EGe")) {
word[wordLength - 3] = '\0';
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "EGe"),
strcat(newSentence, " ");
} else if ((wordLength > 3) && !strcmp(&word[wordLength - 3], "Goa")) {
word[wordLength - 3] = '\0';
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "Goa"),
strcat(newSentence, " ");
} else if ((wordLength > 3) && !strcmp(&word[wordLength - 2], "Da")) {
word[wordLength - 2] = '\0';
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "Da"),
strcat(newSentence, " "),
} else if ((wordLength > 3) && !strcmp(&word[wordLength - 2], "Ge") && strcmp(word,
"MosHaGe")) {
word[wordLength - 2] = '\0';
strcat(newSentence. word),
strcat(newSentence, " "),
strcat(newSentence, "Ge"),
strcat(newSentence, " "),
} else if ((wordLength > 2) &&
  !strcmp(&word[wordLength - 2], "Ga")
  && strcmp(word, "NuGunGa")
  && strcmp(word, "PyeongGa")) {
word[wordLength - 2] = '\0';
strcat(newSentence, word),
```

```
strcat(newSentence, " "),
strcat(newSentence, "Ga"),
strcat(newSentence, " ");
} else if ((wordLength > 2) &&
  !strcmp(&word[wordLength - 2], "Ni")
  && strcmp(word, "ANi")) {
word[wordLength - 2] = '\0',
strcat(newSentence, word),
strcat(newSentence, " ");
strcat(newSentence, "Ni");
strcat(newSentence, " "),
} else if ((wordLength > 2) && !strcmp(&word[wordLength - 1], "E")
  && strcmp(word, "DE")) {
word[wordLength - 1] = '\0',
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "E"),
strcat(newSentence. " "),
} else if ((wordLength > 2) &&
  !strcmp(&word[wordLength - 2], "Ro")
  && strcmp(word, "HoiRo")
  && strcmp(word, " DaiRo")
  && strcmp(word, " EuRo")) {
word[wordLength - 2] = '\0',
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "Ro"),
strcat(newSentence, " "),
} else if ((wordLength > 2) && !strcmp(&word[wordLength - 2], "Oa")
  && strcmp(word, "NaOa")) {
word[wordLength - 2] = '\0',
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "Oa"),
strcat(newSentence, " "),
} else if ((wordLength > 2) &&
  !strcmp(&word[wordLength - 2], "De")
  && strcmp(word, "NeunDe")) {
word[wordLength - 2] = '\0';
strcat(newSentence, word),
```

```
strcat(newSentence, " ");
strcat(newSentence, "De"),
strcat(newSentence, " "),

} else {
strcat(newSentence, word),
strcat(newSentence, " "),
}
}

free(strıngCopy),
return (strdup(newSentence)),
}

char        *
tokenıze3(char *sentence)
{
char        *romanHangul, *strıngCopy,
char        *words[100];
char        *word, *stem, newSentence[1000],
ınt         nwords, ı, wordLength, sentLength;

sentLength = strlen(sentence) + 1,
strıngCopy = (char *) calloc(sentLength + 3, sızeof(char)),
strcpy(strıngCopy, sentence),
/* strıngCopy = ks2sshr(sentence), */

newSentence[0] = '\0';
nwords = strıng_to_lıst_of_tokens(strıngCopy, words, 0, 100, " \n"),
for (ı = 0, ı < nwords, ı++) {
word = words[ı],
wordLength = strlen(word),


ıf ((wordLength > 8) &&
  !strcmp(&word[wordLength - 8], "BeonJjaı")) {
word[wordLength - 8] = '\0';
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "BeonJjaı"),
```

```
strcat(newSentence, " "),
} else if ((wordLength > 5) &&
   (!strcmp(&word[wordLength - 6], "0Nyeon") ||
    !strcmp(&word[wordLength - 6]. "1Nyeon") ||
    !strcmp(&word[wordLength - 6], "2Nyeon") ||
    !strcmp(&word[wordLength - 6], "3Nyeon") ||
    !strcmp(&word[wordLength - 6], "4Nyeon") ||
    !strcmp(&word[wordLength - 6], "5Nyeon") ||
    !strcmp(&word[wordLength - 6], "6Nyeon") ||
    !strcmp(&word[wordLength - 6], "7Nyeon") ||
    !strcmp(&word[wordLength - 6], "8Nyeon") ||
    !strcmp(&word[wordLength - 6], "9Nyeon"))) {
word[wordLength - 5] = '\0';
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "Nyeon"),
strcat(newSentence, " "),
} else if ((wordLength > 5) &&
  !strcmp(&word[wordLength - 4], "Neun")) {
word[wordLength - 4] = '\0';
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "Neun"),
strcat(newSentence, " "),
} else if ((wordLength > 5) &&
  !strcmp(&word[wordLength - 5], "DoRog")) {
word[wordLength - 5] = '\0',
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "DoRog"),
strcat(newSentence, " "),
} else if ((wordLength > 4) && !strcmp(&word[wordLength - 4], "JuGi")) {
word[wordLength - 4] = '\0';
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "Ju"),
strcat(newSentence, " "),
strcat(newSentence, "Gi"),
strcat(newSentence, " "),
} else if ((wordLength > 4) && (!strcmp(&word[wordLength - 5], "0Ueol") ||
```

```
  !strcmp(&word[wordLength - 5], "1Ueol") ||
  !strcmp(&word[wordLength - 5], "2Ueol") ||
  !strcmp(&word[wordLength - 5], "3Ueol") ||
  !strcmp(&word[wordLength - 5], "4Ueol") ||
  !strcmp(&word[wordLength - 5], "5Ueol") ||
  !strcmp(&word[wordLength - 5], "6Ueol") ||
  !strcmp(&word[wordLength - 5], "7Ueol") ||
  !strcmp(&word[wordLength - 5], "8Ueol") ||
 !strcmp(&word[wordLength - 5], "9Ueol"))) {
word[wordLength - 4] = '\0',
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "Ueol"),
strcat(newSentence, " "),
} else if ((wordLength > 4) && !strcmp(&word[wordLength - 4], "Beon")) {
word[wordLength - 4] = '\0',
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "Beon");
strcat(newSentence, " "),
} else if ((wordLength > 4) && !strcmp(&word[wordLength - 3], "iYa")) {
word[wordLength - 3] = '\0',
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "Ya"),
strcat(newSentence, " "),
} else if ((wordLength > 4) && !strcmp(&word[wordLength - 4], "Geub")) {
word[wordLength - 4] = '\0',
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "Geub"),
strcat(newSentence, " "),
} else if ((wordLength > 4) && !strcmp(&word[wordLength - 2], "Ya")) {
word[wordLength - 2] = '\0',
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "Ya"),
strcat(newSentence, " "),
} else if ((wordLength > 4) && !strcmp(&word[wordLength - 4], "Deul")) {
word[wordLength - 4] = '\0',
```

```
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "Deul");
strcat(newSentence, " ");
} else if ((wordLength > 4) &&
  !strcmp(&word[wordLength - 4], "HaGi") ||
  !strcmp(&word[wordLength - 4], "GaGi"))
 {
word[wordLength - 2] = '\0';
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "Gi"),
strcat(newSentence, " "),
} else if ((wordLength > 4) && !strcmp(&word[wordLength - 3], "Seo")) {
word[wordLength - 3] = '\0',
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "Seo"),
strcat(newSentence, " "),
} else if ((wordLength > 3) && !strcmp(&word[wordLength - 2], "Ji")
  && strcmp(word, "GgaJi")
  && strcmp(word, "GaJi")
  && strcmp(word, "GiJi")
  && strcmp(word, "TamJi")) {
word[wordLength - 2] = '\0',
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "Ji"),
strcat(newSentence, " "),
} else if ((wordLength > 3) && !strcmp(&word[wordLength - 3], "Yeo")
&& strcmp(word, "UiHaYeo")) {
word[wordLength - 3] = '\0',
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "Yeo"),
strcat(newSentence, " "),
} else if ((wordLength > 3) && (!strcmp(&word[wordLength - 4], "0Bun") ||
  !strcmp(&word[wordLength - 4], "1Bun") ||
  !strcmp(&word[wordLength - 4], "2Bun") ||
  !strcmp(&word[wordLength - 4], "3Bun") ||
```

```
   !strcmp(&word[wordLength - 4], "4Bun") ||
   !strcmp(&word[wordLength - 4], "5Bun") ||
   !strcmp(&word[wordLength - 4], "6Bun") ||
   !strcmp(&word[wordLength - 4], "7Bun") ||
   !strcmp(&word[wordLength - 4], "8Bun") ||
  !strcmp(&word[wordLength - 4], "9Bun"))) {
word[wordLength - 3] = '\0',
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "Bun"),
strcat(newSentence, " "),
}
else if ((wordLength > 4) && !strcmp(&word[wordLength - 4], "IEox")) {
word[wordLength - 4] = '\0',
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "I"),
strcat(newSentence, " "),
strcat(newSentence, "Eox"),
strcat(newSentence, " "),
}else if ((wordLength > 3) && !strcmp(&word[wordLength - 3], "Eox")) {
word[wordLength - 3] = '\0',
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "Eox"),
strcat(newSentence, " "),
} else if ((wordLength > 3) && !strcmp(&word[wordLength - 2], "Ix")) {
word[wordLength - 2] = '\0',
strcat(newSentence, word);
strcat(newSentence, " "),
strcat(newSentence, "Ix"),
strcat(newSentence, " "),
} else if ((wordLength > 2) && !strcmp(&word[wordLength - 2], "Go")
  && strcmp(word, "BoGo")
  && strcmp(word, "GeuRiGo")
                && strcmp(word, "ChamGo")) {
word[wordLength - 2] = '\0',
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "Go"),
```

```
strcat(newSentence, " ");
} else if ((wordLength > 2) && (!strcmp(&word[wordLength - 3], "0Il") ||
   !strcmp(&word[wordLength - 3], "1Il") ||
   !strcmp(&word[wordLength - 3], "2Il") ||
   !strcmp(&word[wordLength - 3], "3Il") ||
   !strcmp(&word[wordLength - 3], "4Il") ||
   !strcmp(&word[wordLength - 3], "5Il") ||
   !strcmp(&word[wordLength - 3], "6Il") ||
   !strcmp(&word[wordLength - 3], "7Il") ||
   !strcmp(&word[wordLength - 3], "8Il") ||
  !strcmp(&word[wordLength - 3], "9Il"))) {
word[wordLength - 2] = '\0',
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "Il"),
strcat(newSentence, " "),
} else if ((wordLength > 2) && (!strcmp(&word[wordLength - 3], "0Si") ||
   !strcmp(&word[wordLength - 3], "1Si") ||
   !strcmp(&word[wordLength - 3], "2Si") ||
   !strcmp(&word[wordLength - 3], "3Si") ||
   !strcmp(&word[wordLength - 3], "4Si") ||
   !strcmp(&word[wordLength - 3], "5Si") ||
   !strcmp(&word[wordLength - 3], "6Si") ||
   !strcmp(&word[wordLength - 3], "7Si") ||
   !strcmp(&word[wordLength - 3], "8Si") ||
  !strcmp(&word[wordLength - 3], "9Si"))) {
word[wordLength - 2] = '\0',
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "Si"),
strcat(newSentence, " "),
} else if ((wordLength > 2) &&
  !strcmp(&word[wordLength - 1], "I")
  && strcmp(word, "SaI")) {
word[wordLength - 1] = '\0';
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "I"),
strcat(newSentence, " "),
}else if ((wordLength > 2) &&
```

```
  !strcmp(&word[wordLength - 2], "Eo")
 && strcmp(word, "BaGguEo")) {
word[wordLength - 2] = '\0';
strcat(newSentence, word),
strcat(newSentence, " "),
strcat(newSentence, "Eo"),
strcat(newSentence, " "),
} else {
strcat(newSentence, word),
strcat(newSentence, " "),
}
}

free(strıngCopy),
return (strdup(newSentence)),
}
```

Bibliography

Eric Brill, "A Simple Rule-Based Part of Speech Tagger," Proceedings of the Third Conference on Applied Natural Language Processing, ACL, Trento, Italy, 31 March - 3 April, 1992, pp. 152-155

Eric Brill, "Transformation-Based Error-Driven Learning and Natural Language Processing: A Case Study in Part-of-Speech Tagging," Computational Linguistics 21 (4), 1996, pp 543-565

Bonnie Dorr, "LCS-BASED Korean Parsing and Translation," TCN No 95008, Institute for Advanced Computer Studies and Department of Computer Science, University of Maryland.

D Edegi, M. Palmer, H -S Park, A.-K Joshi, "Korean-to-English Translation Using Synchronous TAGs", Proceedings of the 1st Conference of the Association for Machine Translation in the Americas, Columbia, Maryland, October 1994, pp 48-55

V. Fromkin, R Rodman, "An Introduction to Language," Holt, Rinehart and Winston 1988

Jim Glass, Joe Polifroni and Stephanie Seneff, "Multilingual Language Generation across Multiple Domains," 1994 International Conference on Spoken Language Processing, Yokohama, Japan, 18-22 September 1994, pp. 983-986

W J Hutchins and H L Somers "An Introduction to Machine Translation," Academic, London, 1992

J -T Hwang, "A fragmentation technique for parsing complex sentences for machine translation," M Eng Thesis, MIT Dept of EECS, June 1997

H. B Ihm, K P. Hong, S I Chang, "Korean Grammar for International Learners," Yonsei University Press 1988.

Young-Suk Lee "Scrambling as Case-Driven Obligatory Movement", IRCS Report No : 93-06, University of Pennsylvania

Young-Suk Lee, Clifford J Weinstein, Stephanie Seneff, Dinesh Tummala "Ambiguity Resolution for Machine Translation of Telegraphic Messages," Proceedings of the Association for Computational Linguistics, Madrid, 7-12, July 1997

Samuel Martin "A Reference Grammar of Korean," Tuttle Language Library 1992.

H -S Park, "Korean Grammar Using TAGs," IRCS Report 94-28, Institute for Research in Cognitive Science, University of Pennsylvania 1994

Sung S Park, "Application of Part of Speech Tagger in Robust Machine Translation System", M Eng Thesis, MIT Dept of EECS, May1998

Stephanie Seneff "TINA A Natural Language System for Spoken Language Applications," Computational Linguistics 18 (1), 1992, pp 61-92

B M Sundheim "Plans for a Task-Oriented Evaluation of Natural Language Understanding Systems," Proceedings of DARPA Speech and Natural Language Workshop, Philadelphia, 21-23 February 1989, pp. 197-202

D Tummala, S Seneff, D Paul, C Weinstein, and D Yang, "CCLINC System Architecture and Concept Demonstration of Speech-to-Speech Translation for Limited- Domain Multilingual Applications," Proc 1995 ARPA Spoken Language Technology Workshop, Austin, Tex, 22-25 Jan 1995, pp 227-232

Clifford J. Weinstein, Young-Suk Lee, Stephanie Seneff, Dinesh R Tummala, Beth Carlson, John T Lynch, Jung-Taik Hwang, and Linda C Kukolich "Automated English-Korean Translation for Enhanced Coalition Communications," Lincoln Laboratory Journal Volume 10, Number 1, 1997, pp 35-60

D Yang, "Korean Language Generation in an Interlingua-Based Speech Translation System," Technical Report 1026, MIT Lincoln Laboratory, Lexington, Mass , 21 Feb 1996, DTIC#ADA-306658

THESIS PROCESSING SLIP

FIXED FIELD: ill _____ name _____

index _____ biblio _____

▶ COPIES: (Archives) Aero Dewey (Eng) Hum

Lindgren Music Rotch Science

TITLE VARIES: ▶ ☐ _____

NAME VARIES: ▶ ☑ Dong, Joon
middle name

IMPRINT: (COPYRIGHT) _____

▶ COLLATION: 67 l

▶ ADD DEGREE: [redacted] S.B. ▶ DEPT. E.E.

SUPERVISORS: _____

NOTES:

cat'r _____ date
page
▶ S10131

▶ DEPT: E.E.

▶ YEAR: 1998 ▶ DEGREE: M. Eng.

▶ NAME: KIM, Philip D.